CYCLE: Learning to Self-Refine the Code Generation

YANGRUIBO DING, Columbia University, USA
MARCUS J. MIN, Columbia University, USA
GAIL KAISER, Columbia University, USA
BAISHAKHI RAY, Columbia University, USA

Pre-trained code language models have achieved promising performance in code generation and improved the programming efficiency of human developers. However, their self-refinement capability is typically overlooked by the existing evaluations of code LMs, which focus only on the accuracy of the one-time prediction. For the cases when code LMs fail to implement the correct program, developers actually find it hard to debug and fix the faulty prediction since it is not written by the developers themselves. Unfortunately, our study reveals that code LMs cannot efficiently self-refine their faulty generations as well.

In this paper, we propose Cycle framework, learning to self-refine the faulty generation according to the available feedback, such as the execution results reported by the test suites. We evaluate Cycle on three popular code generation benchmarks, HumanEval, MBPP, and APPS. The results reveal that Cycle successfully maintains, sometimes improves, the quality of one-time code generation, while significantly improving the self-refinement capability of code LMs. We implement four variants of Cycle with varied numbers of parameters across 350M, 1B, 2B, and 3B, and the experiments show that Cycle consistently boosts the code generation performance, by up to 63.5%, across benchmarks and varied model sizes. We also notice that Cycle outperforms code LMs that have 3× more parameters in self-refinement.

CCS Concepts: • **Software and its engineering** → *Automatic programming*; *Functionality*.

Additional Key Words and Phrases: Code Language Models, Source Code Modeling, Code Generation, Iterative Programming

ACM Reference Format:
Yangruibo Ding, Marcus J. Min, Gail Kaiser, and Baishakhi Ray. 2024. CYCLE: Learning to Self-Refine the Code Generation. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 108 (April 2024), 27 pages. https://doi.org/10.1145/3649825

1 INTRODUCTION

Pre-trained code language models (code LMs) have achieved great success in code generation, and many of them have been deployed as a part of the integrated development environment (IDE), such as GitHub Copilot [GitHub 2021] and Amazon CodeWhisperer [Amazon 2023], to help human developers improve the programming efficiency. Along this direction, researchers started to conduct empirical and human studies to analyze the strengths and weaknesses of these models [Barke et al. 2023; Guo et al. 2023; Huang et al. 2023]. For example, Barke et al. [2023] propose a grounded theory of code-LM-assisted programming, systematically categorizing the interaction between code LMs and developers into two modes: *acceleration* and *exploration*. They define the *acceleration* mode as the situation when the developers clearly know what are the expected functionalities, code LMs

Authors' addresses: Yangruibo Ding, Columbia University, New York, USA, yrbding@cs.columbia.edu; Marcus J. Min, Columbia University, New York, USA, jm5025@columbia.edu; Gail Kaiser, Columbia University, New York, USA, kaiser@cs.columbia.edu; Baishakhi Ray, Columbia University, New York, USA, rayb@cs.columbia.edu.

ACM 2475-1421/2024/4-ART108
https://doi.org/10.1145/3649825

help them speed up the implementation. On the other hand, the *exploration* mode indicates the scenario when developers do not have concrete plans regarding how to proceed, such as facing some bugs reported by the test suites while developers have not figured out how to fix them. While this study identifies the efficiency of code LMs in the acceleration mode, it recognizes their limitations in exploration mode. For example, the code generated by code LMs in the exploration mode is less trusted than in the acceleration mode, and developers tend to validate them frequently with executions. Moreover, when errors are revealed through executions, developers struggle to debug model-generated code, as it was not authored by themselves, making it more challenging to understand the error within unfamiliar code.

In this paper, we propose Cycle framework, making an attempt to enhance the performance of code LMs in the exploration mode. The fundamental principle underpinning the creation of Cycle is the recognition that expecting code LMs to excel in the exploration mode, where human intentions are often unclear or not explicitly specified, may be overly demanding. However, these models should possess the capability to iteratively improve their code generation based on the feedback they receive from other sources such as execution results reported by test suites. In essence, Cycle aims to empower code LMs to adapt and enhance their output in response to the available feedback, thereby bridging the gap between human developers' exploratory programming needs and the capabilities of code LMs.

Limitations of code LMs in the exploration mode. In this work, we focus on the scenario of code generation that given the natural language (NL) description of a problem, typically wrapped in the docstring, the code LM will implement the program accordingly. For the convenience of our future discussion, we first concretize the *acceleration* mode and *exploration* mode in our scenario.

- Acceleration Mode: Given the NL description of a problem, code LMs directly predict the code accordingly.
- Exploration Mode: If the prediction from the acceleration mode fails the test cases and execution feedback returns, code LMs try to refine the faulty code without further human instructions.

As shown in Figure 1, we prompt GPT-3.5 [Ouyang et al. 2022] (gpt-3.5-turbo-instruct) with a problem description from HumanEval programming benchmark [Chen et al. 2021]. GPT-3.5 could not fulfill the functionality in the acceleration mode, and its generated code failed to pass the test suite of this problem. We could see from GPT-3.5's generation that, though the program is very short, spanning only 14 lines, it is not straightforward for humans to understand. The complexity mainly comes from the nested for-loops, making it even more difficult to manually identify and correct the error. Therefore, motivated by Chen et al. [2023], we tried concatenating the faulty generation and the execution feedback reported by the test suite, as additional references, with the problem description and expected the model to refine the generated code by itself in the exploration mode. Unfortunately, GPT-3.5 could not effectively understand the guidance from the execution feedback and simply copy-pasted the faulty code as its new prediction.

Such weakness of self-refinement in the exploration mode is even more severe in open-source code LMs. We conduct similar experiments with CodeGen (2.7 billion parameters) [Nijkamp et al. 2023b] and StarCoder (3 billion parameters) [Li et al. 2023] on the whole HumanEval benchmark with 164 programming problems. We observe that existing code LMs perform poorly in the exploration mode, failing to self-refine the faulty generations according to the execution feedback. CodeGen generates an exact copy of the faulty code as its refined prediction in 42.2% cases while StarCoder copies in 64.8% cases. Such weak self-refinement capability in the exploration mode is concerning, as it brings further burden to the human developers to fix the bugs brought by the model-generated code.

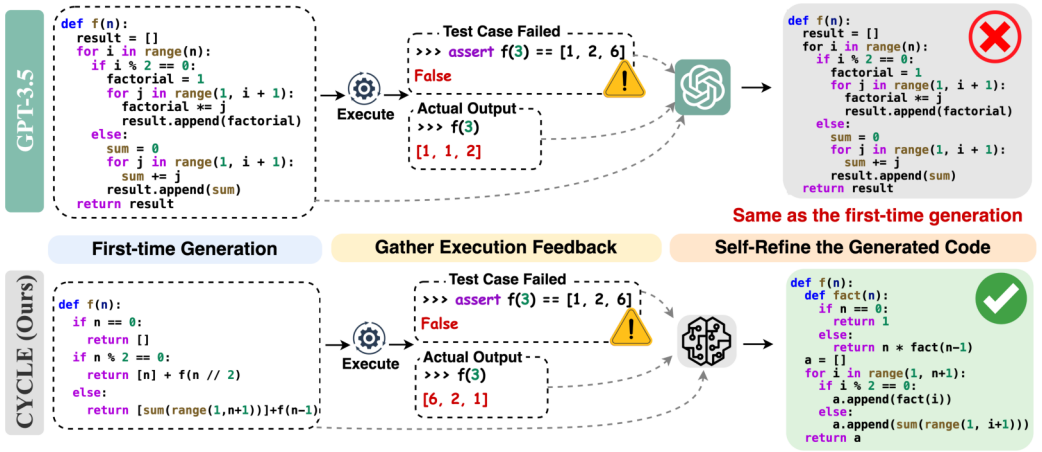


Fig. 1. Motivation Example. We prompt GPT-3.5 (gpt-3.5-turbo-instruct) and CYCLE to implement a program according to a problem description from the HumanEval programming benchmark (Task No. 106). While both failed to pass the test suite in the acceleration mode, CYCLE successfully refined its own generation referring to the execution feedback. In contrast, GPT-3.5 could not self-refine effectively.

Our Approach. In this work, we argue that code LMs should be enhanced in exploration Mode with self-refinement capability by leveraging the available feedback from the execution results. In fact, a model trained with such execution feedback has the potential to perform better even in the acceleration mode.

To this end, we design CYCLE, a framework that teaches code LMs to self-refine (by continuing training a pre-trained model) by jointly attending to three information sources: (i) the high-level problem description in natural language, (ii) the incorrect code the model may have generated in the previous attempts, and (iii) the execution feedback. We have developed an input template that consolidates these three information sources and employs them to train the code LM. While traditional code LMs' training primarily relies on the first information source, the inclusion of prior incorrect code aids the model in achieving a more comprehensive grasp of its own errors. The execution feedback, in turn, guides the code LM in generating programs that align precisely with the problem description.

Nonetheless, when we naively include the previously generated erroneous code in the input, the code LM often resorts to a shortcut, essentially copying from the incorrect input when generating new code. To deter the code LM from adopting such shortcuts, we employ a masking technique, referred to as the Past Generation Mask (PGM). This strategy slightly obfuscates the incorrect past generations, motivating the model to explore a more extensive range of solutions for code refinement. Additionally, to strike a balance between the proficiency of code generation in acceleration and exploration mode, we employ a data mixing strategy that manipulates the ratio of self-refinement features and general code completion features.

To efficiently train a Code LM using the above strategy we have to curate data that simulate the exploration mode development. Thus, we further design an automatic training data generation phase as existing pre-training code datasets [Kocetkov et al. 2022; Nijkamp et al. 2023b; Xu et al. 2022] are challenging to be customized for self-refinement training. Our data collection phase automatically prompts the pre-trained code LMs to reveal their own strengths and weaknesses in code generation, verified by executing test cases, and constructs data samples to reinforce their strengths while refining their weaknesses.

Finally, we implement Cycle to realize an automated self-refinement workflow that imitates the iterative programming practice of human developers. The workflow first prompts code LMs to initialize the implementation according to the high-level problem description and then continuously verifies the correctness of prediction with execution and aggregates comprehensive information for further refinement.

Results. We evaluate Cycle's code generation capability with three popular programming benchmarks: HumanEval [Chen et al. 2021], MBPP-Sanitized [Austin et al. 2021], and APPS [Hendrycks et al. 2021]. To illustrate the effectiveness and generalizability of Cycle, we train four variants of Cycle with varied parameter sizes ranging from 350M to 3B. From the evaluation results, we conclude that Cycle is pretty effective at self-refinement, consistently boosting the code generation performance, by up to 63.5% relative improvement, across four model sizes on all three benchmarks, while maintaining decent one-time generation capacity. With efficient self-refinement learning, Cycle-350M outperforms StarCoder-1B across all three programming benchmarks, and Cycle-1B matches the performance of StarCoder-3B. With the in-depth analysis, we also empirically reveal that Cycle is effective at capturing execution feedback and has great potential to assist human developers with iterative programming.

Novelty and Contributions. We make the following novel contributions:

- Our work sheds light on the weaknesses of code LMs in self-refinement, revealing that these models are not effective at understanding the execution feedback and correcting their own mistakes accordingly.
- To fulfill the code LMs' potential in self-refinement, we propose Cycle, a framework that enhances the code LMs' generation performance by learning to refine their own generated code. We first propose a knowledge-distillation-based data collection approach to automatically construct samples to teach code LMs to self-refine. We then propose a training strategy designed specifically for learning self-refinement. Finally, we implement an iterative self-refinement workflow that automates the process of generating code in exploration mode.
- We conduct extensive experiments on three popular code generation benchmarks with four Cycle variants across 350M to 3B model parameters, and show that Cycle consistently increases the code generation performance by up to 63.5%. Cycle could also match or even outperform baseline code LMs with 3× parameters.
- We perform in-depth analysis to discuss Cycle's design and performance from multiple perspectives. We conclude with insights and takeaways to motivate further research in improving code LM's self-refinement capability, which hopefully assists human developers with iterative programming and generally increases code LM's performance in exploration mode.

We anonymously release our code, data, and model checkpoints. The artifact is available at https://github.com/ARiSE-Lab/CYCLE_OOPSLA_24.

2 OVERVIEW

In this section, we briefly introduce Cycle, explaining the high-level designs and the intuitions behind them. We present the overview of Cycle in Figure 2. At a high level, Cycle contains three phases.

Phase-I: Data Preparation for Self-Refinement. The data to train code LMs for refinement capability requires carefully crafted features, such as the developing log with iterative correction of code snippets and their error-exposing feedback loops. These features do not naturally come along with the large-scale pre-training datasets [Kocetkov et al. 2022; Nijkamp et al. 2023b; Xu et al. 2022]

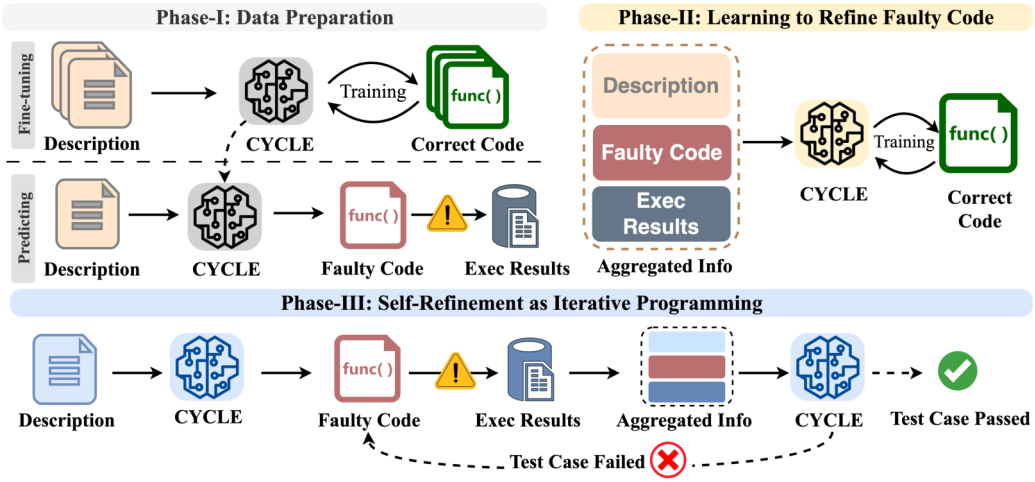


Fig. 2. Overview of CYCLE.

that typically prioritize data quantity while not providing interfaces for customization. Therefore, we propose an automated approach to distill such features from the pre-trained code LMs and construct the datasets on top of them.

The general idea is to prompt the pre-trained code LMs with programming problems, asking them to generate code to fulfill the requested functionalities. These problems should be well-defined and accompanied by test suites and canonical solutions so that the model's generation can be efficiently verified. When the code LM makes mistakes, we gather its faulty generations and the corresponding execution feedback reported by the test suites. Then we construct data samples accordingly, which could be leveraged to teach code LMs to correct their own mistakes by referring to the execution feedback and the canonical solutions accompanied by the programming challenges.

Fine-tuning Code LMs with Correct Code. Code LMs are pre-trained with up to trillions of code tokens, but the noise inherited in the pre-training data could trigger unexpected predictions [Li et al. 2022] or even vulnerable code [He and Vechev 2023]. We should avoid these malicious behaviors when constructing samples for self-refinement. The unexpected predictions might reveal random errors and could not get meaningful feedback after execution. Also, when the vulnerable code is generated, it might bring a security threat to the execution system. Therefore, we propose to first fine-tune the pre-trained code LMs on the correct code to minimize the model's malicious behaviors. Concretely, we collect a decent number of programming challenges that are accompanied by canonical solutions, and we fine-tune the pre-trained code LMs to predict these guaranteed correct programs conditioned on the problem description.

Prompt Code LMs to Distill the Weaknesses. With the fine-tuning on the canonical solutions, we expect the model to make faulty predictions that are not too off from the correct code. We prompt the fine-tuned model with the problem description and then execute its prediction with the accompanied test suites. The execution results with error messages expose the models' strengths and weaknesses in code generation, which are collected and saved as resources to build training samples to learn the refinement.

Phase-II: Learning to Refine the Faulty Generation. With the data prepared by Phase-I, we construct the training samples to learn code refinement based on three types of information. Specifically, we aggregate the problem description, the faulty code generated by the code LM, and the corresponding execution results, using our proposed template (more details in Section 3.2.1), to formulate the

model input, so that the model could jointly attend to comprehensive information simultaneously. Then we use the accompanying canonical solutions as the target for the model to predict. Different from the fine-tuning in Phase-I, which only teaches the model to fulfill the functionalities described in natural language, the code refinement training in this phase exposes the model to its own mistakes and the execution feedback, together with the problem description, which forces the model to both reason about the misalignment between its generation in the past and the problem description and learn to understand the implicit guidance in the execution results. We carefully design our training strategy to effectively and efficiently learn the code refinement, and we will introduce more details of our approach in Section 3.2.

Phase-III: Self-Refinement as Iterative Programming. After learning about the code refinement from Phase-II, we deploy the model to automatically generate code according to the problem description and, similar to the iterative programming practice of human developers, iteratively refine the code to fulfill the required functionality. When the problem description is fed into the model, the model will first generate the code at its best, and the generation will automatically be executed with the test suite. If failed test cases are detected, our framework will automatically aggregate the description, the faulty code, and the execution feedback using the template proposed in Phase-II. Finally, the aggregated information will be again fed into the model for self-refinement.

3 APPROACH

In this section, we explain CYCLE in detail. We will illustrate the concrete approaches we designed to teach the code language models (code LMs) about self-refinement for code generation.

3.1 Phase-I: Data Preparation

Training code LMs to possess self-refinement capabilities demands a dataset rich in specific features, such as development logs showcasing the iterative correction of code snippets, as well as feedback loops that highlight the errors. Existing large-scale pre-training datasets [Kocetkov et al. 2022; Nijkamp et al. 2023b; Xu et al. 2022] are primarily designed to amass vast amounts of data but are not inherently equipped to offer these nuanced features. Therefore, an efficient data collection method is required to extract these specialized features. To this end, we propose a knowledge distillation [West et al. 2022] approach that prompts the pre-trained code LMs to showcase their capabilities, revealing their strength as well as exposing their weaknesses. Subsequently, we will construct the datasets to reinforce their strengths while learning to self-refine their weaknesses.

3.1.1 Fine-tune Code LMs with Semantically Correct Code. Code language models [Chen et al. 2021; Li et al. 2023; Nijkamp et al. 2023b; Rozière et al. 2023] are typically pre-trained with a huge amount of source code, learning to predict up to trillions of code tokens. However, the innate noise in the pre-training data could affect its accuracy in code generation. As the large corpora are mostly collected from open-source resources [Kocetkov et al. 2022; Rozière et al. 2023; Xu et al. 2022], the quality of training samples varies significantly. Specifically, the high-quality code snippets well align with their accompanying natural-language docstrings or comments, such as those from mature and actively maintained projects or forked commercial software, while a certain amount of the noisy snippets have vulnerable functionalities and semantic misalignment, such as those from developing projects or starter-level programmers. This means that during training, the model is exposed to, and possibly memorizes, both correct (aligned) and incorrect (misaligned) programs. The consequential challenge during the knowledge distillation is that, given a natural language (NL) prompt, the model holds the potential to generate either semantically accurate or erroneous code. This behavior is generally determined by the frequency ratio of correct to incorrect code observed

during pre-training. It has been verified by existing works that the noise in the pre-training data could bring unexpected behavior [Li et al. 2022] or security vulnerabilities [He and Vechev 2023].

To minimize the malicious behaviors of code LMs during the knowledge distillation, we first fine-tune the pre-trained code LMs using only "guaranteed correct" code. Specifically, we collect canonical solutions from programming challenges that are already verified by the test suite and fine-tune code LMs to predict these solutions token by token. Different from the pre-training of code LMs [Li et al. 2023; Nijkamp et al. 2023a,b; Rozière et al. 2023] that predicts every token in the corpora, our fine-tuning is designed to only limited to the code tokens within the canonical solution and the NL description will be regarded only as context. Specifically, the NL description is a sequence of m tokens, $NL = \{nl_0, nl_1, ..., nl_m\}$, and the canonical solution is a sequence of n tokens, $C = \{c_0, c_1, ..., c_n\}$, we apply the standard language modeling loss [Brown et al. 2020; Radford and Narasimhan 2018; Radford et al. 2019] to optimize the fine-tuning:

$$\mathcal{L}_{fine-tune} = \sum_{n \in |C|} -logP(c_n \mid NL, c_1, c_2, ..., c_{n-1}) \tag{1}$$

Such a design makes the model's prediction more focused, and the learning process is more narrowed to only optimize the model towards the code generation. We observe that, after this fine-tuning, the model not only generates better code quality but also knows when to terminate the generation more accurately than the pre-trained code LMs without such fine-tuning. This is mainly because the pre-training asks code LMs to predict tokens until the maximum context length is reached, while we fine-tune the model to focus on the canonical solutions that are terminated naturally when the functionality is complete.

3.1.2 Prompt Code LMs to Distill the Weaknesses. After the fine-tuning, the model is primed to generate solutions that, even if faulty, are proximate to canonical solutions. This leaves us great chances to create a dataset with a decent amount of paired samples. Such a pair, including a correct and a wrong solution targeting the same problem, is quite valuable, revealing through error messages the subtle nuances where the model's code generation aligns or diverges from the desired outcome. These errors aren't merely mistakes; they provide insights into the model's comprehension and interpretation, shedding light on its strengths and weaknesses. We hope to teach code LMs to capture and learn to transform the wrong code to its correct counterpart by fixing the subtle error according to the execution feedback. Such a transformation well imitates the process of self-refinement.

To construct such samples, we prompt code LMs with problem descriptions and verify their predictions using the accompanying test suites. For those problems that the model fails to predict correctly, we gather its faulty generation and the errors thrown by the test suite execution. We will pair this fault with the ground-truth implementation to construct training samples for the next phase of learning. For the correctly predicted problem, we will also gather its generation to replace the canonical solution of this problem. For the next phase of learning, we will train the model to predict its own generated correct code to maintain and reinforce such knowledge.

3.2 Phase-II: Learning to Refine the Faulty Generation

With the constructed samples in the previous phase, we introduce the training process of CYCLE, which is designed for learning to refine the faulty code.

CYCLE will be firstly initialized with the fine-tuned checkpoints that we introduced in Section 3.1.1, and then continue training with the samples described in Section 3.1.2. As these samples include the errors made by the fine-tuned code LM, teaching the same model about the code refinement

aligns with our final goal that we expect the code LMs to refine the faulty code generated by itself, *i.e.,* "self"-refinement.

3.2.1 Aggregate the Problem Description, Faulty Generation, and Execution Results as a Joint Prior Condition. To effectively teach code LMs about code refinement, we propose a template that aggregates information from multiple resources. We show the template in Figure 3. To ensure the code's naturalness remains intact, we adopt a strategy of encapsulating our template within docstrings or comments. This template comprises six essential components.

First, we encapsulate the problem description within docstrings. Next, we introduce a negative prefix that signifies the beginning of a flawed code generation, followed by the actual erroneous code. In a manner similar to the negative prefix, we employ an execution prefix to precede the feedback regarding the code's execution. This feedback meticulously outlines the test results, encompassing specific failed test cases, input data, the anticipated output, and the actual misaligned output. Finally, our template concludes with a positive prefix designed to prompt the generation of the correct code solution.

```
def f(n):
    """Implement the function f that
    takes n as a parameter...
    """                                          Problem Description (Prompt)
    ### Bad Solution                             Negative Prefix
    # def f(n):                                  Faulty Generation
    #     if n == 0:
    #         return []
    #     if n % 2 == 0:
    #         return [n] + f(n // 2)
    #     else:
    #         return [sum(range(1,n+1))]+f(n-1)
    ### Execution Failure 0                      Feedback Prefix
    # Test Case Failed: assert f(3) == [1, 2, 6]
    # Input: 3
    # Expected Output: [1, 2, 6]
    # Actual Output: [6, 2, 1]                   Execution Feedback
    ### Good Solution:                           Positive Prefix
```

Fig. 3. Template to aggregate the problem description, faulty generation, and the execution feedback.

3.2.2 Learning to Self-Refine. With the proposed template above, we efficiently aggregate all information together for the model to jointly attend to. To learn the refinement, the model will be fed with the aggregated information and learn to predict the canonical solution of this problem. Specifically, we define the aggregated information, at a high level, as the combination of problem description, NL, faulty generation, FG, and the execution feedback, EF, so $AGGR = \{NL, FG, EF\}$. The target canonical solution, C will be predicted token by token, and thus the loss can be represented as:

$$\mathcal{L}_{self-refine} = \sum_{n \in |C|} -logP(c_n \mid NL, FG, EF, c_1, c_2, ..., c_{n-1}) \tag{2}$$

Though the learning process is mostly about the next token prediction [Radford and Narasimhan 2018; Radford et al. 2019], it is quite effective at learning the self-refinement, which will be illustrated in Section 5.1. The main reason for its effectiveness is that the learning objective forces the model to learn different knowledge from distinct resources of information. To correctly predict each code token c_i, the model needs to understand NL, FG, and EF respectively, and jointly decide how to take advantage of each resource.

Past Generation Mask (PGM) When aiming to refine faulty code generations, there is a risk that, during the training process, the model pays too much attention to the prior faulty generation, *i.e., FG*, it might end up taking shortcuts by copying the tokens to minimize the self-refinement loss mentioned above, since the faulty code has significant overlap, in terms of tokens, with the target implementation (Section 3.1.2). Instead of genuinely understanding and rectifying the code

based on the feedback, the model could resort to merely copying or reproducing large portions of the past generation. Such shortcuts are difficult to detect during the training, as the loss will not significantly differ regardless of model takes the shortcut or not, but the model relying on the shortcuts becomes useless when being deployed.

To alleviate such risk, we are motivated by the efficient masking approaches in deep learning literature, such as dropout [Srivastava et al. 2014] and forgetful causal mask [Liu et al. 2023], and we propose to randomly mask $p\%$ of tokens in the faulty generation FG, which is the main resource for copying. Concretely, we manipulate the attention mask in the Transformer architecture [Vaswani et al. 2017], which will make the masked token invisible to other tokens without the explicit removal of tokens. We will study the effectiveness of this masking strategy in Section 5.2. Note that the masking ratio, p, is a tunable hyperparameter, and we will study its impact in Section 5.4.

3.2.3 Mixture of Data Resources. Data is always the key to the success of language model training, and existing code LMs [Li et al. 2023; Rozière et al. 2023] have illustrated the necessity of combining multiple data resources with carefully tuned proportions. In the quest to enhance the proficiency of our model for self-refinement, we identified the necessity to curate an optimal mix of self-refine samples, as described in Section 3.1.2, and the canonical solutions used in Section 3.1.1. The rationale behind this strategy is to balance Cycle's capability in acceleration and exploration mode.

The proportion of these data resources also introduces a nuanced layer of complexity to the training regimen. An imbalance, especially an excessive reliance on self-refine samples, could lead the model to overfit code refinement. In such a scenario, it expects, and perhaps becomes overly dependent on, the trifecta of the natural language prompt, the prior faulty generation, and the execution feedback. This could compromise its innate capacity to generate code accurately in situations where only a simple prompt is provided. Conversely, if the original code samples dominate the training set, the model might not sufficiently internalize the mechanisms of self-correction and iterative refinement. This intricate balance underscores the importance of meticulous data engineering in training a versatile and robust code language model. We will study the impact of such data mixture in Section 5.4

3.3 Phase-III: Iterative Self-Refinement with Execution Feedback

While the Phase-II training is formulated as a one-step refinement from the faulty code to its correct version, we implement the automated inference framework as an iterative programming workflow. There are three main reasons. First, we assume that one-step refinement is not enough to fulfill the model's best capacity, as there is always a gap between the perfect performance that the training tries to approach and the final inference performance that could really be achieved. Second, the diversity of self-refinement samples should enable Cycle with the iterative capability of refinement Third, our design is to harmonize the automated code generation with human-like introspection and iterative improvement. By incorporating feedback loops, setting clear stop criteria, and leveraging the structured aggregation from Phase-II, we aim to transform code generation from a one-time event to a dynamic, self-evolving process. This methodology, inspired by human programming practices, sets the stage for more resilient, accurate, and context-aware code outputs.

Specifically, as shown in Figure 2, upon receiving the problem description, the model initially produces what it perceives as the optimal code. This generated code is then automatically tested against the relevant test suite. Should any test cases fail, our framework seamlessly compiles the original description, the incorrect code, and the associated execution feedback, all according to the template from Phase-II. This consolidated input is then presented back to the model, guiding its subsequent refinement efforts. The process of self-refinement is not endless considering the overhead and computation it might cost, and it will be stopped by three scenarios: (1) if the refined

code successfully passes the test suite, or (2) if a threshold for the maximum self-refine times is reached, or (3) if the refined code remains the same as the previous faulty code, which means the model could no more improve the code according to the execution feedback.

4 EXPERIMENTAL SETUP

4.1 Training Datasets

We formulate the training dataset for Cycle with samples collected from CodeContest [Li et al. 2022]. CodeContest includes programming competition problems curated from Codeforces [Mirzayanov 2020] and CodeNet [Puri et al. 2021]. Each problem is accompanied by both canonical solutions and wrong human submissions written in three programming languages (Python, Java, C++), as well as executable test cases to verify the correctness of solutions.

To construct our training datasets, we focus on the training split of CodeContest which originally contained 13,328 problems. We further filter out those problems with long descriptions (> 512 tokens), as they could take too much input length of Transformer-based LM and leave insufficient length to include the code solutions that Cycle will be trained to predict. In addition, we remove the problems without Python solutions. Finally, our training set ended up with 7,108 problems from CodeContest. We also constructed a held-out validation set from CodeContest valid split, which originally contained 117 problems, to monitor the training process and the model's generalizability to unseen data during training. After the same filtering as our training set, we keep 77 problems in the validation set.

To fine-tune the code LMs for the data preparation phase (Section 3.1.1), we sample up to 50 canonical solutions for each of the problems, resulting in 233,703 training samples and 3,833 validation samples. To build the samples for the self-refinement training (Section 3.1.2), we sample 10 generations for each problem from the fine-tuned code LMs and execute the test suites to verify their correctness and collect the execution feedback. In total, we have 71,080 training samples and 770 validation samples for the self-refinement learning phase.

4.2 Evaluation Benchmarks

To evaluate the code generation performance, we use three popular programming benchmarks: HumanEval [Chen et al. 2021], MBPP-S [Austin et al. 2021], and APPS [Hendrycks et al. 2021].

HumanEval HumanEval contains 164 hand-written programming problems by Chen et al. [2021]. Each problem includes a function signature and docstring as the description, and the model is asked to predict the function body. Each problem also maintains on average 7.7 unit test cases, in the format of assertions, to verify the correctness of the model prediction.

MBPP-S Mostly Basic Programming Problems (MBPP) benchmark contains 974 short Python programs constructed by crowd-sourcing a pool of crowdworkers who only have basic knowledge of Python. Each problem contains a short problem statement and provides 3.0 unit test cases to verify the functionality. Austin et al. [2021] further sanitized the benchmark by manually inspecting and removing the ambiguous or unexpected problems, resulting in a version called MBPP-Sanitized (or MBPP-S for short) with a total of 426 problems of better quality. In our evaluation, we apply this sanitized version of MBPP-S for more accurate evaluation.

APPS APPS is a benchmark collected to evaluate the code generation models. It includes a total of 10,000 problems collected from open-access coding websites, accompanied by a total of 131,777 test cases and 232,421 canonical solutions. To avoid data leakage, we focus on the test split with 5,000 problems. In addition, given that APPS has significantly overlapped data resources with our training data from CodeContest, we apply an aggressive filtering strategy to deduplicate. Specifically, we exhaustively calculate the fuzzy edit similarity between the problem description in APPS and in

CodeContest, and if the similarity is over 60%, we will remove the whole problem from APPS to avoid the data memorization issue. Finally, after the filters, we ended up with 1,280 problems as our evaluation problem set.

4.3 Models

To illustrate the generalizability of Cycle's design, we train four variants of Cycle with varied sizes of model parameters: 350M, 1B, 2.7B, and 3B. These variants are initialized from the checkpoints of two open-source code LMs families: CodeGen [Nijkamp et al. 2023b] and StarCoder [Li et al. 2023].
CodeGen is a set of autoregressive code LMs, with varied sizes (350M to 16B), pre-trained using next-token prediction objective on a large-scale dataset collected from ThePile [Gao et al. 2021], Google BigQuery, and BigPython [Nijkamp et al. 2023b]. The dataset includes both natural language and code samples with over 550 billion tokens. The model maintains the context length of 2,048 BPE [Kudo and Richardson 2018] tokens.
StarCoder is a set of code LMs, with varied sizes (1B to 15.5B), that are pre-trained on over 1 trillion tokens from The Stack [Kocetkov et al. 2022] dataset, using both next-token prediction and the fill-in-the-middle [Bavarian et al. 2022; Fried et al. 2023] objectives. StarCoder family could consume up to 8,192 BPE tokens.

Specifically, Cycle variants are initialized from CodeGen-350M, StarCoder-1B, CodeGen-2.7B, and StarCoder-3B respectively. We load the checkpoints from the Hugging Face Model Hub [HuggingFace 2023].

4.4 Configurations and Hyperparameters

We conduct our experiments on 2× NVIDIA GeForce RTX 3090 with 24GB GPU memory each. The model is implemented mainly with PyTorch [*et al.* 2019] and Hugging Face Transformers [*et al.* 2020] library.

For training, we consider a batch size of 512 samples with 2,048 BPE [Kudo and Richardson 2018] tokens. We apply a standard learning rate descending strategy of code LM [Li et al. 2023; Nijkamp et al. 2023b; Rozière et al. 2023] that the early phase of training applies a higher learning rate than the later phase, and small models use a higher learning rate than large models. Concretely, for the fine-tuning in the data preparation phase (Section 3.1), Cycle uses [5e-5, 2e-5, 1e-5, 1e-5] for [350M, 1B, 2.7B, 3B] respectively, and for the self-refinement learning phase (Section 3.2), Cycle uses [2e-5, 2e-5, 5e-6, 5e-6] for the aforementioned model sizes respectively. All the training applies a cosine learning rate decay scheduler with warmup steps. For both the fine-tuning in the data preparation phase and the self-refinement learning, we train Cycle for only one epoch on the corresponding dataset. The PGM masking rate (Section 3.2.2) for training is 0.05, and the ratio of self-refinement samples is 25% (Section 3.2.3).

For inference, we adapt the standard nucleus sampling [Holtzman et al. 2020] with the top-p probability of 0.95. For HumanEval and MBPP-S benchmarks, we ask the model to generate up to 256 BPE tokens, and for the APPS benchmark, the model will generate up to 512 BPE tokens, as the latter's problem is more difficult and the solutions are typically longer. For the self-refinement process during the inference, we set up a maximal refinement step of 4, and this choice is based on the tradeoff between the inference overhead and the performance. We will study more about the number of refinement steps in Section 5.2.1 and the inference overhead in Section 5.3.4

5 EVALUATION

In this section, we evaluate Cycle and analyze it by asking the following research questions:

- RQ1: How effective is Cycle in code generation compared to the existing code LMs?

- RQ2: How do Cycle's different designs contribute to its performance?
- RQ3: How is Cycle's iterative self-refinement different from Top-K generations?
- RQ4: How will PGM's masking ratio and data mixture proportion affect the performance?

5.1 RQ1. Cycle's Performance in Code Generation

In this section, we present the main evaluation regarding Cycle's performance in code generation, and we illustrate its effectiveness by comparing it to existing code LMs of varied size, across three popular programming benchmarks.

Setup. We evaluate the model's code generation capability in two main settings: one-time generation and iterative self-refinement. For the one-time generation, we follow the original design of the evaluation benchmarks [Austin et al. 2021; Chen et al. 2021; Hendrycks et al. 2021], where the description of programming challenges will be fed into the model as the prompt, and the model will implement the program accordingly. We use the accompanied test suites of each programming challenge to verify the correctness of the prediction, where a test suite includes multiple test cases to evaluate the code with distinct perspectives.

For the iterative self-refinement performance evaluation, we follow the workflow of Cycle's inference framework (Section 3.3). The model will take the one-time generation as the starting point for the refinement, and repeat the process up to four times (Section 4.4). If the generated code passes the test suite at a time step, the self-refinement of this sample will be terminated, and this sample will be regarded as a success. If the model cannot predict a correct program with four times of self-refinement, the sample will be regarded as a failure.

Baselines. We consider vanilla open-source code LMs from CodeGen [Nijkamp et al. 2023b] and StarCoder [Li et al. 2023] families as the first set of baseline. Specifically, we consider CodeGen-350M, StarCoder-1B, CodeGen-2.7B, and StarCoder-3B. CodeGen and StarCoder are two of the most popular code LMs with state-of-the-art code generation capabilities, and since Cycle variants are initialized from these models to continue learning the self-refinement (Section 3.1.1), comparing with them will directly reflect the effectiveness of Cycle's proposed training.

In addition, we create a stronger set of baselines from the fine-tuned checkpoints we introduced in Section 3.1.1. We further train these checkpoints with the canonical solution from Cycle's training datasets but no self-refinement signals are included. This set of baselines has two main improvements compared to the vanilla code LMs. First, it is trained with the same data samples with the same amount of epochs as Cycle, which directly verifies whether the naive training on the canonical solution is enough to grant the model self-refine capability. Second, this set of baseline is trained with canonical solutions only, so comparing to it illustrates the value of Cycle's self-refinement training that learns to jointly understand the faulty generation and execution feedback as well. We call this set of baselines "code LMs fine-tuned with correct code".

Note that, since baseline models have never been exposed to the template we designed for Cycle to perform self-refinement (Section 3.2.1), we notice that applying such a template to baseline models will hurt their performance by misleading the model to keep generating comments rather than the real code. Alternatively, to maximize their performance for a fair comparison, we borrow the idea from existing work [Chen et al. 2023] to wrap the faulty code and the execution feedback into the docstring as plain text, and it turns out that the baseline models could normally generate code after this adaption.

Findings. The results of one-time code generation and self-refinement across four sizes of code LMs (350M, 1B, 2.7B, 3B) are reported in Table 1.

Finding-1.1: Self-Refinement Capacity Does Not Come Along Naturally with Code LMs' Pre-training. As we can see from the table, baseline code LMs, though pre-trained on tons of code samples,

Table 1. Comparing CYCLE's performance with baseline models in both one-time generation and iterative self-refinement settings. We consider four sizes of code LMs to discuss, ranging from 350M to 3B. The first row of each section is the "vanilla code LM" baseline, the second row of each section is the "code LMs fine-tuned with correct code" baseline, and the last row is CYCLE variant of the same size.

Model	One-time			Self-Refine		
	HumanEval	MBPP-S	APPS	HumanEval	MBPP-S	APPS
CodeGen–350M	12.2	19.0	6.9	12.2 +0.0%	21.8 +14.8%	6.9 +0.0%
+ FT w/ Correct	12.2	19.2	**7.5**	14.0 +14.9%	23.0 +19.5%	7.7 +2.8%
CYCLE–350M	**14.0**	**19.9**	**7.5**	**20.7** +47.9%	**32.6** +63.5%	**8.7** +15.6%
StarCoder–1B	15.9	25.8	7.3	16.5 +3.8%	28.1 +9.1%	7.3 +0.0%
+ FT w/ Correct	**18.3**	**26.0**	8.6	18.9 +3.3%	28.3 +9.0%	9.3 +8.3%
CYCLE–1B	**18.3**	25.8	**8.9**	**22.0** +20.0%	**35.8** +39.1%	**10.9** +22.8%
CodeGen–2.7B	**21.9**	34.7	7.1	23.8 +8.4%	35.4 +2.0%	7.1 +0.0%
+ FT w/ Correct	**21.9**	**36.8**	9.0	23.8 +8.4%	40.5 +10.2%	9.7 +7.9%
CYCLE–2.7B	21.4	35.8	**9.1**	**29.3** +37.1%	**48.5** +35.3%	**11.6** +27.6%
StarCoder–3B	23.8	35.1	7.3	26.8 +12.8%	40.5 +15.3%	7.4 +1.1%
+ FT w/ Correct	**24.4**	35.8	**9.0**	24.4 +0.0%	40.8 +13.7%	10.2 +13.9%
CYCLE–3B	**24.4**	**36.3**	**9.0**	**29.9** +22.5%	**51.3** +41.3%	**11.3** +25.3%

primarily focus on one-time code generation and struggle to self-refine the faulty generation in the past by understanding the execution feedback. It is also evident that, though the one-time code generation performance aligns with the scaling law of neural language models [Kaplan et al. 2020], the increase in model sizes does not necessarily translate to better self-refinement capabilities[1]. For example, baseline code LMs with sizes ranging from 350M to 2.7B all fail to correctly self-refine a single example in APPS (0.0% improvement after performing self-refinement). This verifies our assumption that existing code LMs are not exposed to sufficient signals during the pre-training to learn how to self-refine, and naively stacking more model parameters or collecting more data from the wild is not that helpful.

We notice that fine-tuning the baseline code LMs with only semantically correct code, as indicated by the "+ FT w/ Correct" rows, shows only marginal improvements in both one-time generation and self-refinement. This means that merely training on correct code is not enough to equip the models with the skill of rectifying their own mistakes, which requires a decent understanding of the execution feedback and the capability of fixing the errors accordingly.

Finding-1.2: CYCLE is Effective at Improving Code LM's Self-Refinement Capacity and Generalizable to Varied Model Sizes. As shown in Table 1, across three programming benchmarks, CYCLE consistently levels up the code generation performance with self-refinement, resulting in up to 63.5% relative improvement compared to one-time generation. CYCLE with 350M parameters notably outperforms StarCoder-1B, which maintains 3× more parameters, across all three benchmarks, highlighting the effectiveness of CYCLE's self-refinement and the efficiency of the proposed training strategy.

Compared to baseline models, CYCLE has notably stronger capabilities at correcting faulty generations in the past. For example, while the vanilla CodeGen-2.7B model could not refine its own wrong predictions in APPS, while CYCLE-2.7B successfully refines 27.6% more samples which it failed to predict for the first time. This suggests that, by training to refine code based on execution

[1]Note that our finding is concluded based on the code LMs with sizes between 350M and 3B parameters. We did not study the emergent ability of significantly larger models (*e.g.*, with 175B and 540B parameters) [Wei et al. 2022b].

feedback and previous mistakes, CYCLE builds a more holistic understanding of code, enabling it to not just generate code, but also understand its intricacies, potential pitfalls, and nuances.

In addition, as we introduced in Section 3, Cycle does not require to be trained from scratch, and rather, it loads pre-trained code LMs as the starting point and further teaches the model to self-refine. As we can see from Table 1, Cycle is effective at varied sizes of code LMs and consistently boost the performance for two different neural architectures[2], CodeGen and StarCoder. This empirically proves the generalizability of Cycle that it can always be applied in a plug-and-play style, suggesting the potential of taking advantage of more powerful code LMs that will be trained and released in the future.

Finding-1.3: Cycle Maintains Decent Capacity in One-time Code Generation. One recognized risk of continuing training unsupervised/self-supervised models, which are pre-trained with large-scale data, on limited, carefully crafted samples is the *model shift* problem [Wang et al. 2021; Wang and Schneider 2015]. The model could shift towards the local optimal of the limited but new data while the previous knowledge is eventually wiped out. As we can see in Table 1, Cycle does not suffer this issue; it successfully maintains the one-time generation capability. This empirically verifies that our proposed alignment of prompt, fault generation, and execution feedback (Section 3.2.1) is intuitive to the model, preventing the significant distribution shift. Also, the data mixture strategy (Section 3.2.3) plays a role in neutralizing the distribution gap, and we will further analyze this strategy in Section 5.4. Interestingly, the 350M, 2.7B, and 3B versions of Cycle perform slightly better than the standard baseline code LM. We believe the primary reason for this is the way Cycle is trained. During its training process, Cycle is exposed to both faulty and correct code, while our specialized training method encourages the model to only generate the correct one(Section 3.2.2). This gives Cycle a unique advantage: it understands what good and bad code look like respectively, but it's specifically trained to produce the good one. In comparison, baseline code LMs are not exposed to such preference signals during their pre-training.

Result-1: While maintaining the one-time code generation capacity, mostly improving marginally, Cycle significantly boosts the code LMs' self-refine capacity by learning to understand the execution feedback and faulty code generated in the past. Cycle enables existing code LMs to match or beat larger models with 3× more parameters.

5.2 RQ2. Impacts of Cycle's Different Designs on Code Generation

While we have concluded that Cycle's proposed approach, as a whole, is effective at code generation in Section 5.1, now we delve deeper to analyze how the isolated design contributes to the performance individually.

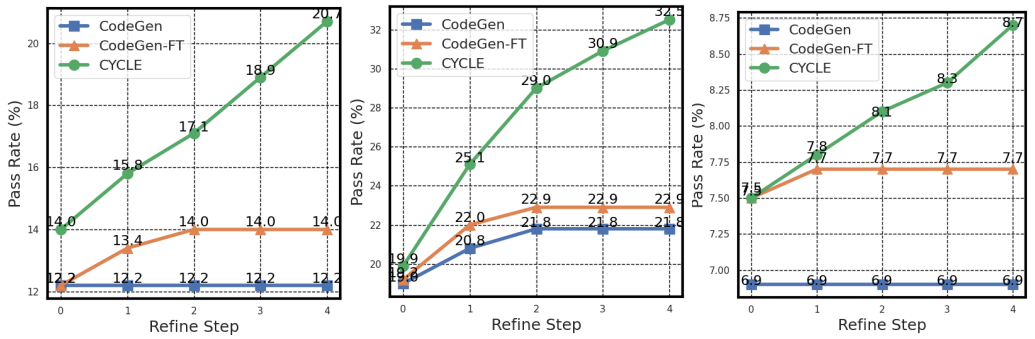
5.2.1 Self-Refinement Continuously Improves Cycle's Performance. Teaching code LMs to self-refine is the core idea behind Cycle's design, so we first study how self-refinement boosts Cycle's code generation capability. To do this, we plot the Cycle-350M's performance at each refinement step, in Figure 4, to reveal the trend. We also plot the baseline models' trends as a comparison.

We can see that Cycle's performance is improved with each refinement step across all three benchmarks. This reveals that Cycle is able to eventually correct its own error, step by step, by understanding the mismatch between the expected dynamic behavior and real implementations

[2]At a high level, CodeGen and StarCoder are both GPT-like Transformer decoder, but their concrete neural architectures (*e.g.*, positional embedding and attention layers) are not the same. More details can be referred from: StarCoder (https://github.com/huggingface/transformers/blob/main/src/transformers/models/gpt_bigcode/modeling_gpt_bigcode.py) and CodeGen (https://github.com/huggingface/transformers/blob/main/src/transformers/models/codegen/modeling_codegen.py)

from the execution feedback. In addition, the figure reveals that CYCLE still does not reach its best performance after four times of refinements, as the curve has not plateaued yet, highlighting its potential to achieve much better results. CYCLE's ability to self-refine also aligns better with human developers' iterative programming practice, where they learn from mistakes and continuously improve. CYCLE exemplifies how code LMs can be designed to imitate iterative programming, bringing real-time improvement to developers' code interactively.

In contrast, baseline models, including the one fine-tuned with the same data as CYCLE, typically plateau after one or two steps of refinement. For example, in Figure 4b that evaluated with MBPP-S benchmark, CYCLE and baseline models start at similar performance, but refinement could not bring further improvement after two steps, while CYCLE continues to go up. This gap sheds light on the necessity of evaluating code LM's capacity for self-refinement: models with similar one-time generation performance might not be similarly helpful, as the one with better self-refinement capacity could assist developers with iterative programming more efficiently in the realistic deployment.



(a) HumanEval (b) MBPP-S (c) APPS

Fig. 4. Performance improvement with self-refinement. The blue curve represents the "vanilla code LM baseline", using CodeGen-350M. The orange curve represents the "code LMs fine-tuned with the correct code". The green curve represents CYCLE-350M.

5.2.2 Execution Feedback Guides CYCLE for Better Self-Refinement. Second, we analyze whether the execution feedback helps CYCLE as expected. To do so, we remove the execution results before each time of refinement, only leaving the problem description, *i.e.,* natural-language prompt, and the past, faulty generation as references. The results are shown in Table 2.

It is evident that execution feedback plays an important role in CYCLE's self-refinement, and removing it significantly drops CYCLE's performance across all three benchmarks. In contrast, baseline models are not sensitive to this removal. After removing the execution results, baseline models perform comparably or even better, suggesting that these models regard execution results mostly as redundant information or even noise. This highlights existing code LM's weaknesses in understanding execution effects and taking advantage of them.

5.2.3 Past Generation Mask (PGM) Effectively Prevents the Exact Copy and Improves CYCLE's Performance. As we introduced in Section 3.2.2, we design PGM to avoid the model taking shortcuts by naively copy-pasting the faulty generation as its prediction, which might also decrease the training loss due to the token overlaps between the faulty code and its refined version [Ding et al. 2020]. To verify the effectiveness of PGM, we train CYCLE-350M-w/o-PGM disabling the PGM (*i.e.,* setting the masking ratio to be 0%) and compare its performance with the standard CYCLE-350M.

Table 2. Cycle's performance decreases when the execution feedback is removed during the self-refinement, while the baseline models could not effectively perceive the execution results.

Benchmark	HumanEval	MBPP-S	APPS
CodeGen			
w/o exec feedback	12.8	24.1	6.9
w/ exec feedback	12.2 ↓ 4.7%	21.8 ↓ 9.5%	6.9 −0.0%
CodeGen + FT w/ Correct			
w/o exec feedback	14.0	26.0	8.3
w/ exec feedback	14.0 −0.0%	23.0 ↓ 11.5%	7.7 ↓ 7.2%
Cycle (Ours)			
w/o exec feedback	15.9	29.5	8.5
w/ exec feedback	**20.7**↑ 20.2%	**32.6**↑ 10.5%	**8.7** ↑ 2.4%

We studied three perspectives of their differences. First, we compare their overall performance of code generation, *i.e.,* the test suite pass rate as we reported in Section 5.1. Second, we analyze the edit distance, in terms of code tokens, between the faulty generation and its refined code predicted by Cycle-350M-w/o-PGM and Cycle, where a higher value indicates the model modifies more tokens to correct the fault. Third, we study how many exact copies happen along the way of the self-refinement process until it reaches the maximum refine limit. A higher exact copy rate indicates the model copy-pastes the fault generation as the self-refinement prediction more frequently.

Table 3. Impact of PGM on Cycle's performance.

Cycle	Pass Rate % (↑)			Token Edit Distance (↑)			Exact Copy Rate % (↓)		
	HEval	MBPP-S	APPS	HEval	MBPP-S	APPS	HEval	MBPP-S	APPS
w/o PGM	20.12	31.38	7.81	42.44	31.34	162.67	8.23	15.66	1.24
w/ PGM	**20.73**	**32.55**	**8.67**	**45.12**	**33.17**	**162.72**	**7.12**	**12.86**	**1.21**

The comparison is shown in Table 3. It is clear that PGM helps to improve the overall performance of self-refinement across all three benchmarks. Also, when PGM is enabled, the model tends to edit more tokens, on average, to refine the faulty generation, and avoid the exact copy at its best.

Result-2: The core feature, self-refinement, significantly enhances Cycle's capability, aligning with iterative programming practices in human developers. In addition, the execution feedback is vital for Cycle's self-improvement, with its removal leading to a notable drop in performance. Also, Past Generation Mask (PGM) effectively prevents naive copy-pasting, leading to better overall performance and more diverse code edits for refinement.

5.3 RQ3. Relationship Between Cycle's Self-Refinement and Top-K Generation

To more comprehensively evaluate code LMs' capacity in code generation, Chen et al. [2021] proposes to generate up to K sequences simultaneously to explore a more diverse search space and use test cases to pick the correct ones as the final prediction. Later, open-source code LMs [Li et al. 2023; Nijkamp et al. 2023b; Rozière et al. 2023] incorporate such a setting as an additional evaluation for the code generation task.

In this section, we explain that CYCLE's self-refinement for code generation is an orthogonal direction and maintains comparable overhead to the top-k generation. In addition, the self-refinement is applicable to the top-k generations to further improve the overall performance.

5.3.1 Preliminary. When generating code autoregressively, code LMs estimate the probability of each possible next token based on the given context. While it is intuitive to always choose the most probable next token during the generation, which is known as greedy decoding, the single generation might not be able to fully expose the diverse knowledge that the model has learned. To explore a wider search space while ensuring coherence, several techniques [Holtzman et al. 2020; Tillmann and Ney 2003] are applied to code LMs for generating multiple predictions simultaneously. Existing code LMs [Chen et al. 2021; Li et al. 2023; Nijkamp et al. 2023b; Rozière et al. 2023] have shown that such a top-k generation could improve the overall accuracy.

Beam Search. Beam search is a deterministic search algorithm for sequence generation, extending greedy decoding to generate multiple, most probable sequences. It explores multiple candidate sequences simultaneously by retaining the top-k most promising ones at each step, and the generated sequences so far will be ranked based on the cumulative token probabilities. This process is repeated, gradually expanding the sequences, and at the end, the top-k sequences with the highest likelihood score will be the final output.

While it allows the model to output more than one sequence, it also introduces significantly more computations and inference overhead. For example, maintaining multiple sequences on GPU during generation requires linearly more GPU memories. Also, the repetitive ranking during the generation makes it more time and computation-consuming than a single generation.

Nucleus Sampling with Temperature. Holtzman et al. [2020] propose nucleus sampling, also known as top-p sampling, to enhance the creativity of sequence generation with language models. Instead of rigidly choosing the top-k most probable sequences like beam search, nucleus sampling dynamically selects a subset of the most probable words at each generation step and samples one out of them.

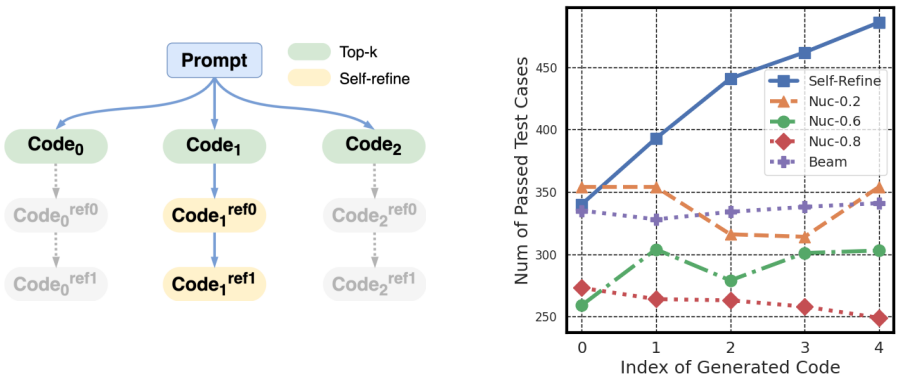
Specifically, when generating the next token, nucleus sampling first cumulates a probability mass of the most likely tokens, called the *nucleus*, until it exceeds the threshold "p" (*e.g.*,, 0.9). Then a concrete token will be randomly sampled from this dynamically determined probability mass. Nucleus sampling can also be coupled with a temperature parameter, which re-scales the likelihood distribution of words and consequently affects the dynamic selection of the *nucleus*. A higher temperature (*e.g.,* 0.8) makes the distribution flatter, involving more tokens into the *nucleus*, while a lower temperature (*e.g.,* 0.2) sharpens the distribution, favoring high-probability tokens.

To generate "k" sequences with nucleus sampling, the typical approach is to duplicate the input "k" times and apply the sampling independently, where, at each step, the randomness of sampling will make the difference. Similar to beam search, as it maintains multiple sequences on GPU, the extra memory overhead is applied.

For the rest of the section, we conduct experiments to study the code LM's performance with top-k generation using beam search and nucleus sampling with different temperatures and compare the results with CYCLE's self-refinement, highlighting their difference and the potential interaction. To save the computation, all experiments are conducted with CYCLE-350M, but the experiments are designed to be generalizable to all sizes of code LMs, and we expect the observations to be maintained among different model sizes.

5.3.2 CYCLE's Self-Refinement is Orthogonal to Top-k Generation. As we show in Figure 5a, the iterative process of self-refinement is conceptually different from the top-k generation. Concretely, top-k generation produces multiple code candidates (green nodes in Figure 5a) with the same prior condition, *i.e.,* only the prompt, which explores the *breath* of the code generation. In contrast, the

process of self-refinement is iteratively updating the prior condition, where the old generation and the execution results will be continuously updated and concatenated to the prompt as additional references (yellow nodes in Figure 5a). Therefore, different from the breath exploration of top-k generation, self-refine is improving a specific generation with directional guidance in *depth*.



(a) Conceptual illustration of the difference

(b) Empirical evidence from MBPP-S

Fig. 5. Cycle's self-refinement is orthogonal to the top-k generation. Conceptually, top-k generation explores in breadth, while self-refinement improves a specific generation in depth. Empirically, when compared to nucleus sampling (with the temperature of 0.2, 0.6, 0.8) and beam search, self-refinement optimizes the generated code towards execution guidance, while top-k produces more diverse programs that pass a similar amount but complementary test cases.

We also conduct experiments to study how the orthogonal instincts perform differently during the code generation.

Setup. We consider four types of top-k generation as a comparison: nucleus sampling with the temperature of (0.2, 0.6, 0.8), and beam search. Specifically, the code LM will generate the top-5 sequences as the prediction, where 5 is also the number of code Cycle produces along the iterative self-refinement according to our configuration (Section 4.4). Then we indexed the generated code for each method following their generation order. Finally, we check how many and which test cases each generation will pass individually across all methods. The experiment is conducted with the MBPP-S benchmark which contains 1,323 test cases in total.

Findings. As shown in Figure 5b, Cycle eventually passes more test cases in MBPP-S step by step, since it learns to improve the old generation according to the execution results, revealing the behavior of exploring code generation in depth. On the contrary, the number of passed test cases is mostly similar across different generations of a specific top-k generation method, either nucleus sampling or beam search. After checking in detail, we realize that the accumulated number of passed test cases across all five generations is significantly higher than any individual generation. For example, all five generations from nucleus sampling with temperature 0.8 passed 513 test cases altogether, while one generation could only pass roughly 260 test cases. This reveals that top-k generation samples diverse code as the prediction, exploring the breadth of the search space.

5.3.3 Cycle could further refine top-k generations. Motivated by the findings in 5.3.2, we delve deeper to study whether Cycle's self-refinement is applicable to top-k generation and whether combining these two orthogonal techniques could further push the performance boundary. Specifically,

Cycle is supposed to be able to improve any past generation, so we evaluate Cycle's performance to refine the top-k generation. As we show in Figure 5, the top-k generations (green nodes) will be improved by Cycle independently, producing refined programs for each of them (gray and yellow nodes).

Setup. We evaluate Cycle's capacity in refining top-k generation using HumanEval, MBPP-S, and APPS benchmarks, similar to our main evaluation (Section 5.1). We again choose nucleus sampling with the temperature of (0.2, 0.6, 0.8) and beam search to generate the top-5 predictions, and each generation will be refined four times, resulting in a total of twenty-five predictions.

Table 4. The test suite Pass@5 rate for three programming benchmarks. Cycle's self-refinement is applicable to top-k generation and further improves its performance.

Method	One-time (Pass@5)			Self-Refine (Pass@5)		
	HumanEval	MBPP-S	APPS	HumanEval	MBPP-S	APPS
Beam Search	13.4	21.8	8.5	25.6 +91.0%	47.3 +117.2%	12.3 +44.0%
Nuc. Samp. (tmp=0.2)	15.2	26.7	8.8	25.6 +68.0%	45.9 +71.9%	11.6 +30.9%
Nuc. Samp. (tmp=0.6)	18.9	32.1	9.1	28.1 +48.4%	46.6 +45.3%	11.9 +30.0%
Nuc. Samp. (tmp=0.8)	18.9	29.7	8.9	25.6 +35.4%	47.3 +59.1%	12.1 +36.1%

Findings. The results are reported in Table 4. We can see that Cycle's self-refinement consistently and significantly boosts the pass@5 rate across four methods of top-k generation, up to 117.2% relative improvement. This verifies that the self-refinement is compatible with the top-k generation and can further improve its performance.

5.3.4 The Overhead of Cycle's Self-Refinement is Comparable to Top-k Generation. Another difference between Cycle's self-refinement and top-k generation is that, during the inference, the former is an iterative process while the latter is a simultaneous process. It is difficult to conceptually reason about which approach is supposed to have higher inference overhead since self-refinement is inevitably a sequential process, while top-k generation requires more GPU memory and additional computation of gathering the most probable tokens at each generation step.

Setup. To study and compare the overhead between Cycle's self-refinement and top-k generation, we record two things during their inference: (1) the inference time (in seconds), and (2) the number of executions the machine performs to verify the correctness or collect the execution feedback.

Note that the comparison is conducted based on the restriction that "only five sequences can be generated", no matter whether the process is iterative or simultaneous. Consequently, top-k generation methods will produce five sequences with *one-time inference*, while Cycle produces five sequences by *iteratively generating only one sequence* at a time, executing test cases and collecting feedback, and running inference again to refine the past generation.

Findings. We show the comparison in Table 5. When we compare the inference time of different methods, we notice that Cycle's inference time with self-refinement of generating five sequences is actually shorter than top-k generation, even if self-refinement is an iterative process. The main reason is that, restricted by the fixed size of GPU memory, the batch size of top-k generation has to be reduced when generating more sequences simultaneously, while self-refinement only generates one prediction at a time, so the batch size is significantly larger. The efficient parallel computation of GPU for a larger batch compensates for the time consumption of Cycle's sequential refinement, while the smaller batch size of top-k generation results in more batches that have to be sequentially fed into GPU instead.

Table 5. The comparison of inference overhead between Cycle's self-refinement and top-k generation. "Infer.(s)" represents the model inference time in seconds, "#Exec" represents the number of executions performed for evaluation, and "Pass(%)" represents the ratio of passed test suites (*i.e.,* the ratio of programming problems solved by the method)

Method	HumanEval			MBPP-S			APPS		
	Infer. (s)	#Exec.	Pass (%)	Infer. (s)	#Exec.	Pass (%)	Infer. (s)	#Exec.	Pass (%)
Beam Search@5	507.6	3263	13.4	1222.2	3473	21.8	4804.4	32472	8.5
Nuc. Samp.@5 (tmp=0.2)	425.5	4723	15.2	1084.4	5630	26.7	5179.9	31189	8.8
Nuc. Samp.@5 (tmp=0.6)	428.9	6422	18.9	1147.8	8224	32.1	5213.7	32412	9.1
Nuc. Samp.@5 (tmp=0.8)	423.9	6612	18.9	1084.7	8568	29.7	5224.8	32458	8.9
Cycle @5	403.2	6725	20.7	762.6	8755	32.3	4238.3	32475	8.7

Conceptually, the number of executions of test cases should be exactly the same between top-k generation and Cycle's self-refinement, since they generate the same number of predictions for each programming problem in the benchmarks, and the number of test cases for each problem is fixed. However, in practice, we notice that the numbers are varied across methods, and Cycle is more comparable to nucleus sampling with a high temperature but higher than that with a low temperature and the beam search. The main reason is that our implementation of the execution framework follows the original design of HumanEval, where Chen et al. [2021] minimizes the number of executions by executing the same predictions only once. As a result, the more diverse the predictions are, the more executions will be performed, and that is why we see an increase in the execution number when the temperature increases.

To conclude, we could not see a significant gap between top-k generation and Cycle's self-refinement, in terms of both performance and inference overhead, since they have advantages and disadvantages in orthogonal angles. As we found in Section 5.3.3, the better solution is to take advantage of both techniques.

Result-3: Cycle's self-refinement is an orthogonal direction to top-k generation with comparable overhead, where the former improves one single generation in depth iteratively while the latter encourages exploring the breadth of the sequence space for the one-time generation. Further, Cycle could be applied to top-k generation to further improve the overall performance of code generation.

5.4 RQ4. Ablation Study of Past Generation Mask and Data Mixture

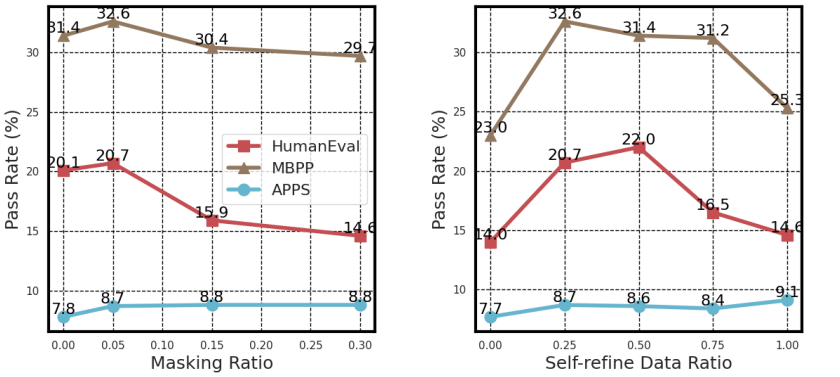
In this section, we conduct an ablation study to analyze the impact of Past Generation Mask's (PGM) masking ratio and the data combination for self-refinement training, since these empirical choices are expected to have non-trivial impacts on the learning process.

As we introduced in Section 3.2.2, PGM randomly masks a certain ratio of code tokens to prevent the model from lazily taking shortcuts, using the exact copy of the past, faulty generation as the refined prediction. However, different masking ratios could result in different learning behaviors. Concretely, when the masking ratio is high, there will be limited tokens for the model to refer to, and the model has to produce most tokens from scratch. On the contrary, when the masking ratio is low, the model could learn to borrow some helpful pieces of code, such as the code structure and the meaningful identifiers while not naively replicating past errors.

In Section 3.2.3, we proposed to mix self-refine samples (Section 3.1) and original code samples to construct the data for training, ensuring the model effectively learns self-refining while maintaining the capacity of general code generation. However, the proportion of these two data resources could affect the learning process. Specifically, when self-refine samples are overly dominant, the model

will be optimized mostly towards self-refinement, where the natural language prompt, faulty generation, and the execution results are all required to present as the prior condition, while losing the capacity in one-time code generation where only the prompt is given, and vice versa.

Setup. To conduct the ablation study on the PGM masking ratio and the data combination proportion, we train CYCLE-350M multiple times with controlled settings and evaluate across the three programming benchmarks to compare across different settings for concluding the trend. Specifically, to study the impacts of PGM masking ratio we train CYCLE-350M four times with ratios of [0.0, 0.05, 0.15, 0.30] while maintaining all other training settings the same. Similarly, to study the difference across varied data mixture proportions, we train CYCLE-350M five times with the self-refine data ratios of [0%, 25%, 50%, 75%, 100%].



(a) The impact of PGM masking ratio (b) The impact of data mixture proportion

Fig. 6. Ablation Study

Findings. The results regarding PGM masking ratios are plotted in Figure 6a. From the figure, we deduce that PGM operates optimally at a lower masking ratio. Specifically, a masking ratio of around 0.05 emerges as an optimal point, indicating that the slight obfuscation can effectively prevent the exact copy during self-refinement while retaining valuable referring tokens in the faulty generation. However, going too high with the masking ratio seems to counteract its benefits, especially evident in the MBPP's sharp decline.

The results of the data mixture proportion are plotted in Figure 6b. We can see that an amalgamation of original and self-refined code samples offers superior training outcomes than one resource standalone. A balance, possibly around the 25% ratio, appears to be a sweet spot. Solely depending on one kind of data, especially pure self-refined data, does not yield the best results, highlighting the importance of diverse training samples. Interestingly, we notice the performance on APPS exhibits a relatively stable pass rate, starting at 7.7%, slightly peaking at 8.7% for a 25% data ratio, and settling around 9.1% for a full self-refinement. We speculate that the slightly higher performance in APPS when trained with 100% self-refine samples is due to the similar distribution between our training data resources (Section 3.1) and APPS, where both mostly contain more challenging programming problems than HumanEval and MBPP.

Result-4: In conclusion, while PGM and data mixture are valuable techniques in the realm of self-refinement, careful calibration of their parameters, specifically masking ratio and data combination proportion, is essential for optimal performance.

6 RELATED WORK

Pre-trained Code Large Language Models for Code Generation. The bloom of generative pre-trained large language models (LLMs) like GPT-3 [Brown et al. 2020] in natural language processing has inspired a series of models that are pre-trained on source code with a focus on the task of code generation [Zan et al. 2023]. Codex [Chen et al. 2021] is a series of proprietary Code LLMs up to 12B fine-tuned from GPT-3 on publicly available code from GitHub. The 12B Codex-S, which is further fine-tuned on standalone functions achieves 37.7 Pass@1 on the HumanEval benchmark also proposed by Chen et al. [2021]. GitHub Copilot [GitHub 2021], a production version of Codex shows potential to assist software development through its code completion ability.

Following Codex, many open-source models have been proposed. CodeGen [Nijkamp et al. 2023b] is a series of open-source Code LLMs up to 16B trained on English, multiple programming languages, and then Python datasets in order. The 16B CodeGen achieves 29.28 Pass@1 on the HumanEval benchmark. However, only left-to-right code completion doesn't cover all software engineering needs such as code debugging and refactoring. Therefore, a class of models like Incoder [Fried et al. 2023] and SantaCoder [Allal et al. 2023] explores another pre-training task, fill-in-the-middle (FIM), which tasks the model to fill in a masked region based on the surrounding context. However, FIM models can only help with debugging or refactoring after human software engineers locate the regions of code with bugs or need improvement. This doesn't align with real-life software development scenarios, where human engineers often spend the most time trying to pinpoint the regions that need to be refined.

Later, the success of ChatGPT [OpenAI 2022], a series of proprietary models that can follow instructions in a conversational way, opens up new opportunities to train Code LLMs to refine code in a flexible manner. These models can be prompted to perform unlimited new open-ended generation tasks, even if they have not been trained on these tasks. Particularly for code generation, users can prompt the chat models to edit, debug, and refactor existing code, especially code generated by themselves. The technique behind those versatile chat models is instruction tuning [Mishra et al. 2022; Sanh et al. 2022; Wei et al. 2022a]. It fine-tunes pre-trained LLMs on instruction-answer pairs covering a wide variety of tasks, which aims to generalize the instruction-following ability to unseen tasks. Along this promising direction, most recent open-source models like StarCoder [Li et al. 2023] and Code Llama [Rozière et al. 2023] are released with instruction-tuned versions in addition to pre-trained models. Nonetheless, current instruction tuning procedures and datasets only focus on the one-step instruction-following ability but overlook the potential of raising the model's capability of refining answers in multiple steps.

Improving the Quality of LLM Generation from Feedback. Instruction tuning not only empowers LLMs to multitask but also aligns their generation with human preference since the curated instruction-answer pairs in the datasets imply how humans would appropriately solve certain problems. However, instruction tuning is costly as it requires a large amount of high-quality human-labeled data.

To address this issue, a developed technique, Reinforcement Learning from Human Feedback (RLHF) [Christiano et al. 2023; Ouyang et al. 2022; Stiennon et al. 2022; Ziegler et al. 2020] is adopted to further improve instruction-tuned LLMs. Specifically, during the RLHF process, human annotators first grade the quality of various LLM generations by assigning a score, then a reward model is

trained to emulate human grading, and finally, the LLM is fine-tuned using reinforcement learning with the reward model. In this way, RLHF helps LLMs capture implicit human feedback information by training them for higher rewards. The most powerful chat models like GPT-4 [OpenAI 2023], Claude [Anthropic 2023], and Llama2-Chat [Touvron et al. 2023] are all fine-tuend using RLHF. However, in the RLHF framework, the original human feedback is typically a single discrete score, which is implicit and sometimes even conflicting. Moreover, improving generation using the RLHF requires updating the model's parameters, which cannot help enhance real-time user experience. For example, within the RLHF framework, human software engineers can give a score to the model-generated code as real-time feedback, but it's not practical to update the model's parameters immediately and this single score doesn't provide any explicit guidance about how to improve future generations.

To complement the limitations of RLHF, efforts have been made to provide LLMs with explicit feedback as prompts and leverage their instruction-following ability to refine previously generated outputs [Madaan et al. 2023; Nair et al. 2023; Shinn et al. 2023]. Typically, the explicit feedback is a critique or an explanation generated by the same model. Therefore, these methods can be categorized as "Self-Refinement with Prompting". For code generation, in particular, Zhang et al. [2023] uses unit test outputs and error messages as explicit feedback for self-refinement. Chen et al. [2023] further includes self-generated explanations and execution traces to the feedback. Self-Refinement with Prompting does improve the quality of code generation, but it demands the model to have a strong instruction-following capability as well as a long context window to include different types of feedback.

7 LIMITATIONS AND THREATS TO VALIDITY

To prove the concept of the above goal with the best efficiency, we choose to focus Cycle on Python language as its first attempt. Python has gained immense popularity in recent years and stands as one of the most widely used programming languages. Its reputation for being beginner-friendly, versatile, and having a clean, readable syntax makes it a natural choice for initial trials. However, this makes our current version of Cycle not applicable to other languages, such as C++ and Java, and it is challenging to unify the execution across multiple languages due to the complicated dependency. We regard the multi-lingual version of Cycle as an exciting extension of this work.

Another limitation of our framework is the dependency on canonical solutions. Grounding our framework in canonical solutions ensures closer proximity to correct code, but such solutions do not always exist. We have to rely on the coding challenge benchmark, such as CodeContest to collect such data, and such data is restricted by scope and might not be fully reflecting the complexity of the realistic development.

In addition, we assume the existence of unit test cases for the generated code snippets. This assumption might not be generalizable to collecting large-scale data. While well-developed projects typically maintain such test cases, it could be difficult to automatically pair the function with its corresponding unit test due to the varied configurations across different projects.

In conclusion, while our self-refinement framework has made promising strides in enhancing code generation, there remain exciting opportunities for growth. Each limitation not only informs us of the challenges but also provides a clear path for future research and improvement. Our vision is to harness these insights to evolve and make our framework even more robust and adaptable to the dynamic world of coding.

8 CONCLUSION

In this paper, we propose Cycle framework, making an attempt to teach code LMs to self-refine according to the faulty generation in the past and the execution feedback. We evaluate Cycle's code generation capability with three popular programming benchmarks: HumanEval [Chen et al. 2021], MBPP-Sanitized [Austin et al. 2021], and APPS [Hendrycks et al. 2021]. To illustrate the effectiveness and generalizability of Cycle, we train four variants of Cycle with varied parameter sizes ranging from 350M to 3B. From the evaluation results, we conclude that Cycle is pretty effective at self-refinement, consistently boosting the code generation performance, by up to 63.5% relative improvement, across four model sizes on all three benchmarks, while maintaining decent one-time generation capacity. With efficient self-refinement learning, Cycle-350M outperforms StarCoder-1B across all three programming benchmarks, and Cycle-1B matches the performance of StarCoder-3B. With the in-depth analysis, we also empirically reveal that Cycle is effective at capturing execution feedback and has great potential to assist human developers with iterative programming.

DATA-AVAILABILITY STATEMENT

We release our code, data, and model checkpoints to encourage further exploration in this direction. The artifact that supports the results discussed in this paper is available at https://github.com/ARiSE-Lab/CYCLE_OOPSLA_24.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable feedback and comments. This work was supported in part by an IBM Ph.D. Fellowship, DARPA/NIWC-Pacific N66001-21-C4018, NSF CNS-2247370, CCF-2221943, CCF-2313055, CCF-1845893, and CCF-2107405. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect those of IBM, DARPA, or NSF.

REFERENCES

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023. SantaCoder: don't reach for the stars! arXiv:2301.03988 [cs.SE]

Amazon. 2023. Amazon CodeWhisperer: Build applications faster and more securely with your AI coding companion. https://aws.amazon.com/codewhisperer/.

Anthropic. 2023. Introducing Claude. https://www.anthropic.com/index/introducing-claude.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. 7, OOPSLA1, Article 78 (2023), 27 pages. https://doi.org/10.1145/3586030

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient Training of Language Models to Fill in the Middle. arXiv:2207.14255 [cs.CL]

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165 [cs.CL]

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, and *et al.*. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. arXiv:2304.05128 [cs.CL]

Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2023. Deep reinforcement learning from human preferences. arXiv:1706.03741 [stat.ML]

Yangruibo Ding, Baishakhi Ray, Devanbu Premkumar, and Vincent J. Hellendoorn. 2020. Patching as Translation: the Data and the Metaphor. In *35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. https://doi.org/10.1145/3324884.3416587

Adam Paszke *et al.*. 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library.*

Thomas Wolf *et al.*. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999 [cs.SE]

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. 2021. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *CoRR* abs/2101.00027 (2021). arXiv:2101.00027 https://arxiv.org/abs/2101.00027

GitHub. 2021. GitHub Copilot: Your AI Pair Programmer. https://copilot.github.com/.

Qi Guo, Junming Cao, Xiaofei Xie, Shangqing Liu, Xiaohong Li, Bihuan Chen, and Xin Peng. 2023. Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study. arXiv:2309.08221 [cs.SE]

Jingxuan He and Martin Vechev. 2023. Large Language Models for Code: Security Hardening and Adversarial Testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS'23)*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, J. Vanschoren and S. Yeung (Eds.), Vol. 1. Curran. https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/c24cd76e1ce41366a4bbe8a49b02a028-Paper-round2.pdf

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rygGQyrFvH

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

HuggingFace. 2023. *Hugging Face Model Hub*. https://huggingface.co/models.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. arXiv:2001.08361 [cs.LG]

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The Stack: 3 TB of permissively licensed source code. *arXiv preprint arXiv:2211.15533* (2022).

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Brussels, Belgium, 66–71. https://doi.org/10.18653/v1/D18-2012

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, and *et al.*. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL]

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, and *et al.*. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158 arXiv:https://www.science.org/doi/pdf/10.1126/science.abq1158

Hao Liu, Xinyang Geng, Lisa Lee, Igor Mordatch, Sergey Levine, Sharan Narang, and Pieter Abbeel. 2023. Forgetful causal masking makes causal language models better zero-shot learners. https://openreview.net/forum?id=YrZEKNLWhlp

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. arXiv:2303.17651 [cs.CL]

Mike Mirzayanov. 2020. *Codeforces: Results of 2020*. https://codeforces.com/blog/entry/89502.

Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. 2022. Cross-Task Generalization via Natural Language Crowdsourcing Instructions. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Dublin, Ireland, 3470–3487. https://doi.org/10.18653/v1/2022.acl-long.244

Varun Nair, Elliot Schumacher, Geoffrey Tso, and Anitha Kannan. 2023. DERA: Enhancing Large Language Model Completions with Dialog-Enabled Resolving Agents. arXiv:2303.17071 [cs.CL]

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023a. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. arXiv:2305.02309 [cs.LG]

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023b. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=iaYcJKpY2B_

OpenAI. 2022. Introducing ChatGPT. https://openai.com/blog/chatgpt/.

OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs.CL]

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *CoRR* abs/2105.12655 (2021). arXiv:2105.12655 https://arxiv.org/abs/2105.12655

Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training. https://api.semanticscholar.org/CorpusID:49313245

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. https://api.semanticscholar.org/CorpusID:160025533

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Tali Bers, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M. Rush. 2022. Multitask Prompted Training Enables Zero-Shot Task Generalization. arXiv:2110.08207 [cs.LG]

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv:2303.11366 [cs.AI]

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 56 (2014), 1929–1958. http://jmlr.org/papers/v15/srivastava14a.html

Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul Christiano. 2022. Learning to summarize from human feedback. arXiv:2009.01325 [cs.CL]

Christoph Tillmann and Hermann Ney. 2003. Word Reordering and a Dynamic Programming Beam Search Algorithm for Statistical Machine Translation. *Computational Linguistics* 29, 1 (2003), 97–133. https://doi.org/10.1162/089120103321337458

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. 6000–6010.

Ximei Wang, Jinghan Gao, Mingsheng Long, and Jianmin Wang. 2021. Self-Tuning for Data-Efficient Deep Learning. In *International Conference on Machine Learning (ICML)*.

Xuezhi Wang and Jeff Schneider. 2015. Generalization Bounds for Transfer Learning under Model Shift. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence* (Amsterdam, Netherlands) *(UAI'15)*. AUAI Press, Arlington, Virginia, USA, 922–931.

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2022a. Finetuned Language Models Are Zero-Shot Learners. arXiv:2109.01652 [cs.CL]

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022b. Emergent Abilities of Large Language Models. arXiv:2206.07682 [cs.CL]

Peter West, Chandra Bhagavatula, Jack Hessel, Jena Hwang, Liwei Jiang, Ronan Le Bras, Ximing Lu, Sean Welleck, and Yejin Choi. 2022. Symbolic Knowledge Distillation: from General Language Models to Commonsense Models. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Seattle, United States, 4602–4625. https://doi.org/10.18653/v1/2022.naacl-main.341

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3520312.3534862

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large Language Models Meet NL2Code: A Survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Toronto, Canada, 7443–7464. https://doi.org/10.18653/v1/2023.acl-long.411

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. arXiv:2305.04087 [cs.SE]

Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving. 2020. Fine-Tuning Language Models from Human Preferences. arXiv:1909.08593 [cs.CL]

Received 21-OCT-2023; accepted 2024-02-24