PERFCODEGEN: Improving Performance of LLM Generated Code with Execution Feedback

Yun Peng[†], Akhilesh Deepak Gotmare[*‡], Michael R. Lyu[†], Caiming Xiong[‡], Silvio Savarese[‡], Doyen Sahoo[‡]
[†] *The Chinese University of Hong Kong, Hong Kong, China*
[‡] *Salesforce Research, Singapore*
{ypeng, lyu}@cse.cuhk.edu.hk, {akhilesh.gotmare, cxiong, ssavarese, dsahoo}@salesforce.com

***Abstract*—Large Language Models (LLMs) are widely adopted for assisting in software development tasks, yet their performance evaluations have narrowly focused on the functional correctness of generated code. Human programmers, however, expect AI assistants to generate not only correct but also optimally efficient code. We propose PERFCODEGEN, a training-free framework that enhances the performance of LLM-generated code by incorporating feedback based on runtime during test case execution into the self-refinement iterations. With PERFCODEGEN, we achieve speedups for a significantly higher proportion of problems compared to using the base LLM with sophisticated prompting techniques. Applied to open-weight language models like Phi-3-mini, PERFCODEGEN achieves code optimization rates comparable to naive prompting of powerful closed models like GPT-4. We achieve state-of-the-art code optimization on benchmarks such as HumanEval, MBPP, and APPS, frequently surpassing the ground truth reference solutions with PERFCODEGEN using GPT-3.5 and GPT-4. Additionally, we demonstrate the effectiveness of our approach in enhancing code quality across a range of open-weight LLMs of varying sizes including Phi-3-mini (3.8B), Llama 3 8B, Mixtral 8x7B (13B active), Command R (35B), and Llama 3 70B. PERFCODEGEN's effectiveness at generating performant code underscores the importance of integrating execution feedback into the code generation process, highlighting a path forward for more robust and reliable AI-driven software development.**

***Index Terms*—Large Language Models, Code Generation, Efficient Code, Runtime Efficiency, Code Optimization**

I. INTRODUCTION

Large Language Models (LLMs) are now widely used for code completion [1], [2], [3], as well as for tasks like unit test case generation [4], bug fixing [5], feature addition [6], and other stages of software development [7], [8]. A recent Github survey [9] highlights this rapid proliferation, estimating that 92% of U.S. based developers are already using AI coding tools. However, despite this widespread adoption of LLMs, evaluation has almost exclusively focused on the functional correctness of generated code [10], [11], [12], [13], often overlooking other key aspects of code quality. Runtime efficiency of a program, in particular, is a central consideration in software design, due to its significant impact on user experience, serving costs and carbon footprint of a software application [14], [15]. Recent work [16] studying non-functional requirements such as efficiency, security and maintainability of LLM-generated programs reveals that current models generally falter on these key quality metrics. This is particularly concerning, as less experienced developers have been shown to be more likely to accept AI-suggested code [17], often neglecting quality aspects, which burdens the code review and maintenance stages [18].

Some prior work [19], [20] has proposed fine-tuning LLMs to generate performance improvements for a given working program. However, this approach is challenging to scale due to the need for parallel training data in the code domain, which can be significantly more expensive to collect and manually validate than natural language data. Additionally, these methods require the availability of a functionally correct input program to optimize, which is not typically the case when writing code from scratch. Moreover, they do not leverage execution feedback from unit tests, which has been shown to be crucial in improving code correctness [21]. This lack of execution feedback utilization is a notable limitation, as unit tests provide valuable insights into runtime behavior and performance characteristics of a program under test.

Prompting advances such as Chain of Thought [22], [23] and Self-Refine [24] have enhanced LLM output quality without additional training, but limitations remain, including issues with hallucinations, sycophancies, and unreliability in refining initial responses [25], [26], [27]. To address these, several recent works [28], [29], [30], [31], [32] have proposed providing verbal feedback or grounding from an automatic tool or environment to the LLM during the refinement stage. However, these approaches often do not assess or improve on quality aspects beyond mere functionality when generating code.

We propose PERFCODEGEN, a framework that leverages code execution feedback in the iterative self-refinement stages of an LLM to improve the performance of generated code beyond merely ensuring functional correctness. First, we use environment feedback based on unit test execution to self-refine code generated from the base LLM for correctness. This yields a larger set of correct solutions for our framework to optimize, increasing the likelihood of generating an optimally efficient program in the subsequent optimization phase. For performance refinement, we first assess the runtime required by correct solutions for each unit test. We then provide verbalised performance feedback to the LLM indicating the most expensive unit test encountered during execution and instruct it to optimize the previously generated solution using this feedback. The performance feedback is designed to resemble how a human programmer would optimize a program by identifying the most time-consuming portions of

* Corresponding Author.

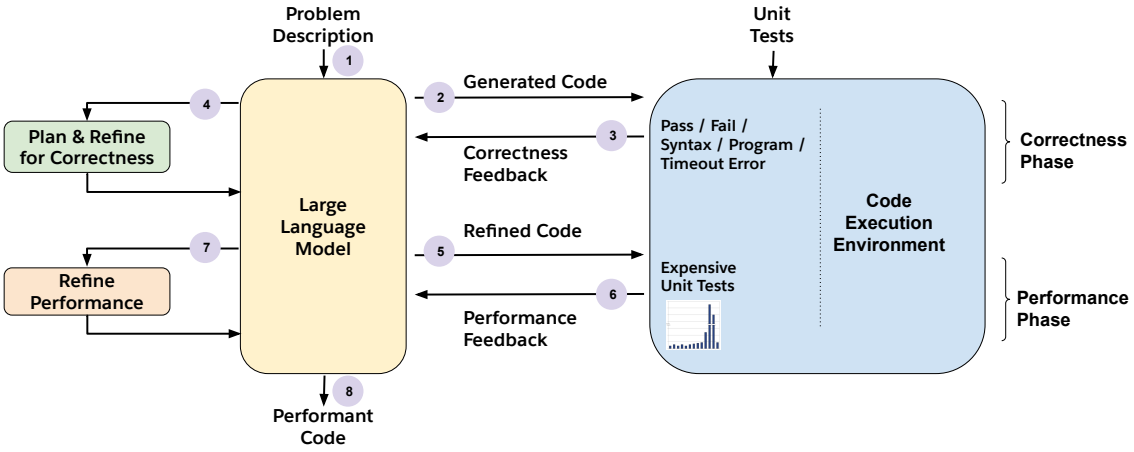

Fig. 1. PERFCODEGEN Given a problem description (1), we prompt the LLM to generate a candidate solution (2), which is passed to an execution environment to collect feedback on correctness (3). The LLM is then prompted to self-reflect on this feedback in a planning stage, and accordingly generate a refinement using this context (4). This process is iterated over till correctness (4-2-3). Correct code obtained from this phase is self-refined for runtime-efficiency (7), and then passed to the environment to be executed (5) and the most time-consuming unit test(s) is identified and passed as performance feedback to the LLM (6), that acts on it by generating a self-refinement to make the correct solution more efficient (7). This refinement is tested for correctness (2, 3) and passed as the final code solution to the problem (8) if correct, else we fall back to the correct program from (3) if any.

it. We evaluate PERFCODEGEN's effectiveness in generating runtime-efficient code on tasks from three widely used Python programming benchmarks: HumanEval [1], MBPP [2] and APPS [10]. We use 5 open-weight and 2 closed language models of varying sizes (Phi-3-mini 3.8B, Llama 3 8B, Mixtral 8x7B (13B active), Command R 35B, Llama 3 70B, GPT-3.5 and GPT-4), and witness consistent and significant gains in the correctness and runtime efficiency of LLM-generated programs with PERFCODEGEN's verbalised execution feedback.

Our work is organized as follows: In Section II, we provide a detailed methodology of the proposed PERFCODEGEN framework. Section III presents experimental results demonstrating its effectiveness, accompanied by ablations comparing it with several other prompting approaches in Section III-C. We present a discussion on the limitations and threats to validity of our work in Section IV. In Section V, we discuss related work, followed by a brief conclusion in Section VI.

II. PERFCODEGEN METHODOLOGY

We begin by prompting a given LLM with a problem description x in natural language that details the requirements of the program to be generated. We sample K candidate generations $C = \{y_x^i\}_{i=1...K}$ with nucleus sampling. We use execution feedback to refine the incorrect programs within C in order to increase the number of correct programs in this initial seed set. In the subsequent performance optimization stage, we provide verbalised feedback to the LLM indicating the most expensive unit tests for a code solution and obtain a refinement using this performance feedback - 1 per correct program from the pool of k programs generated in the correctness phase. We detail the correctness enhancement phase in Section II-A, and the performance refinement phase in Section II-B. Figure 1 illustrates the overall PERFCODEGEN framework with its correctness and performance improvement phases.

A. Generating Correct Code

We assess the correctness of LLM generated programs using a set of J unit tests $U(x) = \{u_x^j\}_{j=1}^J$ corresponding to a problem x. After assessing correctness of all candidates, we iteratively refine incorrect ones based on execution feedback from the unit tests. For any $y_x^i \in C$ that fails any of the unit tests, we prompt the LLM again, this time incorporating the environment feedback for correctness verbalised as part of the prompt along with the failed solution and one of the failing unit tests. The LLM is instructed to reflect and plan, and then generate a refined correct solution based on the intermediate reflection and planning result (prompts in Figure 2).

The inclusion of a reflection-planning phase, as suggested by prior work [22], [23], increases the likelihood of the LLM generating a correct solution. The final refinement of y_x^i generated by the LLM is then tested for correctness, and is passed for the performance optimisation stage if it passes all the unit tests. Subsequently, a set C_{correct} is constructed by removing incorrect samples from C and including their refined versions, if any of them achieve correctness. We could continue this iterative approach to further improve the correctness rate of a given LLM, but to minimize computational costs, we pause after this one iteration. Besides, we observe that the gains in correctness significantly diminish after the first iteration of refinement. Note that this stage is shared across all the different performance refinement methods studied in this work. Having a larger number of correct candidates as seeds for performance refinements benefits the framework's effectiveness, as this implies higher likelihood of PERFCODEGEN generating an optimally efficient program. The problem x is considered as unsolved for correctness by the given LLM if $C_{\text{correct}} = \phi$.

B. Optimising Correct Code using Execution Feedback

For all correct solutions $y_x^i \in C_{\text{correct}}$ that are constructed using the samples and refinements as described in Section II-A, we prompt the model to (self-)refine its solution to optimise performance while preserving functional equivalence. If this refinement breaks correctness, we stop and return the fastest program from C_{correct}. If this refinement preserves correctness, we identify the unit test that this refined solution spends the most time on, and provide it as part of the feedback to the LLM for further performance refinement. This approach mirrors how developers would identify the most time consuming pieces (hot spots) or performance bottlenecks in their code to come up with runtime improving code changes.

To identify such an informative unit test u_x^f, we measure the execution time consumed by our initial refinement (still denoted by y_x^i for simplicity) to pass each available unit test u_x^j corresponding to the given problem x. This involves conducting E independent executions for each solution-test pair in identical compute environments. After sorting this set of E observations, let $t(y_x^i, u_x^j)[e]$ be the e-th smallest execution time consumed by y_x^i on the j-th unit test u_x^j. We then calculate the empirical estimate of the expected execution time of a solution on a unit test, excluding the two outliers (smallest and largest execution times) to minimize the impact of potential measurement noise as follows:

$$\hat{t}(y_x^i, u_x^j) = \frac{1}{(E-2)} \sum_{e=2}^{E-1} t(y_x^i, u_x^j)[e]; \tag{1}$$

$$u_x^f = \text{argmax}_{u_x^j} \hat{t}(y_x^i, u_x^j). \tag{2}$$

Our approach banks on the hypothesis that the f-th unit test u_x^f of a problem x, that corresponds to the largest execution time, as defined above, can be highly informative in optimising the performance of y_x^i if included in the feedback to the LLM for generating a revision. Therefore, we re-prompt the model with its latest generation y_x^i, its most time consuming unit test u_x^f and an instruction to optimise the performance given this feedback (prompt in Figure 3). This leads to a refinement denoted by $\tilde{y}_x^i$. The fastest amongst the refined correct outputs ($\{\tilde{y}_x^i \,|\, \tilde{y}_x^i \text{ passes correctness}\}_{i=1\ldots K}$) is considered as the final performant solution for x. Unlike the correctness phase, we employ the greedy decoding algorithm (sampling temperature set to 0) in this stage of performance refinement, as we intend to collect only one refinement per correct code piece to minimize LLM inference costs. If none of the refinement $\tilde{y}_x^i$ pass correctness, we fall back to the fastest correct base solution from C_{correct}, where fastest from C_{correct} is also found using E executions on all unit tests. Similar to optimising for correctness, we could continue refining for performance with more iterations, or include a planning step before refinement, but we pause after this one iteration without the planning step for algorithmic simplicity and lower inference costs.

III. Experiments

We describe the experiments demonstrating the effectiveness of PerfCodeGen for generating runtime-efficient programs in this Section. Section III-A outlines our experimental setup, Section III-B provides the main results with all the models on the HumanEval and MBPP benchmarks. In Sections III-C and III-D, we compare PerfCodeGen's execution feedback and planning scheme with alternative prompting strategies.

A. Setup: Metrics, Datasets and Models

To evaluate the correctness and runtime-efficiency of LLM-generated solutions using different approaches, we follow prior work [19] to compute the below metrics using the fastest (Best@k) LLM-generated correct program out of k samples.

- **Percent Optimized** [%Opt]: The proportion of problems where the fastest correct LLM-generated program y_x is more runtime-efficient (at least 10% faster) than the ground truth reference program g_x.

$$\frac{100}{N} \cdot \sum_x \mathbb{1}_{\sum_j \hat{t}(y_x, u_x^j) < 0.9 \cdot \sum_j \hat{t}(g_x, u_x^j)} \tag{3}$$

- For a consistent evaluation across multiple LLMs, this proportion is computed on the subset of problems in the test set where all baseline LLMs have at least one correct solution (details of benchmark in Appendix VIII-A).
- **Percent Correct** [%Correct]: The proportion of problems in the test set where the LLM generates at least one correct solution out of k candidates.
- **Speedup**: For problems where we obtain atleast one correct LLM-generated program y_x, we calculate speedup as the absolute ratio between the execution time (to pass all unit tests) required by the ground truth reference program g_x and the execution time required by the fastest correct LLM-generated program y_x.

$$\frac{\sum_x \sum_j \hat{t}(g_x, u_x^j)}{\sum_x \sum_j \hat{t}(y_x, u_x^j)} \tag{4}$$

Following prior work [16], we rely on an empirical estimation of the execution time of Python programs (averaging over repeated executions after excluding outliers), despite its drawbacks and challenges like high compute requirements. While tools like the gem5 simulator [33] reliably and efficiently determine CPU cycles of a program, they do not offer support for Python. Nevertheless, our qualitative analysis (Listing 1, 2) confirms that the differences observed in execution time ($\hat{t}$) correspond to clear differences in coding patterns. To estimate the execution time of a candidate solution, we use $E = 12$ executions for each unit test as described in Equation (1). We then compute the above three metrics using the fastest correct program y_x obtained from k (Best@k) candidates. If there are multiple ground truth solutions, we only use the fastest one as the reference for computing all metrics. We study the impact of sampling budget on the effectiveness of our framework by using 3 different settings with $k \in \{1, 8, 20\}$.

We perform our analysis by treating %Opt as the superior metric over speedup when comparing different methods. A method achieving higher %Opt would be considered more

TABLE I
%OPT, %CORRECT AND SPEEDUP RESULTS ON HUMANEVAL AND MBPP WITH k OF 1 AND 8.

Model	Method	Best@1			Best@8		
		%Opt ($\pm\delta$)	%Correct ($\pm\delta$)	Speedup	%Opt ($\pm\delta$)	%Correct ($\pm\delta$)	Speedup
HumanEval							
Phi-3-mini	Base	14.63	51.83	1.24	35.98	78.05	1.44
	PERFCODEGEN	18.29 (+3.66)	57.32 (+5.49)	1.23	40.85 (+4.87)	85.37 (+7.32)	1.68
Mixtral-8x7B	Base	9.43	27.04	1.31	19.5	63.52	1.37
	PERFCODEGEN	11.32 (+1.89)	32.70 (+5.66)	1.31	27.67 (+8.17)	75.47 (+11.95)	1.71
Command R	Base	19.02	54.6	1.37	25.15	71.17	1.43
	PERFCODEGEN	20.25 (+1.23)	57.06 (+2.46)	1.37	32.52 (+7.37)	79.75 (+8.58)	1.46
Llama 3 8B	Base	15.85	56.71	1.35	29.88	76.22	1.39
	PERFCODEGEN	17.07 (+1.22)	62.80 (+6.09)	1.29	31.10 (+1.22)	81.71 (+5.49)	1.62
Llama 3 70B	Base	24.39	75.61	1.39	33.54	84.15	1.42
	PERFCODEGEN	25.61 (+1.22)	84.76 (+9.15)	1.62	39.02 (+5.48)	93.29 (+9.14)	1.71
GPT-3.5	Base	16.05	54.94	1.37	29.63	78.4	**2.34**
	PERFCODEGEN	22.22 (+6.17)	67.28 (+12.34)	1.34	38.89 (+9.26)	90.12 (+11.72)	2.33
GPT-4	Base	24.54	72.39	**1.81**	39.26	88.96	1.82
	PERFCODEGEN	**28.83** (+4.29)	**87.12** (+14.73)	1.68	**46.63** (+7.37)	**94.48** (+5.52)	1.91
MBPP (test)							
Phi-3-mini	Base	20.26	61.21	2.61	38.36	75.86	3.16
	PERFCODEGEN	20.26 (+0.00)	69.40 (+8.19)	2.50	44.40 (+6.04)	84.48 (+8.62)	3.03
Mixtral-8x7B	Base	11.26	45.95	1.46	22.07	66.67	1.63
	PERFCODEGEN	13.06 (+1.8)	53.15 (+7.2)	1.34	38.29 (+16.22)	78.38 (+11.71)	2.83
Command R	Base	15.86	55.51	1.68	25.11	69.16	2.27
	PERFCODEGEN	17.18 (+1.32)	56.83 (+1.32)	1.73	31.72 (+6.61)	75.33 (+6.17)	2.78
Llama 3 8B	Base	16.02	59.74	2.21	28.14	71.86	2.23
	PERFCODEGEN	17.75 (+1.73)	68.40 (+8.66)	1.91	32.90 (+4.76)	82.68 (+10.82)	3.03
Llama 3 70B	Base	18.92	65.32	1.44	26.13	77.93	1.63
	PERFCODEGEN	17.12 (-1.8)	68.47 (+3.15)	1.68	34.68 (+8.55)	88.29 (+10.36)	2.21
GPT-3.5	Base	44.1	66.38	3.42	57.21	76.86	3.69
	PERFCODEGEN	47.16 (+3.06)	73.36 (+6.98)	**4.19**	**71.18** (+13.97)	88.21 (+11.35)	**4.13**
GPT-4	Base	20.87	72.17	2.76	43.48	82.17	2.85
	PERFCODEGEN	**30.87** (+10)	**86.52** (+14.35)	2.70	56.52 (+13.14)	**93.91** (+11.74)	4.01

effective than one with lower %Opt, irrespective of the speedups observed. A larger proportion of tasks solved optimally would generally be more preferable than a fewer proportion of tasks solved more optimally. Speedup should only be analyzed in conjunction with %Opt and %Correct, not in isolation, as it is defined on the problems where we obtain a correct LLM-generated program. Note that the %Correct metric is equivalent to the commonly reported pass@k, when $n = k$. However, since our approach leverages unit tests in execution feedback, it is not fair to compare our correctness metric with those from previous works [29] that do not assume access to unit tests, and instead reserve them for correctness evaluation. Our study presents an alternative formulation where we seek to generate a program with optimal runtime efficiency, given all unit tests for a task. Our proposed framework can nonetheless be applied in settings where we do not have any unit tests, by generating synthetic unit tests for a problem using LLMs [4], [29]. For simplicity, we instead re-purpose the HumanEval, MBPP and APPS datasets which include tests commonly used to evaluate LLMs on programming tasks. Appendix VIII-A details our pre-processing and dataset sanitisation steps.

We include open weight LLMs of varying sizes in our experiments: Phi-3-mini 3.8B [34], Llama 3 8B [35], Mixtral 8x7B (13B active params, [36]), Command R 35B, Llama 3 70B. We also include the closed and commercial GPT-3.5 [37] and GPT-4 models via OpenAI APIs. We provide the details of our OpenAI API usage, LLM hosting and inference environment, and code execution environments in Appendix VIII-B. Note that given our choice of Python datasets, we are unable to compare with [24] where CodeLlama 13B was fine-tuned for generating C++ performance improvements.

B. PERFCODEGEN *Results*

We report the performance of all candidate LLMs on problems from HumanEval and MBPP (test) in Table I using our PERFCODEGEN framework and the aforementioned metrics, with a sampling budget of $k = 1$ and 8 samples. Gains in %Opt and %Correct observed by applying our PERFCODEGEN framework are indicated in the relevant cells

($\pm\delta$) for each model. On both the benchmarks, we witness that PERFCODEGEN leads to significant improvements in %Opt and %Correct for all candidate base LLMs at both the sampling budgets (k of 1 and 8). Statistical significance of these results is established in Section VIII-E.

An example of PERFCODEGEN's output for a HumanEval problem is shown in Listing 1, where it generates an efficient iterative approach that is faster than the reference program which takes a recursive approach. Listing 2 contains an example from MBPP (test) where PERFCODEGEN leads to an efficient closed-form solution for computing the average of cubes of the first n natural numbers, whereas the reference solution computes this result using an iterative approach.

```
'''
Problem:
The FibFib number sequence is a sequence
similar to the Fibbonacci sequnece
that's defined as follows:
fibfib(0) == 0
fibfib(1) == 0
fibfib(2) == 1
fibfib(n) == fibfib(n-1) + fibfib(n-2) + fibfib(n-3)
    .
Please write a function to
efficiently compute the n-th element
of the fibfib number sequence.
>>> fibfib(1)
0
>>> fibfib(5)
4
>>> fibfib(8)
24
'''
# Optimal solution generated by PerfCodeGen
def fibfib(n: int):
    if n == 0 or n == 1:
        return 0
    if n == 2:
        return 1
    (a, b, c) = (0, 0, 1)
    for _ in range(3, n + 1):
        (a, b, c) = (b, c, a + b + c)
    return c
# Original ground truth solution in HumanEval:
def fibfib(n):
    if n == 0:
        return 0
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fibfib(n - 1) + fibfib(n - 2)
    + fibfib(n - 3)
```

Listing 1. An optimal solution generated by PERFCODEGEN in HumanEval

```
'''
Problem: Write a python function to find the
average of cubes of the first n natural numbers.
'''

# Optimal solution by PerfCodeGen:

def solution(n):
    sum_of_cubes = (n*(n+1)/2.0)**2
    return sum_of_cubes/n


# Original ground truth (suboptimal) in MBPP:

def solution(n):
    sum = 0
    for i in range(1, n + 1):
        sum += i*i*i
    return round(sum / n, 6)
```

Listing 2. An optimal solution generated by PERFCODEGEN in MBPP

On HumanEval (Table I top), with PERFCODEGEN, we notably enhance the runtime-efficiency of programs generated by open weight models Phi-3-mini (18.20 %Opt at $k = 1$) and Llama 3 70B (25.61 %Opt at $k = 1$) making them comparable to GPT-4 in the base setting, which attains the highest base %Opt of 24.54 (at $k = 1$). Similarly, with PERFCODEGEN, open weight models Command R and Llama 3 8B achieve %Opt of 20.25 and 17.07 respectively, slightly better than GPT-3.5's base performance of 16.05 on %Opt. We continue to observe this trend at the higher budget of $k = 8$ samples.

While we elevate the performance of open weight models to match the base performance of closed commercial models, we witness even higher gains in %Opt and %Correct when using PERFCODEGEN on the closed GPT-3.5 and GPT-4 models. At $k = 1$, the gains in %Opt are 4.29 and 6.17 on GPT-3.5 and GPT-4 respectively, while the gains on open weight models are in the lower range of 1.22 to 3.66. Similarly, at $k = 8$, we witness gains in %Opt of 9.26 for GPT-3.5 and 7.37 and GPT-4, whereas the gains for open weight models are in the range of 1.22 to 8.17. This observation can be attributed to the differences in the reasoning capabilities of these model categories, as the effectiveness of the self-refinement stage heavily relies on the reasoning capability of the base LLM.

When increasing the sampling budget from $k = 1$ to 8, we observe improvements in the optimisation rate (%Opt). The gains with PERFCODEGEN over the base LLM performance are higher at $k = 8$ samples for all models than at the lower sampling budget of $k = 1$. We also note that despite the higher correctness and optimisation rate, PERFCODEGEN maintains similar speedup as the base LLM, demonstrating its ability to generate runtime-efficient or performant solutions for a larger number of problems than the base LLM.

On MBPP (Table I bottom), we continue to witness significant gains in %Opt when using PERFCODEGEN in all cases with both k of 1 and 8 samples. Only exception to this observation is with Llama 3 70B at $k = 1$, whose performance marginally drops on MBPP (test) with our method, likely due to the high variance in estimating %Opt with a single sample. This drop can be mitigated in practice by leveraging execution time evaluation to fall back to the base LLM output in cases where our refinement is correct but suboptimal. However, we avoid doing so here for a stricter evaluation of our scheme.

Our observations outlined before for HumanEval results also hold true for experiments on MBPP (test split). Open weight models with PERFCODEGEN often match the performance of closed models with base prompting. Optimisation rate (%Opt) increases with an increase in the sampling budget (k), and the gains in %Opt with PERFCODEGEN at $k = 8$ are larger than the gains at $k = 1$. Finally, we report larger gains in %Opt when using PERFCODEGEN with closed commercial

models than the gains with open weight models. We continue to witness improvements on %Opt and %Correct with all models and both the datasets when increasing the sampling budget to $k = 20$ as listed in Appendix Table V. We provide results on APPS (test) in Appendix VIII-D, where LLMs struggle to attain high correctness due to the higher difficulty of problems.

C. Alternatives to PERFCODEGEN's Execution Feedback

We evaluate some competitive prompting techniques for the performance improvement phase of PERFCODEGEN as alternatives to the verbalised execution feedback. Besides naive prompting, we evaluate 9 different prompting schemes for this phase using GPT-3.5 as the base model on tasks from the HumanEval and MBPP (test) datasets. We exclude other models from this analysis to avoid the high costs associated with LLM inference. We exclude the significantly larger APPS test set from this comparative analysis, which contains roughly 20x more problems, each with approximately 3x as many test cases on average, making it prohibitively expensive to perform a comprehensive analysis with multiple prompting schemes.

As possible strategies for the performance refinement phase, we first consider three single-round prompting methods. First, we use vanilla prompting for performance improvement, instructing the LLM to optimize the correct code while ensuring functional equivalence (**Perf Improvement Prompt**). Next, we evaluate a 3-shot **In-context learning** baseline, which includes three examples of program refinements along with optimization instructions. We also assess an improved prompt that includes common Python optimization tricks as **Pre-defined Strategies**, along with the usual optimization instruction. We then consider two multi-round approaches similar to those in [24], [23] which include a planning or analysis stage. In the **Plan and Refine** approach, we prompt the LLM to generate a plan for performance refinement, then prompt it again with this output plan to implement the proposed refinement. In **Analyze and Refine**, we prompt the LLM to first analyze the time complexity of the generated program, then prompt it again with this intermediate analysis to refine the code.

We also evaluate some multi-agent prompting approaches as adaptations of ChatDev [38] and MetaGPT [39] for code optimisation. First, we implement a multi-agent coder-reviewer setup for performance refinement (**Multi-Agent w/ Reviewer**), where a coder refines the base solution and a reviewer provides feedback, followed by another refinement attempt based on this feedback. Additionally, we implement a more elaborate variant with leader-coder-reviewer roles (**Multi-Agent w/ Team**), where the three agents take turns planning, refining, and reviewing code. Finally, we implement two variants leveraging execution feedback for performance improvements after the LLM attempts to refine the correct code solution. In the first variant (**Direct Execution Feedback**), we execute and evaluate the effectiveness of the refinement and verbalize the result (positive if the refinement is faster than the base, negative otherwise), feeding this feedback to the LLM for another refinement attempt. Finally, in **PERFCODEGEN**, as described in Section II, we provide the most time-consuming unit test as feedback to the LLM (prompt shown in Figure 3).

Table II lists the %Opt and Speedup results with all the methods on the HumanEval and MBPP datasets using GPT-3.5. We observe that the base model generations offer significant performance optimizations (16.05% on HumanEval and 44.1% on MBPP) over the ground truth. However, the gains with Perf Improvement Prompt, multi-round (Plan and Refine, Analyze and Refine), and Multi-Agent (w/ Reviewer, w/ Team) techniques are insignificant and often negative. This can be explained by the cascading of LLM errors [25] over multiple steps of reasoning. Direct execution feedback produces mixed results, with a %Opt gain of 5.55 on HumanEval and a drop of 1.31 on MBPP. In contrast, PERFCODEGEN results in substantial gains in optimization rate on HumanEval (6.17 gain in %Opt) and MBPP (3.06 gain in %Opt) problems, validating the higher performance-improvement effectiveness of execution feedback verbalised using the most time-consuming unit test.

D. Role of Planning in Correctness Refinement

In this section, we assess the utility of the planning step in PERFCODEGEN's correctness phase. Specifically, we compare the correctness achieved by using direct execution feedback without a planning step (refining using execution feedback only) with the correctness achieved using PERFCODEGEN that utilizes a planning stage after consuming the execution feedback (plan then refine). Prompts corresponding to these two approaches are presented in Figure 2 in the Appendix.

We implement the (direct) Testcase Feedback approach as follows: starting with the base LLM generation, we evaluate correctness based on available unit tests and instruct the LLM to refine its solution according to the environment output (Figure 2(b)). In contrast, PERFCODEGEN incorporates an additional planning step based on verbalized environment output (failure type on the unit tests). The generated plan is then included in the prompt for refinement in the subsequent step.

Results with these two approaches are shown in Table III on the HumanEval and MBPP datasets with GPT-3.5 and Llama 3 70B. Both approaches achieve higher correctness than the base LLM. With $k = 1$, we observe that the additional planning step of PERFCODEGEN often leads to slightly lower correctness gain compared to the direct approach. However, with a k of 8 and 20, we observe higher correctness rate with the planning step. Notably, PERFCODEGEN achieves the highest correctness rates on both benchmarks, underscoring the utility of its additional planning step. As discussed in Section II-A, a high correctness rate is essential for generating optimized solutions effectively across a larger proportion of problems. PERFCODEGEN is more likely to produce maximally optimal solutions by refining from a larger pool of correct candidate solutions, benefiting from the greater diversity within the seed set. These results also suggest that the planning step could also be effective when included in the performance refinement phase at the cost of additional LLM inference.

TABLE II
COMPARISON OF PERFCODEGEN ALTERNATIVES IN THE PERFORMANCE-REFINEMENT STAGE ($k = 1$ SAMPLE) USING GPT-3.5.

Prompting Method Summarized Instruction(s)	**HumanEval**			**MBPP (test)**		
	%Opt	($\pm\delta$)	**Speedup**	**%Opt**	($\pm\delta$)	**Speedup**
Base LLM Generation GPT-3.5 prompted to solve a problem	16.05	-	1.37	44.1	-	3.42
Perf Improvement Prompt Optimize given code, maintaining equivalence	19.14	(+3.09)	**1.39**	44.1	(+0.00)	3.45
In-context learning + Here's an example of optimisation: *{demo}*	19.14	(+3.09)	1.38	44.98	(+0.88)	3.20
Pre-defined Strategies + Here are common ways to optimize: *{strategies}*	16.67	(+0.62)	1.27	32.75	(-11.35)	2.86
Plan and Refine (a) Generate optim. plan (b) Optimize w.r.t. plan	19.14	(+3.09)	1.34	45.85	(+1.75)	3.30
Analyze and Refine (a) Analyze $\mathcal{O}$ time (b) Optimize given (a)'s analysis	18.52	(+2.47)	1.35	46.29	(+2.19)	3.32
Multi-Agent w/ Reviewer (a) Coder: Optimize (b) Reviewer: Suggest changes	18.52	(+2.47)	1.25	41.48	(-2.62)	3.57
Multi-Agent w/ Team (a) Leader: Plan optim (b) Coder Optimize w.r.t. plan (c) Reviewer: Suggest changes to b	20.37	(+4.32)	1.28	42.79	(-1.31)	3.20
Direct Execution Feedback (a) Optimize (b) It worked/didn't, try again	21.60	(+5.55)	1.38	42.79	(-1.31)	3.58
PERFCODEGEN (a) Optimize given code (b) Your costliest unit test is *{ test }*, optimize accordingly	**22.22**	(+6.17)	1.34	**47.16**	(+3.06)	**4.19**

TABLE III
%CORRECT USING GPT-3.5 AND LLAMA 3 70B AT k OF 1, 8 AND 20.

Method	**Best@1**	**Best@8**	**Best@20**
GPT-3.5 on HumanEval			
Base	54.94	78.40	84.57
Test Case Feedback	**72.84**	85.80	90.74
PERFCODEGEN	68.52	**90.12**	**93.83**
Llama 3 70B on HumanEval			
Base	75.61	84.76	89.02
Test Case Feedback	84.76	90.24	92.07
PERFCODEGEN	**85.37**	**93.90**	**95.12**
GPT-3.5 on MBPP (test)			
Base	66.38	76.86	79.48
Test Case Feedback	**78.60**	**90.83**	91.27
PERFCODEGEN	73.36	88.21	**92.58**
Llama3 70B on MBPP (test)			
Base	65.32	77.93	81.53
Test Case Feedback	**72.97**	87.39	91.89
PERFCODEGEN	68.47	**88.29**	**93.24**

IV. DISCUSSION

In this section, we present a discussion on the limitations and threats to validity of our work.

A. Limitations and Future Work

The challenge of writing performant and high-quality software with LLMs spans various levels of granularity, from line-level optimizations to multi-class project repositories [40]. In our current scope, we focus on generating performant modules or Python functions, which are typically small components of real-world systems. However, addressing this challenge comprehensively should ideally involve ensuring architectural design patterns such as minimal redundancy or wasteful computation across the entire scope of a project or repository.

Another limitation is our focus only on measuring the runtime performance of LLM-generated code, disregarding memory consumption, which can be a crucial consideration in many applications. Future extensions of PERFCODEGEN could prioritize optimizing for both aspects (runtime and space complexity) or allow users to specify preferences for optimization. Additionally, beyond performance, developers desire attributes like readability, ease of maintenance, security, and harmlessness [16], [41], which are not accounted for, within the scope of our current work. While our work could be adapted to incorporate feedback from different environments or tools evaluating these attributes, achieving a balance in optimizing code generation across these dimensions is non-trivial. Finally, we note that the effectiveness of our approach is heavily dependent on the code reasoning capabilities of the

underlying LLMs, and future work could consider curating training strategies specifically for performant code generation.

B. Threats to Validity

As emphasized in prior research, reliably measuring the runtime performance of code poses significant challenges [19]. A piece of code may exhibit varying execution times across different compute environments, even with identical underlying hardware. Tools like the gem5 simulator [33], that are used to determine the instruction count of a program, do not support the execution of Python programs to the best of our knowledge. To mitigate this, we ensure identical compute environments for each candidate code snippet and run only a single Python program at any given time to minimize effects from concurrent execution. However, averaging execution time measurements from 10 independent runs of each program significantly adds to our execution costs. Future work could explore more efficient methods for reliably measuring runtime, by determining the instruction count of LLM-generated programs deterministically.

V. Related Work

A large volume of prior work on improving code quality has proposed a code-to-code editing formulation in the form of tasks like fixing bugs [42], performance improving edits [19], [20], improving maintainability [43], [44], and security enhancing edits [45], [46], [47], [48], [41]. Contrary to this approach, we formulate a text-to-code task for our work on runtime efficiency aspect of quality improvements. As programmers continue to rely on prompting LLMs for generating programs for repetitive tasks in software engineering [17], [49], [50], [51], we opine that it is critical for research on code quality to focus on the prompting stage by studying natural language inputs that describe developer intent or program specifications.

To improve the general LLM output quality [52] post the pre-training and supervised instruction fine-tuning [53] stages, recently proposed algorithms like RLHF [37], [54] and DPO [55] that use human preference data have become industry standard [56]. While one could continue to scale these approaches for improving LLM-generated code quality, this would require gathering large-scale preference data for code, which is arguably more difficult and expensive than collecting natural language response preferences. Besides needing an extensive number of samples, RL techniques also involve expensive model fine-tuning and are known to be notoriously prone to training instabilities [57], [58]. A recent study [59] bypasses this labeled data limitation and proposes reinforcement learning using execution feedback (no human supervision) to improve the functional correctness of generated code. These training advancements have also been partially extended [60] to improve the performance aspect of LLM-generated code.

Our work builds upon the success of prior works like Self-Refine [24], Scratchpads for LLMs [23] and Self-Debug [28] that propose LLM based self-refinements to improve output quality by adding intermediate planning or analysis stages during inference. Our framework is also closely related to Reflexion [29] that uses environment or tool feedback to improve LLM output quality, but focuses only on functional correctness in the context of code generation. Scaling of inference time computation [61], [62] is emerging as a prominent theme in improving LLM output quality (functional correctness for code domain). With PerfCodeGen, we extend these ideas to improve program runtime efficiency, an aspect that has been largely ignored by predominant research on LLMs in favor of functional correctness.

VI. Conclusion

We introduce PerfCodeGen, a novel framework that leverages code execution feedback during the iterative self-refinement stages of LLMs to enhance the runtime-efficiency of generated code. We show that using our approach, open weight LLMs like Phi-3-mini can achieve code optimization rates comparable to naively prompting closed LLMs like GPT-4. Our evaluation of PerfCodeGen on three widely used Python programming benchmarks using both open weight and closed language models of varying sizes, demonstrates consistent and significant gains in correctness and runtime efficiency across tasks from the benchmarks studied. On a large proportion of the tasks from HumanEval and MBPP, we achieve programs with state-of-the-art code optimization rates using PerfCodeGen, leading to programs that are significantly faster than the reference solutions present in the datasets. We find that verbalised execution feedback including the most expensive unit test has significantly higher effectiveness in generating performant code than other complex multi-step and multi-agent prompting schemes. Our findings underscore the importance of integrating execution feedback into the code generation process, highlighting a path forward for more robust and reliable AI-driven software development.

VII. Data Availability

The code of PerfCodeGen is released at https://github.com/SalesforceAIResearch/perfcodegen.

VIII. Appendix

A. Sanitized Benchmarks

TABLE IV
Details of sanitized benchmarks used in the evaluation of PerfCodeGen.

Benchmark	#Problem	#Groundtruth	#Testcase
HumanEval	164	1.0	9.6
MBPP-test	257	1.0	3.0
APPS-test Original	5,000	30.4	21.2
APPS-test Sanitized	3,249	26.9	27.4

Detailed information on the 3 benchmarks we use is presented in Table IV. We use the sanitized benchmarks for correctness evaluation and the common subsets for time efficiency evaluation. In our evaluation process, we initially verify whether the provided ground truth programs within each benchmark can successfully pass the associated test cases. Notably, we identify 1,511 instances in the APPS benchmark

where the ground truth solutions fail to meet the test case criteria. This observation aligns with previous findings reported by [63]. To ensure a fair comparison among the different benchmarks, we exclude the aforementioned 1,751 instances from our analysis, retaining the remaining 3,249 instances for correctness evaluation.

Additionally, for assessing runtime efficiency, we construct subsets from the three benchmarks, comprising problems for which all baseline methods can generate at least one functionally correct program. Runtime-related data are computed solely on these subsets, rather than the entire benchmark, to maintain consistency in evaluation conditions. The metric of %Opt is then calculated only on the subset of problems where all baseline LLMs (i.e., the models being compared) have at least one correct solution. This ensures fairness in comparison by focusing on problems that all models can solve.

B. Compute

LLM inference: We use the vLLM [64] library on a node with 16 Nvidia A100 GPUs for approximately three weeks to complete all the experiments in this work.

OpenAI API: We use the gpt-4-0613 and gpt-3.5-turbo-0125 model endpoints from OpenAI. In total, we required nearly 411k GPT-4 and 1.5M GPT-3 requests for all our experiments, contributing to the major costs of this study.

Code Execution: We use 40 instances of virtual machines (n1-highmem-16 GCP instances), each with 16 CPUs and 104 GB RAM for executing all the LLM generated programs generated in our experiments. We employ these instances for roughly four weeks to complete the execution of all the LLM generated programs using different frameworks in our study. Interestingly, unlike most LLM research, gathering this environment feedback tends to be the much costlier bottleneck in our experiments compared to the LLM inference costs.

C. Best@20 Results of PERFCODEGEN for all Models

Results with PERFCODEGEN at the highest sampling budget of $k = 20$ in our experiments are provided in Table V.

TABLE V
PERFCODEGEN RESULTS ON HUMANEVAL, MBPP AND APPS WITH A k OF 20.

Model	%Opt%	%Correct	Speedup
Best@20 HumanEval			
Phi-3	47.56	92.68	1.81
Mixtral-8x7B	42.14	87.42	1.85
Command R	46.63	87.12	2.01
Llama3 8B	43.9	87.2	2.27
Llama3 70B	45.73	94.51	1.84
GPT-3.5	43.21	93.83	**2.33**
GPT-4	**55.21**	**95.71**	1.98
Best@20 MBPP (test)			
Phi-3	59.91	89.22	3.40
Mixtral-8x7B	47.3	87.39	3.52
Command R	46.26	85.9	3.21
Llama3 8B	43.72	86.15	3.30
Llama3 70B	44.59	93.24	2.88
GPT-3.5	**75.98**	92.58	**4.25**
GPT-4	66.96	**95.65**	4.16
Best@20 APPS (test)			
Phi-3	0.65	21.46	2.83
Mixtral-8x7B	0.59	21.42	1.75
Command R	1.26	34.53	1.05
Llama3 8B	0.95	22.44	**2.86**
Llama3 70B	1.39	37.08	1.06
GPT-3.5	2.06	51.34	1.24
GPT-4	**2.95**	**72.18**	2.57

D. Results on APPS

Results on the APPS dataset are reported in Table VI. Unlike MBPP and HumanEval which mostly contain basic Python problems, the APPS [10] benchmark involves tasks of significantly higher difficulty including problems from competitions like IOI, USACO and ACM. Additionally, unlike MBPP and HumanEval where we have a single human written solution per problem, APPS offers over 25 reference solutions (on average per problem) authored by programmers competing on websites like Codewars, AtCoder, Kattis, and Codeforces. Hence, the difficulty in generating a solution that is more performant than the best solution from the human written set is much greater on APPS problems than the HumanEval and MBPP problems. Given this higher difficulty, we observe a significantly lower correctness rate compared to HumanEval and MBPP with all LLMs. With a k of 1, most models fail to generate a solution more optimal than the best ground truth reference for over 1% of the dataset. Nevertheless, using PERFCODEGEN's execution feedback, we observe modest gains in %Opt and speedup, and large gains in %Correct of GPT-3.5 and GPT-4 generations on APPS-test problems, whereas open models benefit less due to task difficulty and their limited reasoning skills. Notably, the gap in correctness rates between commercial GPT models and open models is significantly wider on APPS than on simpler problems from HumanEval and MBPP, highlighting the capability differences between the two sets of models.

E. Statistical Significance

Our main results are based on findings in Table I and Table VI, where we report that using PERFCODEGEN leads to significant gains in the %Opt metric compared to the base LLM. For results with the GPT-4 model, we compute the Z-scores to compare PERFCODEGEN's output with that of the base model: $-0.878\,(k = 1)$ and $-1.34\,(k = 8)$ on HumanEval, $-2.588\,(k = 1)$ and $-2.947\,(k = 8)$ on MBPP and $-1.8\,(k = 1)$ and $-2.675\,(k = 8)$ on APPS. The improvement obtained with PERFCODEGEN is thus statistically significant with $\alpha < 0.05$ on the MBPP and APPS problems, and with a lower confidence ($\alpha < 0.2$) on HumanEval which has a smaller number of problems (164).

F. Prompts

We list the prompts used in PERFCODEGEN to generate runtime-efficient code solutions in Figure 2 (Correctness Phase) and Figure 3 (Performance Phase).

TABLE VI
%CORRECT, %OPT, AND SPEEDUP RESULTS ON APPS WITH k OF 1 AND 8.

Best@1 - APPS (test)				
Model	**Method**	**%Opt** ($\pm\delta$)	**%Correct** ($\pm\delta$)	**Spdup**
Phi-3-mini	Base	0.09	6.51	1.03
	Ours	0.09 (+0.0)	8.40 (+1.89)	1.00
Mixtral-8x7B	Base	0.12	6.33	0.97
	Ours	0.19 (+0.07)	7.08 (+0.75)	1.95
Command R	Base	0.31	14.27	1.00
	Ours	0.43 (+0.12)	16.48 (+2.21)	1.00
Llama 3 8B	Base	0.18	9.38	2.29
	Ours	0.25 (+0.07)	10.21 (+0.83)	2.62
Llama 3 70B	Base	0.37	18.65	1.04
	Ours	0.31 (-0.06)	19.30 (+0.65)	1.04
GPT-3.5	Base	0.49	25.18	**1.33**
	Ours	0.58 (+0.09)	30.56 (+5.38)	1.32
GPT-4	Base	0.31	36.63	0.98
	Ours	**0.61** (+0.30)	**45.62** (+8.99)	1.26
Best@8 - APPS (test)				
Model	**Method**	**%Opt** ($\pm\delta$)	**%Correct** ($\pm\delta$)	**Spdup**
Phi-3-mini	Base	0.28	13.77	1.86
	Ours	0.40 (+0.12)	16.73 (+2.96)	2.02
Mixtral-8x7B	Base	0.31	11.02	1.08
	Ours	0.40 (+0.09)	14.50 (+3.48)	1.81
Command R	Base	0.58	24.86	1.00
	Ours	0.86 (+0.28)	28.82 (+3.96)	1.00
Llama 3 8B	Base	0.52	15.8	3.07
	Ours	0.61 (+0.09)	17.71 (+1.91)	3.07
Llama 3 70B	Base	0.65	26.12	1.04
	Ours	0.86 (+0.21)	27.94 (+1.82)	1.04
GPT-3.5	Base	1.11	39.92	1.25
	Ours	1.48 (+0.37)	46.26 (+6.34)	1.24
GPT-4	Base	1.14	57.75	1.15
	Ours	**1.96** (+0.82)	**65.80** (+8.05)	**1.90**

Round 1:
Your generated solution for the problem is not correct and cannot pass the following test case:

{testcase}

The error message is as follows:

"{error}"

Could you please analyze the reason of failure and propose a strategy to modify your solution so that it can pass the above test case?
Round 2 :
Could you please modify your solution so that it can fulfill the requirements in the problem and pass the test case?
Give your solution as follows. Wrap it with "'python"'.

(a) Reflection and Test Case Feedback

Your generated solution for the problem is not correct and cannot pass the following test case:

{testcase}

The error message is as follows:

"{error}"

Could you please modify your solution so that it can fulfill the requirements in the problem and do not have any syntax error?
Give your solution as follows. Wrap it with "'python"'.

(b) Test Case Feedback

Fig. 2. The two correctness phase prompts discussed. PERFCODEGEN's Reflection and Test Case feedback prompt (Plan then Refine) in (a). (Direct) Test Case Feedback in (b).

Round 1: Good job! You generated the correct solution for the problem! Now let's step further and optimize the time performance of the solution.
Based on the correctly generated solution, could you please refine it so that it consumes less time in the execution?
Please make sure your refined solution is functionally equivalent with the original solution and do not change the input-output format and the name of the major components.
Give your solution as follows. Wrap it with "'python"'.
Round 2:
We tested your optimized program and found that the following test case costs the most time in execution.

{testcase}

Could you please refine your optimized program according to the test case below?
Please make sure your refined solution is functionally equivalent with the original solution and do not change the input-output format and the name of the major components.
Give your solution as follows. Wrap it with "'python"'.

Fig. 3. PERFCODEGEN Testcase Feedback prompt used.

REFERENCES

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[2] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[3] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *ICLR*, 2023.

[4] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," 2022. [Online]. Available: https://arxiv.org/abs/2207.10397

[5] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," 2024.

[6] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," *arXiv preprint arXiv:2404.05427*, 2024.

[7] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "Metagpt: Meta programming for a multi-agent collaborative framework," 2023.

[8] C. Qian, X. Cong, W. Liu, C. Yang, W. Chen, Y. Su, Y. Dang, J. Li, J. Xu, D. Li, Z. Liu, and M. Sun, "Communicative agents for software development," 2023.

[9] I. Shani, "Github research blog: Survey reveals AI's impact on the developer experience," 2023. [Online]. Available: https://github.blog/2023-06-13-survey-reveals-ais-impact-on-the-\developer-experience/

[10] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with apps," *NeurIPS*, 2021.

[11] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[12] R. Puri, D. S. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker *et al.*, "Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," *arXiv preprint arXiv:2105.12655*, 2021.

[13] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[14] D. Landes and R. Studer, "The treatment of non-functional requirements in mike," in *European software engineering conference*. Springer, 1995, pp. 294–306.

[15] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering*. Springer Science & Business Media, 2012, vol. 5.

[16] M. Singhal, T. Aggarwal, A. Awasthi, N. Natarajan, and A. Kanade, "Nofuneval: Funny how code lms falter on requirements beyond functional correctness," *arXiv preprint arXiv:2401.15963*, 2024.

[17] T. Dohmke, M. Iansiti, and G. Richards, "Sea change in software development: Economic and productivity analysis of the ai-powered developer lifecycle," *arXiv preprint arXiv:2306.15033*, 2023.

[18] W. Harding and M. Kloster, "Coding on copilot: 2023 data suggests downward pressure on code quality," https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality, 2023, accessed: 2024-05-20.

[19] A. Madaan, A. Shypula, U. Alon, M. Hashemi, P. Ranganathan, Y. Yang, G. Neubig, and A. Yazdanbakhsh, "Learning performance-improving code edits," *CoRR*, vol. abs/2302.07867, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.07867

[20] S. Garg, R. Z. Moghaddam, C. B. Clement, N. Sundaresan, and C. Wu, "Deepperf: A deep learning-based approach for improving software performance," *arXiv preprint arXiv:2206.13619*, 2022.

[21] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 21 314–21 328, 2022.

[22] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[23] M. Nye, A. J. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan *et al.*, "Show your work: Scratchpads for intermediate computation with language models," *arXiv preprint arXiv:2112.00114*, 2021.

[24] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang *et al.*, "Self-refine: Iterative refinement with self-feedback," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[25] J. Huang, X. Chen, S. Mishra, H. S. Zheng, A. W. Yu, X. Song, and D. Zhou, "Large language models cannot self-correct reasoning yet," *arXiv preprint arXiv:2310.01798*, 2023.

[26] P. Laban, L. Murakhovs' ka, C. Xiong, and C.-S. Wu, "Are you sure? challenging llms leads to performance drops in the flipflop experiment," *arXiv preprint arXiv:2311.08596*, 2023.

[27] M. Sharma, M. Tong, T. Korbak, D. Duvenaud, A. Askell, S. R. Bowman, N. Cheng, E. Durmus, Z. Hatfield-Dodds, S. R. Johnston *et al.*, "Towards understanding sycophancy in language models," *arXiv preprint arXiv:2310.13548*, 2023.

[28] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[29] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao, "Reflexion: Language agents with verbal reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[30] Z. Gou, Z. Shao, Y. Gong, Y. Shen, Y. Yang, N. Duan, and W. Chen, "Critic: Large language models can self-correct with tool-interactive critiquing," *arXiv preprint arXiv:2305.11738*, 2023.

[31] S. Welleck, X. Lu, P. West, F. Brahman, T. Shen, D. Khashabi, and Y. Choi, "Generating sequences by learning to self-correct," *arXiv preprint arXiv:2211.00053*, 2022.

[32] T. R. Sumers, S. Yao, K. Narasimhan, and T. L. Griffiths, "Cognitive architectures for language agents," *arXiv preprint arXiv:2309.02427*, 2023.

[33] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[34] M. Abdin, S. A. Jacobs, A. A. Awan, J. Aneja, A. Awadallah, H. Awadalla, N. Bach, A. Bahree, A. Bakhtiari, H. Behl *et al.*, "Phi-3 technical report: A highly capable language model locally on your phone," *arXiv preprint arXiv:2404.14219*, 2024.

[35] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[36] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," *arXiv preprint arXiv:2401.04088*, 2024.

[37] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.

[38] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun, "Chatdev: Communicative agents for software development," *arXiv preprint arXiv:2307.07924*, 2023. [Online]. Available: https://arxiv.org/abs/2307.07924

[39] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "MetaGPT: Meta programming for a multi-agent collaborative framework," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VtmBAGCN7o

[40] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *International Conference on Machine Learning*. PMLR, 2023, pp. 31 693–31 715.

[41] M. Bhatt, S. Chennabasappa, C. Nikolaidis, S. Wan, I. Evtimov, D. Gabi, D. Song, F. Ahmad, C. Aschermann, L. Fontana *et al.*, "Purple llama cyberseceval: A secure coding benchmark for language models," *arXiv preprint arXiv:2312.04724*, 2023.

[42] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the aaai conference on artificial intelligence*, 2017.

[43] B. Loriot, F. Madeiral, and M. Monperrus, "Styler: learning formatting conventions to repair checkstyle violations," *Empirical Software Engineering*, vol. 27, no. 6, p. 149, 2022.

[44] N. Al Madi, "How readable is model-generated code? examining readability and visual inspection of github copilot," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[45] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1865–1879.

[46] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2785–2799.

[47] C. Tony, M. Mutas, N. E. D. Ferreyra, and R. Scandariato, "Llmseceval: A dataset of natural language prompts for security evaluations," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 588–592.

[48] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[49] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[50] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, "Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design," *arXiv preprint arXiv:2303.07839*, 2023.

[51] P. Denny, V. Kumar, and N. Giacaman, "Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language," in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 1136–1142.

[52] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, H. Zhang, B. Zhu, M. Jordan, J. E. Gonzalez, and I. Stoica, "Chatbot arena: An open platform for evaluating llms by human preference," 2024.

[53] J. Wei, M. Bosma, V. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," in *International Conference on Learning Representations*, 2022.

[54] N. Stiennon, L. Ouyang, J. Wu, D. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. F. Christiano, "Learning to summarize with human feedback," *Advances in Neural Information Processing Systems*, vol. 33, pp. 3008–3021, 2020.

[55] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[56] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[57] S. Casper, X. Davies, C. Shi, T. K. Gilbert, J. Scheurer, J. Rando, R. Freedman, T. Korbak, D. Lindner, P. Freire *et al.*, "Open problems and fundamental limitations of reinforcement learning from human feedback," *arXiv preprint arXiv:2307.15217*, 2023.

[58] Y. Wang, Q. Liu, and C. Jin, "Is rlhf more difficult than standard rl? a theoretical perspective," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[59] J. Gehring, K. Zheng, J. Copet, V. Mella, T. Cohen, and G. Synnaeve, "Rlef: Grounding code llms in execution feedback with reinforcement learning," *arXiv preprint arXiv:2410.02089*, 2024.

[60] D. Nichols, P. Polasam, H. Menon, A. Marathe, T. Gamblin, and A. Bhatele, "Performance-aligned llms for generating fast code," *arXiv preprint arXiv:2404.18864*, 2024.

[61] B. Brown, J. Juravsky, R. Ehrlich, R. Clark, Q. V. Le, C. Ré, and A. Mirhoseini, "Large language monkeys: Scaling inference compute with repeated sampling," *arXiv preprint arXiv:2407.21787*, 2024.

[62] M. Hassid, T. Remez, J. Gehring, R. Schwartz, and Y. Adi, "The larger the better? improved llm code-generation via budget reallocation," *arXiv preprint arXiv:2404.00725*, 2024.

[63] S. Dou, Y. Liu, H. Jia, L. Xiong, E. Zhou, W. Shen, J. Shan, C. Huang, X. Wang, X. Fan, Z. Xi, Y. Zhou, T. Ji, R. Zheng, Q. Zhang, X. Huang, and T. Gui, "Stepcoder: Improve code generation with reinforcement learning from compiler feedback," 2024.

[64] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.