This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3709872

On the Time Efficiency of Code Generated and Optimized by Large Language Models

Yun Peng, Jun Wan, Xiaoxue Ren*, Tim Menzies, Michael R. Lyu

***Abstract*—Code generation has largely improved development efficiency in the era of large language models (LLMs). With the ability to follow instructions, current LLMs can be prompted to generate and optimize code solutions given detailed descriptions in natural language. Many research efforts are devoted to improving the correctness of LLM-generated code by proposing different benchmarks. Despite the focus on correctness, the time efficiency of LLM-generated code solutions is underexplored. Current correctness benchmarks are not suitable for time efficiency evaluation since their test cases cannot well distinguish the time efficiency of different code solutions. Besides, the current execution time measurement is not stable and comprehensive, threatening the validity of the time efficiency evaluation.**

To address the challenges in the time efficiency evaluation, we propose COFFE, a benchmark for evaluating the time efficiency of LLM-generated code solutions in code generation and code optimization. COFFE contains 398 and 358 problems for function-level and file-level code generation, respectively. To improve the distinguishability, we design a novel stressful test case generation approach with contracts and two new formats of test cases to improve the accuracy of generation. To improve the robustness, we also design a test case input scale mutation method to generate test cases with different input scales. For the time evaluation metric, we propose efficienct@k based on CPU instruction count to ensure a stable and solid comparison between different solutions. We evaluate 19 popular LLMs and 7 LLM-based code optimization methods on COFFE and identify 10 findings. Based on the findings, we draw some implications for LLM researchers and software practitioners to facilitate future research and usage of LLMs in code generation and optimization.

***Index Terms*—Code Generation, Code Optimization, Large Language Model, Code Efficiency, Test Case Generation**

I. INTRODUCTION

Nowadays, large language models (LLMs) such as GPT-4 [80] and Llama3.1 [71] have demonstrated great ability to solve different software engineering tasks. With the ability to follow instructions [13], [73], [85], [108], [112], LLMs can act like human developers, promptly handle the instructions and generate completed code, reviews, or comments. By writing new code and refining existing code based on natural language descriptions, code generation and optimization can significantly enhance the efficiency of software development.

Yun Peng is with Institute of Systems for Advanced Computing, Fudan University (Email: yunpeng4@sigsoft.org).

Jun Wan and Xiaoxue Ren are with the School of Software Technology, Zhejiang University, and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security (Email: 22451014@zju.edu.cn; xxren@zju.edu.cn).

Tim Menzies is with North Carolina State University (Email: timm@ieee.org).

Michael R. Lyu is with The Chinese University of Hong Kong (Email: lyu@cse.cuhk.edu.hk).

*Xiaoxue Ren is the corresponding author.

They are thus critical software engineering problems being studied by many researchers. Researchers have proposed different approaches to make use of LLMs on code generation and code optimization via prompting engineering [8], [11], [33], [47], [52], [59], [69], [75], [89], [94], [98], [102], [120], multiple-agent cooperation [43], [44], [46], [104], [113], [118], and retrieval augmentation [34], [65], [87], [101], [116], [121].

To facilitate the evaluation of LLM-generated code, many benchmarks such as HumanEval [9], MBPP [5], CodeContests [60], and APPS [42] have been proposed to evaluate the correctness of generated code solutions, and we refer to them as correctness benchmarks. These benchmarks include coding tasks drafted by experienced developers [5], [9] or collected from coding competitions [42], [60], with several test cases for each problem to examine the correctness of LLM-generated code solutions. With the correctness benchmarks, researchers can thoroughly study and further improve the ability of LLMs to generate correct code. Built upon current advanced techniques, powerful LLMs such as GPT-4 (Aug 2024 version) have obtained remarkable performance with the Pass@1 of 87.2% on the function-level benchmark HumanEval [9], reported by the EvalPlus leaderboard [63][1].

However, correctness benchmarks alone are insufficient to comprehensively evaluate LLMs' ability to generate and refine code, especially when these models are increasingly used to generate code solutions for software products [115]. In real-world software development, both correctness and time efficiency are crucial for ensuring software quality. Correct but time-inefficient code can lead to a lot of CWE issues [14]. Recent work [99] on LLM-based code generation steps further to generate correct and efficient code, and recent work [33], [89] on code optimization focuses on improving the time efficiency based on LLMs. They directly adopt existing correctness benchmarks and measure the execution time of LLM-generated code solutions to determine the time efficiency. We argue that current correctness benchmarks are not suitable for time efficiency evaluation for the following challenges:

Challenge 1: Existing correctness test cases cannot well distinguish the time efficiency of different code solutions.

C1.1: Limited Distinguishability. Test cases in correctness benchmarks usually have small input scales since they aim to cover most corner cases to detect potential logical errors in code. However, such test cases can hardly distinguish the time efficiency of different code solutions since code with different time complexities may cost similar time under small inputs. Therefore, it is necessary to include test cases with

[1]Data collected in May 2025.

larger inputs so that we can better distinguish code solutions with different time efficiency. We refer to such test cases as stressful test cases.

C1.2: Limited Robustness. Existing test cases have similar input scales, so they prefer code solutions with better time efficiency under certain input scales, leading to potential evaluation bias. To ensure the robustness of time efficiency evaluation, it is necessary to include test cases with different input scales to evaluate the time efficiency of code solutions comprehensively.

The generation of multiple input-scale stressful test cases is not straightforward and cannot be simply handled by existing correctness test case generation methods. Stressful test cases usually consume much more time, so traditional execution-based test case generation methods with many iterations of complete executions are too time-consuming to be adopted. LLM-based test case generation methods without execution can generate stressful test cases quickly, but they are limited by context windows and can hardly maintain the long inputs in results, threatening the accuracy of stressful test case generation. Furthermore, both existing execution-based and LLM-based test case generation methods do not consider the impacts of input scales, so they are unable to control the input scales of generated test cases.

Challenge 2: Execution time metric is unstable and not comprehensive for time efficiency evaluation. Unlike correctness evaluation, which can be easily repeated on any computer machine, execution time measurements highly rely on the machine where the experiments are conducted. Shypula *et al.* [99] find that two single time measurements of the code solution on the same environment can differ as much as 1.91×. Unstable execution time measurements threaten the validity of time efficiency evaluation. Besides, previous work [49], [99] regards time efficiency evaluation as independent of correctness evaluation for code generation, but time efficiency evaluation is conducted upon correctly generated code solutions. Using separate metrics to evaluate the correctness and time efficiency makes it hard to comprehensively evaluate the code generated or optimized by LLMs. For example, in code optimization, it is hard to compare a method that refines code solutions with higher correctness but lower time efficiency, and another method that refines code solutions with higher time efficiency but lower correctness. Currently, no single metric evaluates both the correctness and time efficiency of LLM-generated code.

To address the first challenge, we **1) propose a new time efficiency benchmark named COFFE that provides stressful test cases with different input scales for a distinguishable and robust time efficiency evaluation.** Specifically, COFFE is built upon existing correctness benchmarks HumanEval [9], MBPP [5] for function-level code generation, and CodeContests [60] and APPS [42] for file-level code generation. Hence, it contains two splits for function-level and file-level code generation. We adopt a "*generate-and-mutate*" strategy to generate stressful test cases with different input scales for COFFE, in which we first implement STGEN to generate stressful test cases with largest input scales within a time budget (5s for each test case) to ensure the distinguishability (C1.1), and then implement STMUT to mutate the input scales of generated stressful test cases to cover the smaller input scales to ensure the robustness (C1.2).

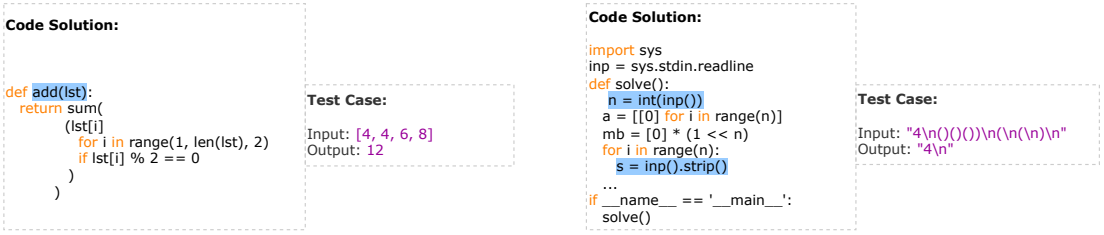
STGEN implements three phases to improve the accuracy of stressful test case generation. In the first phase, STGEN generates contracts that record the dependencies between inputs, and contracts are then used to guide the test case generation in the second phase. An LLM judge checks conflicts between generated contracts and test cases and rejects incorrect test cases in the third phase. By validating test cases on contracts, STGEN can identify incorrect test cases early and provide feedback for LLMs to help fix them. STGEN also uses expressions and generator functions to replace the raw inputs in the stressful test cases to avoid overlong test cases that hinder the generation of LLMs. Given the stressful test cases generated by STGEN, we further design STMUT to mutate their input scales to provide COFFE with test cases of different input scales. STMUT implements description-based and statement-based mutation methods to reduce the input scales, by considering the instructions in problem descriptions and the statements with the largest time costs in code. It then classifies the mutated test cases into different input levels for evaluation.

To address the second challenge, we **2) propose a new metric named *efficient@k* that considers both correctness and time efficiency based on *CPU instruction count* measurements.**. Efficient@k follows the same logic as pass@k [9], and the difference is that it requires a code solution to be correct and faster than the best ground truth solution to contribute. When comparing code solutions and ground truth solutions, we replace execution time with a more stable measurement *CPU instruction count* to conduct a solid comparison.

Evaluation. Experiments demonstrate that STGEN is quite effective in stressful test case generation by correctly generating 89.05%-98.91% of test cases with a 78.68%-96.01% line coverage. Furthermore, the stressful test cases generated by STGEN can much better distinguish the time efficiency of code solutions by achieving the relative standard deviation (RSD) of 39.11% and 30.67% over different function-level and file-level code solutions generated by DeepSeek V3 [20], largely improving the RSD of 23.02% and 18.07% on the original correctness test cases. Evaluation results also verify the effectiveness of STMUT in mutating the input scales of stressful test cases by generating 48.59% and 27.16% test cases with significant time differences for function-level and file-level problems, respectively. The effectiveness of STGEN and STMUT ensures the high quality of COFFE. To verify the stability of CPU instruction count, we compare it with execution time and find that CPU instruction count has an RSD of 0.003%-0.005%, which is 1,000× smaller than that of execution time measurement (2.37%-13.27%). This provides a solid basis for the calculation of efficient@k.

Findings. Based on COFFE, we evaluate the time efficiency of code solutions generated by 15 open-source LLMs and four closed-source LLMs, and the code solutions optimized by seven LLM-based code optimization approaches. We identify the following important findings:

- The performance of current LLMs drops significantly



(a) Function-level Code Generation (b) File-level Code Generation

Fig. 1. Examples for function-level and file-level code generation.

in efficient code generation, indicating that the code solutions generated by current LLMs are correct but not time-efficient.
- Compared with function-level code generation, code solutions generated by current LLMs are less efficient and robust in file-level code generation.
- Correctness decrease is the primary threat to current LLM-based approaches in function-level code optimization, while the effectiveness to further improve the time efficiency is the major challenge for current approaches in file-level code optimization.

Contributions. We summarize the contributions of this paper as follows:

- We build COFFE, a benchmark for evaluating the time efficiency of both function-level and file-level code solutions generated and optimized by LLMs.
- We propose STGEN, the first LLM-based stressful test case generation approach that employs contract validation and test cases with expression and generator functions inputs to improve accuracy.
- We propose STMUT, an effective input scale mutation method to generate test cases with different input scales.
- We introduce a novel metric efficient@k, based on stable CPU instruction count measurement, to evaluate the correctness and time efficiency of the LLM-generated code solutions.
- We conduct extensive experiments to evaluate the quality of COFFE, the effectiveness of STGEN and STMUT, and the ability of current LLMs to generate efficient code and current LLM-based code optimization approaches to improve the time efficiency.

This paper is an extension of our FSE 2025 paper [90]. Compared with the preliminary version, we further study the robustness of code generated by current LLMs on stressful test cases with different input scales. Besides, we extend the research scope to LLM-based code optimization approaches that aim to improve the time efficiency of code. We make the following specific changes as compared with the previous version:

- We modified the abstract and the introduction to extend our research scope.
- We added a new section *Input Scale Mutation for Stressful Test Cases* for STMUT, a novel approach to generate stressful test cases with different input scales.
- We added two new research questions (RQ3, RQ5) in the evaluation for the effectiveness of STMUT and the performance of seven LLM-based code optimization approaches. We also added a subsection in RQ4 to discuss the robustness of LLM-generated code under different input scales.

II. PROBLEM DEFINITION

Currently, there are three types of code generation tasks: function-level, file-level, and repo-level code generation. We mainly focus on the first two types of code generation since repo-level code generation involves different modules in the repositories and third-party dependencies, making it hard to obtain solid time efficiency measurements. To better illustrate the differences between function-level and file-level code generation, we present two examples in Figure 1.

Function-level Code Generation. Function-level code generation takes natural language functionality descriptions as input and generates a single function that satisfies the requirements. The generated function accepts inputs through function parameters. The HumanEval [9] and MBPP [5] benchmarks are designed to benchmark function-level code generation.

Figure 1(a) shows an example function. We observe that the function *add()* only has a parameter named `lst`, and we only need to generate test inputs for this parameter to build a test case. This shows that the number of parameters in functions is determined and functions accept inputs only once from parameters before the function execution. Therefore, **to generate test cases for function-level code generation, we can generate test inputs for each parameter and combine them as a test case.**

File-level Code Generation. File-level code generation generates a complete program file instead of a single function to satisfy specified requirements. The inputs of the program file are managed by *standard input (stdin)* related APIs, e.g., *input()*. File-level code generation tasks frequently appear in coding competitions, based on which researchers built Code Contests [60] and APPS [42] benchmarks.

Figure 1(b) shows an example program file. We observe that this code solution accepts inputs in two locations (highlighted in blue). The input in the first location is used to control how many times the input in the second location will take. This indicates that **the number of inputs for program files is not only determined by the code solution but also by the**

TABLE I
THE STATISTICS OF FOUR SANITIZED BENCHMARKS WE SELECTED TO BUILD COFFE. "ORI.", "VAL." AND "SEL." INDICATE THE ORIGINAL PROBLEMS, VALIDATED PROBLEMS, AND FINALLY SELECTED PROBLEMS IN THE BENCHMARKS. THE OTHER COLUMNS IN THE TABLE REPRESENT THE DATA FOR THE FINALLY SELECTED PROBLEMS.

Benchmark	#Problem			#Solution/Problem	#Test Case/Problem	Level
	Ori.	Val.	Sel.			
HumanEval	164	164	164	1.00	9.57	Function
MBPP	234	234	234	1.00	3.02	Function
Code Contests	111	106	58	80.26	197.53	File
APPS	5,000	3,106	300	64.36	13.94	File

inputs. This poses great challenges in generating test cases for file-level code generation.

As code optimization shares the same outputs with code generation, we use the same terms "function-level" and "file-level" to describe the types of code optimization in this paper without further definitions.

III. METHODOLOGY

This section describes how we build the benchmark COFFE, including selecting the coding problems, proposing STGEN to generate stressful test cases for function-level and file-level code generation, and designing a novel time efficiency metric efficient@k.

A. Data Preparation

To construct COFFE, we collect problems in the test splits of two existing function-level correctness benchmarks (i.e., HumanEval [9] and MBPP [5]), and two existing file-level correctness benchmarks (i.e., APPS [42] and CodeContests [60]). Each benchmark contains multiple coding problems and provides each problem with a description that explains the requirements in natural language, several ground truth solutions that address the problem, and several test cases that evaluate the correctness of generated code solutions. As there are multiple versions for MBPP, we choose the common subset of the sanitized version [38] and the MBPP+ benchmark verified by EvalPlus [62] as our base benchmark to ensure the highest quality.

With the selected benchmarks, we first validate the problems by checking the potential conflicts of the provided test cases and ground truth solutions. Secondly, we select problems that most LLMs could correctly answer to reduce the difficulty of problems for the two file-level benchmarks since a problem is not useful in time efficiency evaluation if no LLM can answer it. We show the statistics of four benchmarks in Table I.

1) Problem Validation: To ensure the quality of test cases and ground truth solutions in the four benchmarks, we run the ground truth solutions in the provided test cases and remove 1) ground truth solutions that cannot pass the provided test cases to ensure consistency, 2) ground truth solutions with file operations to keep safety, and 3) problems without valid ground truth solutions and test cases. We show the number of validated problems in each benchmark in the third column of Table I. All problems in HumanEval and MBPP can be successfully validated, so no problem is removed. For the Code Contests benchmark, we identify five problems with file operations, and we remove them to guarantee the safety of testing environments. For the APPS benchmark, we identify 1,894 problems whose ground truth solutions conflict with the provided test cases. The reason for such conflicts is that the APPS benchmark does not require the output of a code solution to exactly match the expected outputs in test cases to be correct, which differs from the other three benchmarks. We remove the 1,894 problems without exact matches in the APPS benchmark to maintain consistent evaluation standards.

2) Problem Selection: Current LLMs are quite effective in function-level code generation by achieving a pass@1 of more than 80% in the HumanEval benchmark, as discussed in Sec. I. However, they perform much worse in file-level code generation since the most powerful LLM has a Pass@1 of 28.5% on the Code Contests benchmark and a Pass@1 of less than 10% on the APPS benchmark [110], [111]. This limits the usage of the full set of the Code Contests and APPS benchmarks because a problem that no LLM can correctly answer does not contribute to the time efficiency evaluation. Therefore, for the validated problems in the two benchmarks, we sample one code solution with temperature 0 on 14 LLMs used in our experiments described in Table IV and remove 48 and 2,223 problems that code solutions from all LLMs failed in the Code Contests and APPS benchmark, respectively. To balance the number of problems in the function-level split and file-level split of COFFE, we further select 300 problems in the APPS benchmark for which more LLMs can generate correct code solutions. We show the number of selected problems from the four benchmarks and associated statistics in the 4∼6 columns of Table I.

B. Stressful Test Case Generation: STGEN

With the selected problems, we propose a novel LLM-based approach STGEN to generate stressful test cases automatically. In contrast to current LLM-based test case generation methods [7], [57], [61], [86], [96], [97], STGEN aims to generate test cases to evaluate the time efficiency of code solutions under extreme conditions rigorously. This inherently requires constructing exceptionally long and intricate inputs that can hardly be handled by LLMs directly, leading to unsatisfactory accuracy, i.e., the proportion of correctly generated stressful test cases is low.

1) Overview: To improve the accuracy of stressful test case generation, STGEN introduces contracts to guide the test case generation and validate the generated test cases. Contracts are collections of assertion statements that record the type, scale, and internal constraints between the inputs. Providing contracts in the test case generation process can help LLMs understand the dependencies between test inputs. Besides, STGEN can easily identify incorrect test cases from the assertion errors contracts raise. To avoid overlong stressful test cases that hinder the performance of LLMs, we design two new formats of test cases by reformulating the test case generation task into a code generation task: *expression test cases* and *generator test cases*. Different from raw test cases that directly provide test inputs, expression and generator test cases contain code to generate test inputs, which greatly shortens the length of test cases.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3709872

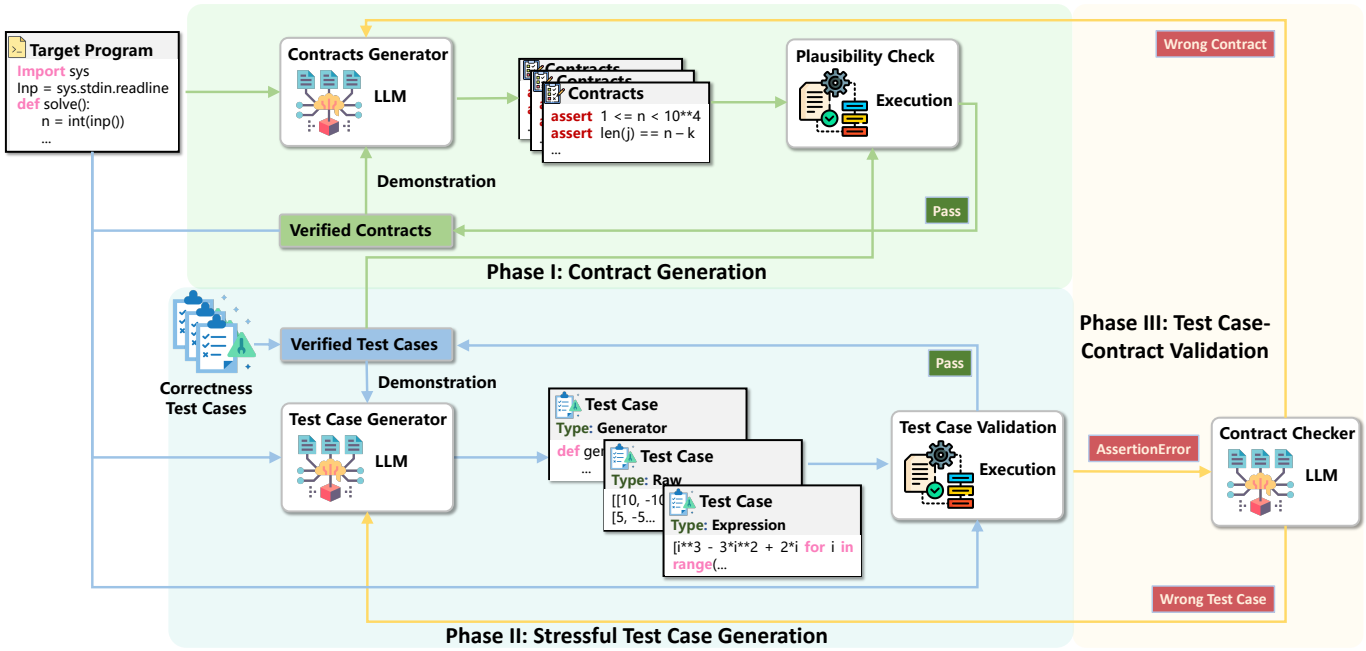


Fig. 2. The overview workflow of STGEN.

We present the overview of STGEN in Figure 2. STGEN does not directly generate stressful test cases. Instead, it decomposes the task into three phases: 1) contract generation, 2) stressful test case generation, and 3) test case-contract pair check. In the first phase, STGEN generates contracts by analyzing the target program, i.e., the ground truth solution for each problem in the benchmark. The generated contracts are then provided as demonstrations for stressful test case generation in the second phase, in which STGEN generates expression and generator test cases instead of raw test cases. Since contracts are also generated and there is no guarantee of their correctness, STGEN enters the third phase if the number of *AssertionError* occurrences for a certain contract exceeds a threshold. In the third phase, STGEN implements an LLM judge to determine the responsibility for conflicts between generated contracts and test cases. The contracts or test cases that are judged to be incorrect will be sent back for regeneration. This iterative process allows the generation of contracts and stressful test cases to mutually reinforce each other.

2) Phase I: Contract Generation: In the first phase, STGEN inserts assertion statements that check the preconditions of inputs as contracts into the target program, such as *assert* $n > 0$. The contracts ensure that the inputs meet the required specifications in format (e.g., variable type), scale (e.g., input length, order of magnitude), and intrinsic constraints (e.g., right triangle side lengths).

The benefits of inserting contracts before stressful test case generation are twofold: 1) **Knowledge Enrichment.** Contracts explicitly indicate the functionality of the target program and the dependencies between inputs, which can help LLM better understand natural language descriptions provided in problems [25], [61]; 2) **Early Validation.** Contracts can identify invalid inputs in test cases at the beginning of program execution and stop the execution-based test case validation process early, which largely improves the efficiency of the test case generation process.

In contract generation, STGEN generates one assertion statement in an iteration and combines all assertion statements into a contract. When generating assertion statements, STGEN prompts LLMs to consider the type, scale, and intrinsic constraints between inputs given the target program, existing correctness test cases, and previously generated assertion statements as demonstrations. STGEN implements the same methodology to generate assertion statements for function-level and file-level target programs. However, STGEN employs different strategies to insert assertion statements into target programs, given the differences between the code solutions in function-level and file-level code generation illustrated in Sec. II.

Function-level Contract Insertion. For function-level target programs with a determined number of inputs, STGEN generates and inserts assertion statements for function parameters at the beginning of the function body. For example, STGEN inserts assertion statements right before the return statements in the function *add()* in Figure 1(a).

File-level Contract Insertion. For file-level target programs with an unknown number of inputs and multiple input locations, STGEN reformulates the contract generation problem into a code editing problem. It first identifies all input locations by checking the related system APIs such as *input()* and then inserts assertion statements for each identified input location sequentially. STGEN inserts assertion statements right after the input locations in most cases. However, as input locations in loops generally assign values for generic types such as *list* and *dict*, STGEN inserts assertion statements after the entire loop where the assignments are complete to check the fully assigned types. For example, STGEN identifies two input locations highlighted in blue in Figure 1(b). STGEN first generates assertion statements for the input that assigns values to variable

n and inserts them right after the assignment. STGEN then generates assertion statements for the second input in the loop, and this time, it inserts assertion statements after the entire *for* loop.

To improve the correctness of generated assertion statements, STGEN tests all generated assertion statements against the correctness test cases each time it inserts a new assertion statement. If the current assertion statement fails on the test cases, STGEN only removes the current assertion statement and regenerates a new one while maintaining the assertion statements correctly generated in previous iterations. The iteration ends until no new assertion statements are generated or a maximum iteration number is reached.

3) Phase II: Stressful Test Case Generation: With the generated contracts as demonstrations, in the second phase, STGEN generates stressful test cases. Unlike correctness test case generation, it is quite challenging to generate stressful test cases because LLMs must generate test cases of maximal length and complexity within the constraints of its finite context window while simultaneously ensuring adherence to intrinsic input constraints specified by contracts. Correctness test cases in current benchmarks [5], [9], [42], [60] are raw test cases that directly provide the values for test inputs. However, due to the limited context window size, it is infeasible to directly generate overlong raw test cases for time efficiency evaluation. For example, it is hard for LLMs to generate a list with more than a million numbers for stressful tests. To address this challenge, we introduce two new formats of stressful test cases:

Expression Test Cases. Expression test cases utilize Python expressions to generate test cases, allowing for more complex input generation while maintaining a compact representation within the LLM's context window. For instance, a list with a million numbers could be easily generated by an expression "*[random.randint(1, 100000) for _ in range(1000000)]*", which is much shorter than listing a million numbers. Expression test cases offer a balance between complexity and conciseness, enabling the creation of structured inputs. They are suitable for function-level test case generation with a determined number of test inputs. To evaluate code solutions on expression test cases, we just need to execute the expressions to get the real test inputs before the code execution.

Generator Test Cases. Generator test cases are Python functions that output the test inputs. It is quite useful for creating stressful test cases that require intricate logical relationships or patterns that are difficult to express in single expressions. For example, it is suitable for file-level code generation where the number of inputs is undetermined. Expression test cases cannot handle this since we do not know how many expressions should be generated.

To generate expression and generator test cases, STGEN prompts the LLMs with contract, verified generated test cases as demonstrations, so LLMs can learn the dependencies between inputs as well as the specific formats of the expected test cases. The generated test cases are then verified against the previously generated contracts and the target program. Test cases that pass the validation of contracts and the execution of the target program are collected to build COFFE. Verified stressful test cases are also used as demonstrations to help generate the following stressful test cases.

4) Phase III: Test Case-Contract Pair Check: Although the generated contracts are verified against the existing correctness test cases, correctness test cases do not cover all possible cases and dependencies among inputs, especially in stressful scenarios. Contracts can still make mistakes and induce false positives. During the test case validation, if a generated test case violates the inserted contract, it triggers an *AssertionError*. If the *AssertionError* consistently occurs for multiple test cases, the contract may be incorrect and thereby hinder the entire stressful test case generation procedure. To mitigate this, when the number of conflicts between contracts and test cases (i.e., *AssertionError* occurrences during execution) exceeds a predefined threshold (5 in this paper), the generated test case and the violated contract are paired for further check by an LLM judge checker in the third phase.

The LLM judge takes all accumulated contract-related execution failure pairs as inputs, along with the target program, to analyze and determine the validity of the contracts and the test cases. The judge reviews the violated contract with exact stressful inputs, rethinks the correctness of the generated contract, and determines the root cause of conflicts. Once the root cause is identified, the relevant judgment results and corresponding failure pairs are sent back to the previous phases for regeneration. By providing feedback for incorrect contracts or test cases, STGEN enhances the robustness of the test case validation and enables the improvements between contract generation and test case generation. To prevent duplicate judgments, once the LLM judge determines that a contract is valid in the third phase, it will not be checked again, and test cases that fail the validation of this contract will be directly rejected in the future.

C. Input Scale Mutation for Stressful Test Cases

Stressful test cases with large-scale test inputs can significantly reduce the perturbation during performance measurements. However, stressful test cases are still insufficient to evaluate the robustness of code solutions. For example, some code may be specially optimized for large inputs but perform poorly under smaller inputs. We propose STMUT to mutate the input scales of the existing stressful test cases and observe the robustness of code generated or optimized by LLMs under different input scales.

1) Overview: We show the overview of our input scale mutation approach STMUT in Fig. 3. Specifically, for each problem in the benchmark, we take the correctness test cases in original benchmarks, the target programs with contracts, and stressful test cases generated by STGEN as inputs, and mutate the input scales of stressful test cases with two methods: *Description-based input scale mutation* and *statement-based input scale mutation*. The two methods will iteratively reduce the input scales of the input stressful test cases and generate the mutated test cases. With the mutated stressful test cases with different input scales, we perform an input level classification to classify the input scales of the mutated stressful test cases into different levels for evaluation.

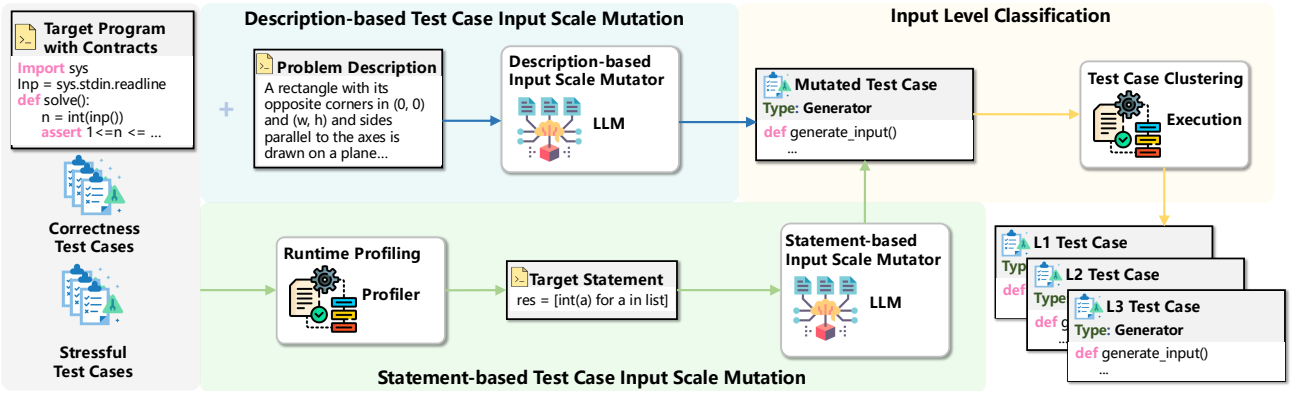


Fig. 3. Overview of input scale mutation.

2) Description-based Input Scale Mutation: Determining the input scale of existing stressful test cases is the key to performing input scale mutation. Description-based input scale mutation directly refers to the instructions in problem descriptions to determine the target input scales. To be specific, we ask the LLM to consider the input scale difference between one correctness test case and one stressful test case based on the problem descriptions of the target program, and then generate a new test case with an average input scale of both correctness and stressful test cases. The generated test case is validated against the target program with contracts, and the valid test case is regarded as the new stressful test case to perform mutation again. This process is conducted iteratively until a pre-defined depth is reached. The depth here affects the number of input levels the mutation method may generate, and we set it to 3 in this paper, as we find a larger depth does not contribute to more input levels due to the nature of problems.

3) Statement-based Input Scale Mutation: Description-based input scale mutation directly refers to the instructions in problem descriptions to determine the target input scales. However, this method sometimes could not produce significant time differences in the target program since the statements with high time costs may not be directly related to the problem descriptions. To handle such cases, we propose a statement-based input scale mutation method by identifying the statements with high time costs first and reducing the input scales based on the identified statements.

We randomly select one correctness test case and one stressful test case to identify the statements with high time costs. We run the two test cases on the target program and collect the runtime information with LineProfiler [84]. LineProfiler records the percentage of time each statement consumes (*time%*), the number of times each statement executes (*hits*), and the time each statement consumes in each execution (*time per hit*). We compare the runtime information between the correctness test case and the stressful test case and select the statement with the highest *time%* differences. Statements with high *time%* differences between correctness and stressful test cases are the most sensitive statements to the input scales and significantly affect the time efficiency of the entire program under different input scales. For the identified statements, we further analyze the *hits* and *time per hit* changes on both test cases. We ask the LLM to generate a test case to reduce the *hits* or *time per hit* of the identified statements by half. If the two metrics both change significantly, we prioritize *hits* as it is easier to control the number of executions for a statement rather than the execution time.

4) Input Level Classification: Given the mutated test cases, we collect the CPU instruction count measurements (discussed in Sec. III-D) on the target program and calculate the difference between them and the original stressful test cases. We follow the difference calculation and filtering method in previous work [99], i.e., the difference is calculated by $\frac{t_{ori}-t}{t_{ori}}$, where t_{ori} is the CPU instruction count consumed by the original stressful test case and t is the CPU instruction count consumed by the mutated test case. The difference is significant if it is larger than 10%, so we filter out the mutated test cases with a difference lower than 10%. To estimate how many levels of input scales exist in the remaining mutated test cases, we regard the test case with the smallest difference as input level L_1 and the test case with the largest difference as input level L_n where $t_{L_n} \leq (1-10\%)^n \times t_{L_1}$. To determine the intermediate levels between L_1 and L_n, we select the mutated test case that has the maximum $(L_1 - t)(t - L_n)$. This makes sure that the input levels are uniformly distributed. In most cases, the level number n is related to the depth we set in the input scale mutation methods. After determining each input level, we finally classify the mutated test cases into the nearest input level. In this paper, we classify the mutated test cases into three input levels, as we set the depth to 3 when mutating the test cases.

D. Time Efficiency Metric: Efficient@k

Previous work [49], [99] intuitively adopts execution time as the performance measurement to evaluate the time efficiency of LLM-generated code. However, execution time measurements could be affected by many factors, such as process scheduling and disk I/O, so it is not stable enough to make a solid comparison between the time efficiency of different code solutions. In this section, we propose to use *CPU instruction count* to replace execution time to measure the time efficiency of code solutions stably. Based on CPU instruction count measurements, we propose a new metric

efficient@k to evaluate both the correctness and time efficiency of code solutions.

1) CPU Instruction Count: To find a more stable measurement to replace execution time, we first look into the factors contributing to the execution time. Patterson and Hennessy [88] define the CPU time cost by a program through the following equation.

$$\text{CPU Time} = [\textbf{Instruction Count}] \times [\text{Clock per Instruction}] \times [\text{Clock Cycle Time}] \quad (1)$$

From the equation, the CPU time of a program is determined by three factors. While *Clock per Instruction* and *Clock Cycle Time* depend on the physical machine where the program runs, the only factor related to the program is *Instruction Count*. Therefore, if a program has a higher CPU instruction count on the same machine, it is less efficient, and vice versa. Unlike the execution time measurements that could be affected by many factors, CPU instruction count measurements are more stable as CPU instruction count for a program does not increase even if the program execution is slowed or stalled by external factors. It is also straightforward to measure CPU instruction count using the system APIs. For example, Linux provides a command tool named *perf* [26] to support CPU instruction count measurements.

2) Efficient@k: CPU instruction count is a stable measurement for the time efficiency evaluation of different code solutions. However, its absolute value is not meaningful as the same code solution has different CPU instruction counts in different machines. Besides, it is not comprehensive as it does not measure the correctness of generated code solutions. To address these problems, we propose a new metric named *Efficient@k*, inspired by the design of *pass@k* [9]. We show the original definition of pass@k in Equation 2 and the definition of proposed efficient@k in Equation 3.

$$\text{pass@k} := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \quad (2)$$

$$\text{efficient@k} := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c_f}{k}}{\binom{n}{k}} \right] \quad (3)$$

Pass@k is an expectation over all problems in the benchmark for the probability that at least one solution in k samples can pass all test cases. In equation 2, total n solutions are sampled from LLMs instead of only k samples to reduce the variance. By running the sampled code solutions on correctness test cases, we can get the solutions c that can pass all the test cases to estimate the probability of correctness. Pass@k is a solid metric with low variance and can be easily reproduced under different platforms.

We follow the idea of pass@k when designing efficient@k. Pass@k requires the correct code solutions c to contribute, while in efficient@k, we collect the number c_f of the correct solutions faster than the best ground truth solution to replace c in pass@k. Therefore, efficient@k evaluates the probability of LLMs to generate correct and fast enough code solutions. Efficient@k compares the CPU instruction count of code solutions and ground truth solutions to determine which runs faster. By doing so, efficient@k does not consider the absolute values of CPU instruction counts to avoid the impacts of specific systems or machines. With a value range from 0 to pass@k, efficient@k combines correctness and time efficiency evaluation to comprehensively evaluate the quality of code solutions.

TABLE II
THE STATISTICS OF COFFE.

Category	#Problem	#Solution /Problem	#Test Case/Problem	
			Correctness	Stressful
Function-level	398	1.00	5.72	4.99
File-level	358	66.93	43.68	4.95

TABLE III
THE STATISTICS OF MUTATED INSTANCES IN COFFE BY STMUT. "L1", "L2" AND "L3" ARE DIFFERENT INPUT LEVELS WHERE "STRESSFUL" > "L1" > "L2" > "L3" > "CORRECTNESS".

Category	#Problem	#Test Case/Problem		
		L1	L2	L3
Function-level	307	3.16	4.40	4.99
File-level	212	4.01	4.79	5.00

E. Code Efficiency Benchmark: COFFE

With the stressful test case generation approach STGEN, we add stressful test cases for each problem selected in Sec. III-A. Specifically, we generate 20 stressful test cases for each problem and measure the CPU instruction count each test case costs. We conduct the measurements 12 times and remove the highest and lowest measurements before calculating the average to ensure the most stable results. In the CPU instruction measurements, we limit the execution time of one single measurement to five seconds so that the measurements for one test case will not exceed one minute. We then rank the average CPU instruction count of each test case and include the five test cases with the highest CPU instruction counts in COFFE. We do not include all generated stressful test cases in COFFE to avoid large time costs in time efficiency evaluation, since stressful test cases generally take much longer time than correctness test cases to execute. We reserve all existing correctness in COFFE to validate the correctness of generated code solutions. We show the statistics of COFFE in Table II.

To further evaluate the robustness of code generated or optimized by LLMs under different input scales, we implement STMUT to mutate the stressful test cases in COFFE and generate mutated test cases with three input levels: "L1", "L2", and "L3", where "L1" is closest to the original stressful test cases and "L3" is the smallest input scale, i.e., "Stressful" > "L1" > "L2" > "L3". Note that stressful test cases of some problems in COFFE could not be mutated into three input levels by STMUT since the problems are not quite sensitive to inputs or involve many system I/O interactions that cost most time. We only include problems with three input scale levels

to ensure a fair comparison between different input levels. We show the statistics of mutated test cases in Table III.

IV. Experiment Setup

A. Research Questions

We focus on the following research questions:

- **RQ1:** How well does CPU instruction count measure time efficiency compared with execution time?
- **RQ2:** How effective is STGen on stressful test case generation and how good are the generated stressful test cases?
- **RQ3:** How effective is STMut on test case input scale mutation?
- **RQ4:** How efficient is the code generated by current LLMs?
- **RQ5:** How effective are current LLM-based code optimization approaches?

B. Metrics

To evaluate the stability of CPU instruction count (RQ1), we introduce the following metrics:

- **Relative Standard Deviation (RSD):** The ratio of the standard deviation to the mean. We use it to measure how stable a performance metric is on the same code solution (the lower, the better) and how well a test case can distinguish different code solutions (the higher, the better). We use "RSD (-)" when it is used to evaluate stability and "RSD (+)" when it is used to evaluate distinguishability.
- **Pearson Correlation Coefficient:** The ratio between the covariance of two variables and the product of their standard deviations. We use it to measure the linear correlation between two metrics.

To evaluate the quality of stressful test cases and the effectiveness of STGen and STMut (RQ2 and RQ3), we introduce the following metrics:

- **Accuracy:** The proportion of test cases generated by a certain method where the target program does not fail.
- **Line Coverage:** The percentage of executed lines in solutions when executing the test cases.
- **Diff%:** We design this metric to evaluate the effectiveness of test case input scale mutation approaches. For all mutated test cases, we select the largest subset where the CPU instruction counts of all test cases are significantly different (>10%) from each other. We calculate Diff% as the ratio of the size of the subset to the total mutated test cases, i.e., the proportion of mutated test cases with significantly different time costs.

To evaluate the efficiency of code solutions generated by LLMs (RQ4), we use the following metrics:

- **Pass@k:** The probability that at least one of the top k-generated code samples for a problem passes the unit tests, as illustrated in Sec. III-D.
- **Speedup:** The ratio $\frac{gt}{o}$ of CPU instruction count of best ground truth solution gt to the CPU instruction count of a code solution o.
- **Efficient@k:** The probability that at least one of the top k-generated code samples for a problem is correct and more efficient than the best ground truth solution, as introduced in Sec. III-D.

C. Baselines

Since there is no previous work on LLM-based stressful test case generation, we select three widely used LLM-based correctness test case generation methods and adapt them into stressful test case generation:

- **Instruction Prompting [103].** Wang *et al.* design several instruction prompt templates to ask LLMs to cover certain lines, branches, or paths of the code in test case generation. We modified their instruction prompt and let LLMs focus on stressful test case generation. This method generates raw test cases.
- **Few-shot Prompting [86].** Few-shot prompting adds several demonstrations to guide LLMs to generate similar test cases. This method generates raw test cases.
- **Generator-based Prompting [64].** Instead of directly generating test cases, this method prompts LLMs to generate a function that derives the test cases. We adapt this method to our stressful test case generation and let LLMs generate functions that produce stressful test cases. This method generates generator test cases.

As there is no previous work on LLM-based test case input scale mutation, we design two typical prompting techniques as baselines to evaluate the effectiveness of STMut (RQ3):

- **Instruction Prompting.** We design a simple instruction prompt to ask LLMs to generate test cases with significantly different input scales.
- **Few-Shot Prompting.** We provide LLMs with three examples showing test cases with different input scales and ask LLMs to generate more.

To evaluate the effectiveness of current LLM-based code optimization approaches (RQ5), we select the following baselines:

- **In-Context Learning Prompt**: We use the same in-context learning prompts from PIE [99] by selecting some slow-fast code pairs as demonstrations.
- **Chain-of-Thought Prompt**: We use the same chain-of-thought prompts from PIE [99].
- **SBLLM** [33]: SBLLM is a search-based LLM framework that enables iterative refinement and discovery of improved optimization methods for code optimization.
- **PIE** [99]: PIE provides a fine-tuning framework for code optimization, we use the model discussed in the paper as our baseline.
- **Self-refine** [69]: Self-refine is a prompt refinement approach and has been demonstrated great performance on code optimization.
- **EffiLearner** [47]: EffiLearner is a self-optimization framework that utilizes execution overhead profiles to improve the efficiency of LLM-generated code.
- **DeepSeek R1** [18]: LLMs with deep thinking abilities accept a simple instruction and conduct a complicated reasoning process before giving the answers. We select

TABLE IV
THE LLMS WE EVALUATE IN THIS PAPER. MODELS HIGHLIGHTED IN GRAY ARE CLOSED-SOURCE MODELS.

Model	Size	Context Size
Phi3 [1]	3.8B	128k
MagicCoder [109]	DS-6.7B/CL-7B	16,384
CodeLlama [95]	7B/13B/34B	16,384
Llama3 [70]	8B/70B	4,096
StarCoder [58]	15B	16,384
WizardCoder [68]	15B	2,048
Codestral [72]	22B	256k
Mixtral [51]	8×7B	32,768
Qwen2.5 [91]	72B	128k
Devstral [92]	123B	256k
DeepSeek V2 [17]	236B	128k
DeepSeek Coder V2 [122]	236B	128k
Llama3.1 [71]	405B	4,096
DeepSeek V3 [20]	671B	128k
DeepSeek R1 [19]	671B	128k
Claude 3.5 Sonnet [4]	-	200k
Gemini 1.5 Pro [15]	-	200k
GPT-3.5 [51]	-	16,385
GPT-4o [81]	-	128k

DeepSeek R1 to represent the performance of these LLMs on code optimization.

We use GPT-4o [81] as the base model for all prompt engineering baselines to keep a fair comparison.

D. Models

To investigate the efficiency of code generated by current LLMs, we select 19 popular models for evaluation. We show the model names, sizes, and context lengths in Table IV. For GPT-3.5 [79] and GPT-4o [81], we use the APIs provided by OpenAI [82] under engines "*gpt-3.5-turbo*" and "*gpt-4o*", respectively. For DeepSeek V2 [17] and DeepSeek V2 Coder [122], we use the APIs provided by DeepSeek [16] under engine "*DeepSeek-V2-0628*" and "*DeepSeek-V2-0724*", respectively. For Codestral [72] and Devstral [92], we use the APIs provided by Openrouter [83] under engine "mistralai/codestral-2508" and "mistralai/devstral-2512", respectively. For Claude 3.5 Sonnect [4], we use the APIs provided by Anthropic [3] under the engine "*claude-3-5-sonnet-20240620*". For Gemini 1.5 Pro, we use the APIs provided by Google [39] under the engine "*gemini-1.5-pro*" to generate code solutions. Due to limited computing resources, for open-source models larger than 13B, we use the API provided by Deep Infra [50] to generate code solutions.

E. Implementation Details

We conduct all experiments on a Linux machine with Ubuntu 20.04.4 LTS. It has an Intel(R) Xeon(R) Platinum 8358P CPU of 2.60G HZ with 128 cores and 2 TB memory. We use the Coverage.py [6] library to measure the line coverage of test cases, the Cirron [100] library to measure the CPU instruction count a program consumes, and the Line Profiler [84] library to collect the runtime information in code execution.

TABLE V
COMPARISON BETWEEN CPU INSTRUCTION COUNT AND EXECUTION TIME ON DIFFERENT BENCHMARKS. "RSD (-)" INDICATES THE AVERAGE RELATIVE STANDARD DEVIATION OF A CERTAIN METRIC WHEN RUNNING MULTIPLE TIMES ON THE SAME GROUND TRUTH SOLUTION. IT EVALUATES THE STABILITY OF DIFFERENT MEASUREMENTS. "CORRELATION" INDICATES THE PEARSON CORRELATION COEFFICIENT BETWEEN CPU INSTRUCTION COUNT AND EXECUTION TIME.

Benchmark	RSD (-)		Correlation
	CPU Instruction Count	Execution Time	
HumanEval	0.005%	5.65%	1.00
MBPP	0.004%	5.31%	1.00
Code Contests	0.003%	2.37%	0.99
APPS	0.003%	2.47%	0.96
Effibench	0.003%	13.27%	0.90

V. EXPERIMENT RESULTS

A. RQ1: CPU Instruction Count vs. Execution Time

To demonstrate that CPU instruction count is more suitable for time efficiency evaluation than execution time, we focus on two aspects: stability, which evaluates how solid the measurement is, and correlation, which evaluates how close two measurements are.

Stability. To compare the stability of CPU instruction count and execution time measurements, we run the ground truth solutions of the validated problems on the correctness test cases from the four correctness benchmarks. Note that we do not run them on our stressful test cases to ensure a fair comparison, since CPU instruction count is involved in building COFFE. We run each solution 12 times and remove the largest and smallest measurements. We then calculate the RSD of the remaining 10 measurements and show the results in the second and third columns of Table V.

As the experiments are repeated on the same ground truth solution and the same test cases, a lower relative standard deviation indicates a more stable measurement. From Table V, we can observe that execution time has an RSD of about 5% on function-level benchmarks HumanEval and MBPP and an RSD of about 2% on file-level benchmarks Code Contests and APPS. On the contrary, CPU instruction count has a more than 1000× smaller RSD (0.003%-0.005%) than execution time on four benchmarks. This indicates that the ten measurements of CPU instruction count remain almost the same on the same program, and CPU instruction count is quite stable in measuring time efficiency.

Apart from the four general code generation benchmarks, we also evaluate the stability of execution time measurement and CPU instruction count on Effibench [49]. Effibench is a code efficiency benchmark that includes efficiency-sensitive problems. From Table V, we observe that the RSD of execution time on Effibench is even larger than that on general code generation benchmarks. This further demonstrates the importance of a stable performance metric for evaluating code efficiency.

TABLE VI
COMPARISON BETWEEN DIFFERENT TEST CASES. "CORRECTNESS" INDICATES THE ORIGINAL CORRECTNESS TEST CASES. "INSTRUCTION", "FEW-SHOT" AND "GENERATOR" INDICATE THE STRESSFUL TEST CASES GENERATED BY THREE BASELINES, RESPECTIVELY. "RSD (+)" INDICATES THE RELATIVE STANDARD DEVIATION ACHIEVED BY TEST CASES ON DIFFERENT CODE SOLUTIONS GENERATED BY TWO POWERFUL LLMS, GPT-4O AND DEEPSEEK V3. IT EVALUATES THE DISTINGUISHABILITY OF DIFFERENT TEST CASES IN TERMS OF TIME EFFICIENCY.

Level	Method	Accuracy	Line Cov.	RSD (+)	
				DeepSeek V3	GPT-4o
Function	Correctness	-	98.46	23.02%	21.35%
	Instruction	83.67	83.87	32.80%	22.21%
	Few-shot	87.07	81.46	37.26%	20.61%
	Generator	86.91	81.84	28.12%	21.32%
	STGEN	**98.64**	**96.01**	**39.11%**	**28.20%**
File	Correctness	-	95.68	18.07%	12.99%
	Instruction	84.52	85.29	18.60%	11.04%
	Few-shot	65.17	66.53	18.63%	8.95%
	Generator	94.86	94.79	26.24%	13.17%
	STGEN	**98.91**	**95.17**	**30.67%**	**14.79%**
Effibench	Original	-	93.86	16.09%	19.01%
	Instruction	85.84	**82.13**	19.09%	25.51%
	Few-shot	80.57	60.15	23.72%	28.18%
	Generator	80.74	57.93	23.28%	29.51%
	STGEN	**89.05**	78.68	**32.77%**	**42.54%**

Correlation. To validate the linear correlation between CPU instruction count and execution time, as described in Equation 1, we calculate the Pearson correlation coefficient between CPU instruction count and execution time, as shown in the last column of Table V. We find that the correlations of the two measurements on all benchmarks are very close to 1.0. This indicates that the two measurements are linearly correlated and verifies the correctness of Equation 1 as the other two factors *Clock per Instruction* and *Clock Cycle Time* do not change in the same testing environment. Therefore, we can replace execution time with CPU instruction count to measure time efficiency.

Answer to RQ1: CPU instruction count is more suitable to evaluate time efficiency since it is much more stable than execution time by achieving a 1000× smaller RSD of 0.003%-0.005%, and it is linearly correlated with execution time with a Pearson correlation coefficient of 0.90-1.0.

B. RQ2: Effectiveness of STGEN *and Distinguishability of Stressful Test Cases*

To answer RQ2, we study the effectiveness of STGEN on stressful test case generation compared with three widely used LLM-based test case generation baselines. For the generated stressful test cases, we evaluate whether they can better distinguish different code solutions generated by LLMs. We show the main results of the comparison between STGEN and baselines in Table VI.

Effectiveness of STGEN. To study how contracts can improve the accuracy of test cases, we compare STGEN with three baselines without contracts and report the accuracy in the third column of Table VI for function-level and file-level splits of COFFE. We do not report the accuracy of the original test cases because they are manually drafted. From the table, we observe that STGEN achieves an accuracy of 98.64% and 98.91%, outperforming the baselines by up to 17.89% and 51.77% in function-level and file-level splits, respectively. It also achieves an accuracy of 89.05% on Effibench, which outperforms the baselines by up to 10.53%. This suggests that almost all stressful test cases generated by STGEN are correct. Without knowledge enrichment and early validation by contracts, on the contrary, baselines fail to generate about 5%-35% of stressful test cases.

Apart from accuracy, a correct test case is representative if it covers most lines of the target program. To ensure the quality of generated stressful test cases, we evaluate the line coverage and report the results in the fourth column of Table VI. We find that STGEN consistently achieves the highest line coverage of 96.01% and 95.17% for function-level and file-level stressful test cases, respectively. On Effibench, it also achieves a high coverage of 78.68%. This demonstrates that the stressful test cases generated by STGEN can thoroughly evaluate the time efficiency of the major code logic in target programs. We also note that the line coverage achieved by STGEN is slightly lower than that achieved by the original correctness test cases. This is reasonable because the number of stressful test cases is much smaller than that of correctness test cases in COFFE, as shown in Table II.

Distinguishability of Stressful Test Cases. To evaluate how well the stressful test cases generated by STGEN can distinguish the time efficiency of different code solutions, we sample 20 code solutions from two powerful LLMs, DeepSeek V3 and GPT-4o, for each problem in COFFE. We then run the sampled solutions on different test cases and collect the CPU instruction count usage. We calculate the RSD on the CPU instruction counts of the sampled 20 code solutions, and a higher RSD indicates better distinguishability. We report the RSD on the code solutions of two models at the fifth and sixth columns of Table VI.

Firstly, we observe that stressful test cases generated by STGEN improve the RSD of original correctness test cases by 69.90% and 32.08% on DeepSeek V3 and GPT-4o, respectively, at the function level, and the improvements are 69.73% and 13.86% on DeepSeek V3 and GPT-4o, respectively, at the file level. STGEN also outperforms all three baselines in terms of RSD on both DeepSeek V3 and GPT-4o. This demonstrates that stressful test cases generated by STGEN can better distinguish different code solutions than original correctness test cases and stressful test cases generated by baselines. Secondly, we find that the generator-based prompting method achieves higher RSD than other baselines. This verifies the effectiveness of generator test cases compared with raw test cases in time efficiency evaluation. However, the generator-based prompting method does not handle multiple parameters in function-level programs well, achieving only 86.91% accuracy. STGEN mitigates this problem by generating expression test cases that follow the formats of raw test cases but introduce small expressions to represent each input. As a result, the expression test cases generated by STGEN for function-level code solutions outperform the generator-based

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2026.3709872

TABLE VII
COMPARISON BETWEEN DIFFERENT TEST CASE INPUT SCALE MUTATION METHODS.

Level	Method	Accuracy	Line Cov.	Diff%
Function	Instruction	**99.97**	97.25	35.37%
	Few-shot	98.80	97.16	39.05%
	STMUT	99.39	96.49	**48.59%**
File	Instruction	97.23	94.40	24.60%
	Few-shot	98.07	94.82	26.74%
	STMUT	**98.31**	94.43	**27.16%**

prompting method by 39.08% and 32.27% in terms of RSD on DeepSeek V3 and GPT-4o, respectively.

We also compare the distinguishability of stressful test cases generated by STGEN with that of original test cases in Effibench, and show the results in Table VI. We observe that STGEN outperforms the original test cases in Effibench by 103.67% and 123.78% in RSD for the solutions of DeepSeek V3 and GPT-4o, respectively. The improvements achieved by STGEN on Effibench are even larger than those on COFFE. This further reinforces that stressful test cases are more effective for efficiency-sensitive problems.

Answer to RQ2: With knowledge enrichment and early validation by contracts, STGEN is quite effective in generating correct stressful test cases with an accuracy of 89.05%-98.91% and line coverage of 78.68%-96.01%. The expression and generator test cases generated by STGEN can better distinguish different code solutions' time efficiency with an RSD of up to 42.54% on GPT-4o.

C. RQ3: Effectiveness of STMUT

To answer RQ3, we study the effectiveness of STMUT by comparing it with two baselines. Since there are no LLM-based methods on test case mutation, we select the widely used instruction and few-shot prompting methods as our baselines. We provide the stressful test cases in COFFE as input and let each method generate 20 mutated test cases with different input scales. We then evaluate the mutated test cases and show the results in Table VII.

Firstly, we find that all methods in test case input scale mutation have an accuracy of 97.23%-99.97%, which is much higher than the accuracy of 65.17%-98.64%, achieved by the methods in stressful test case generation. Besides, the mutated test cases share quite similar line coverage of 96.49%-97.25% on function-level problems and 94.40%-94.82% on file-level problems, which is quite close to the line coverage of 96.01% on function-level problems and 95.17% on file-level problems achieved by the original stressful test cases. This demonstrates the effectiveness of LLMs on mutating existing test cases rather than generating new ones, and further verifies our "generate-and-mutate" strategy instead of directly creating ones with different input scales.

Secondly, STMUT outperforms baselines by up to 37.38% and 10.41% in terms of diff% for function-level and file-level problems, respectively. By leveraging statement-based input scale mutation, STMUT could identify the statement that is mostly sensitive to input and mutate the test case based on the statement. Therefore, STMUT achieves a significant improvement over the instruction prompting method, which only considers the problem description. The improvement of STMUT over the few-shot prompting method further verifies the necessity of understanding the target code in test case generation, which cannot be simply replaced by providing some examples as demonstrations.

Answer to RQ3: With statement-based input scale mutation, STMUT could generate 48.59% and 27.16% test cases with significant time differences for function-level and file-level problems, respectively, outperforming instruction and few-shot prompting methods.

D. RQ4: Time Efficiency of Code Generated by LLMs

1) Performance on Stressful Test Cases: Based on COFFE, we evaluate the time efficiency of code generated by different LLMs. We select 15 popular open-source LLMs and four popular closed-source LLMs, as shown in Table IV. We show the Pass@1, efficient@1, and speedup of all LLMs on COFFE in Table VIII. For efficiency@1 and speedup, we run the experiments 5 times and report the average values with 95% confidence intervals to obtain the most stable measurements.

Overall Time Efficiency. To evaluate the time efficiency of code generated by different models, we study the efficient@1 and speedup. We use efficient@1 to evaluate the probability of an LLM to generate a correct code solution faster than the best ground truth solution and speedup to evaluate how fast the correctly generated code solutions are compared with the best ground truth solutions. From Table VIII, we identify that DeepSeek R1 obtains the highest efficient@1 of 54.17% at the function level and Llama3.1 obtains the highest efficient@1 of 42.71% at the file level. As for the speedup, DeepSeek R1 achieves the highest speedup of 5.85 at the function level, and CodeLlama-13B obtains the highest speedup of 1.90 at the file level.

Finding 1: DeepSeek R1 and Llama3.1 have the highest probability of generating efficient code solutions with an efficient@1 of 54.17% and 42.71%, respectively. DeepSeek R1 and CodeLlama-13B generate the most efficient code solutions with a speedup of 5.85 and 1.90, respectively.

Correctness vs. Time Efficiency When comparing the correctness and time efficiency of code solutions generated by current LLMs, we find that the best efficient@1 are 54.17% and 42.71%, at the function level and file level, respectively, which are much lower than the best pass@1 of 81.66% and 90.78%. This indicates that almost half of the correctly generated code solutions are sub-optimal since they are less efficient than ground truth solutions. Furthermore, the speedups achieved by most LLMs in file-level code generation are lower than 1.0, and some LLMs even obtain a speedup of less than 0.1, indicating their generated code solutions are $10\times$ slower than ground truth solutions. This suggests that efficient code generation remains a significant challenge for current

TABLE VIII
THE CORRECTNESS AND TIME EFFICIENCY OF CODE SOLUTIONS GENERATED BY LLMS IN TABLE IV ON COFFE. EFFICIENCT@1 AND PASS@1 ARE CALCULATED UPON ALL INSTANCES IN COFFE, AND SPEEDUP IS CALCULATED ON CORRECT SOLUTIONS GENERATED BY MODELS. MODELS HIGHLIGHTED IN GRAY ARE CLOSED-SOURCE MODELS. "Δ" INDICATES THE DIFFERENCE OF EFFICIENT@1 AND PASS@1 IN PERCENTAGE (100% - EFFICIENT@1 / PASS@1).

Model	Size	Function-level			File-level		
		Efficient@1 (Δ)	Speedup	Pass@1	Efficient@1 (Δ)	Speedup	Pass@1
Phi3	3.8B	26.03 ±0.57 (40%)	2.54 ±0.04	43.47	5.50 ±1.32 (76%)	0.03 ±0.03	22.63
MagicCoder	DS-6.7B	21.16 ±0.76 (35%)	3.56 ±0.36	32.41	9.89 ±1.49 (57%)	0.10 ±0.01	22.91
	CL-7B	30.39 ±0.61 (35%)	3.38 ±0.03	46.48	3.47 ±1.09 (78%)	0.25 ±0.08	15.92
CodeLlama	7B	26.03 ±0.85 (33%)	2.37 ±0.08	38.69	3.09 ±0.82 (64%)	0.47 ±0.35	8.66
	13B	24.97 ±0.99 (40%)	2.15 ±0.78	41.71	0.68 ±0.34 (70%)	**1.90 ±0.61**	2.23
	34B	40.64 ±0.48 (37%)	3.31 ±0.14	64.74	18.48 ±3.08 (66%)	0.09 ±0.00	53.63
Llama3	8B	27.15 ±0.61 (36%)	3.51 ±0.27	42.46	0.00 ±0.00 (100%)	0.19 ±0.02	0.84
	70B	40.44 ±0.35 (40%)	3.08 ±0.15	67.59	30.94 ±5.44 (55%)	0.16 ±0.02	68.99
StarCoder	15B	37.10 ±1.04 (39%)	3.62 ±0.07	61.31	17.58 ±2.87 (66%)	0.06 ±0.03	51.11
WizardCoder	15B	28.37 ±0.44 (41%)	1.83 ±0.08	48.49	8.38 ±1.19 (59%)	1.06 ±0.69	20.67
Codestral	22B	43.97 ±0.44 (44%)	4.16 ±0.03	78.89	32.07 ±0.75 (59%)	0.20 ±0.00	79.05
Mixtral	8×7B	25.67 ±0.41 (43%)	4.89 ±0.17	44.72	7.57 ±0.94 (67%)	1.37 ±0.13	22.91
Qwen 2.5	72B	46.43 ±0.34 (41%)	4.98 ±0.04	78.89	34.75 ±0.80 (58%)	0.27 ±0.00	83.24
Devstral	123B	43.12 ±0.93 (44%)	4.42 ±0.03	76.38	36.48 ±0.80 (55%)	0.37 ±0.00	81.01
DeepSeek V2	236B	46.22 ±0.61 (41%)	2.60 ±0.13	78.39	35.26 ±4.07 (61%)	0.24 ±0.06	89.94
DeepSeek V2 Coder	236B	46.78 ±0.67 (41%)	2.48 ±0.03	79.90	35.10 ±4.98 (55%)	0.59 ±0.27	78.77
Llama3.1	405B	38.77 ±0.85 (42%)	1.36 ±1.29	67.34	**42.71 ±2.70** (52%)	0.52 ±0.29	89.11
DeepSeek V3	671B	46.13 ±0.65 (41%)	5.18 ±0.05	78.39	39.05 ±0.67 (55%)	0.51 ±0.33	86.59
DeepSeek R1	671B	**54.17 ±0.72** (34%)	**5.85 ±0.05**	**81.66**	41.45 ±0.53 (50%)	1.24 ±0.01	82.40
Claude 3.5 Sonnet	-	42.43 ±0.95 (45%)	1.88 ±2.10	77.64	33.14 ±4.21 (62%)	0.20 ±0.04	86.59
Gemini 1.5 Pro	-	44.50 ±0.77 (41%)	1.72 ±0.03	75.38	35.90 ±4.70 (52%)	0.16 ±0.02	75.44
ChatGPT	-	35.79 ±1.43 (48%)	1.24 ±0.85	68.19	33.42 ±4.04 (56%)	0.25 ±0.09	75.98
GPT-4o	-	43.69 ±0.73 (44%)	2.66 ±3.90	77.64	38.72 ±3.01 (57%)	0.73 ±0.39	**90.78**

LLMs, despite their remarkable performance in generating correct code.

Finding 2: The performance of current LLMs drops significantly in efficient code generation, with the best efficient@1 of 54.17% and 42.71% at the function level and file level, compared with that in correct code generation, with the best Pass@1 of 81.66% and 90.78%. This indicates that the code solutions generated by current LLMs are correct but not time-efficient.

Function-level Code Generation vs. File-level Code Generation. In function-level code generation, we observe that all LLMs achieve a speedup larger than 1.0, indicating that the code solutions generated by current LLMs are more efficient than ground truths. However, only three LLMs achieve a speedup larger than 1.0 in the file-level code generation. This suggests that current LLMs cannot generate faster code solutions than existing solutions in COFFE. Besides, the efficient@1 achieved by current LLMs at the function level is also better than that achieved by current LLMs at the file level. For example, the efficient@1 of Phi3 drops by 78.87% from function-level to file-level code generation. Furthermore, the performance drop from pass@1 to efficient@1 in function-level code generation is 33%-48%, compared with 50%-100% in file-level code generation. This indicates that current LLMs perform much worse in file-level efficient code generation than in function-level efficient code generation.

Finding 3: Compared with function-level code generation, code solutions generated by current LLMs are less efficient in file-level code generation, evidenced by the significantly lower speedup, lower efficient@, and larger performance drop from pass@1 to efficient@1.

Impacts of Different Model Sizes. To study the impact of different model sizes on the time efficiency of code solutions generated by current LLMs, we observe changes

TABLE IX
THE TIME EFFICIENCY OF CODE SOLUTIONS GENERATED BY LLMS IN TABLE IV ON COFFE. WE ONLY LIST THE EFFICIENT@1 HERE. MODELS HIGHLIGHTED IN GRAY ARE CLOSED-SOURCE MODELS. "L1", "L2" AND "L3" ARE THREE INPUT SCALE LEVELS STMUT GENERATES. EACH LEVEL HAS AT LEAST 10% CPU INSTRUCTION COST DIFFERENCE AND "STRESSFUL" > "L1" > "L2" > "L3" > CORRECTNESS. "RSD (-)" INDICATES THE RELATIVE STANDARD DEVIATION OF EFFICIENCT@1 ON THREE INPUT LEVELS. IT IS USED TO EVALUATE THE ROBUSTNESS OF CODE SOLUTIONS GENERATED BY DIFFERENT LLMS ON DIFFERENT INPUT SCALES.

Model	Size	Function-level				File-level			
		L1	L2	L3	RSD (-)	L1	L2	L3	RSD (-)
Phi3	3.8B	24.69 ±1.09	23.97 ±0.84	22.35 ±0.22	5.06%	7.08 ±0.00	6.98 ±0.26	6.13 ±0.00	7.76%
MagicCoder	DS-6.7B	21.04 ±0.61	20.52 ±0.95	19.87 ±0.29	2.86%	10.94 ±0.64	9.34 ±0.26	8.30 ±0.32	13.96%
	CL-7B	30.49 ±0.36	29.45 ±0.84	30.10 ±0.84	1.75%	5.38 ±0.32	4.62 ±0.26	3.68 ±0.26	18.68%
CodeLlama	7B	26.45 ±0.53	24.76 ±0.29	23.39 ±1.32	6.16%	3.87 ±0.49	2.83 ±0.41	2.36 ±0.00	25.59%
	13B	25.54 ±1.05	23.84 ±0.60	24.17 ±0.34	3.68%	0.94 ±0.00	0.47 ±0.00	0.47 ±0.00	43.30%
	34B	41.17 ±1.05	39.80 ±1.23	41.69 ±1.03	2.39%	18.11 ±0.79	16.51 ±0.41	13.87 ±0.52	13.25%
Llama3	8B	25.93 ±0.46	25.47 ±0.66	25.67 ±0.66	0.90%	0.00 ±0.00	0.00 ±0.00	0.00 ±0.00	-
	70B	43.52 ±0.36	41.76 ±1.09	42.61 ±0.97	2.06%	28.49 ±1.58	28.11 ±1.35	25.75 ±0.32	5.41%
StarCoder	15B	37.00 ±0.84	35.83 ±0.76	36.61 ±1.09	1.63%	18.30 ±0.76	17.45 ±0.41	16.98 ±0.41	3.81%
WizardCoder	15B	27.95 ±1.05	27.10 ±1.01	26.71 ±0.81	2.33%	4.81 ±0.26	4.62 ±0.26	4.53 ±0.32	3.07%
Codestral	22B	45.60 ±0.90	45.41 ±1.42	47.04 ±1.27	1.94%	36.51 ±0.32	34.25 ±0.67	30.19 ±0.93	9.52%
Mixtral	8x7B	24.69 ±0.53	24.04 ±0.92	23.91 ±0.46	1.73%	6.89 ±0.32	5.94 ±0.32	5.38 ±0.52	12.58%
Qwen 2.5	72B	47.23 ±0.76	47.23 ±0.76	48.40 ±0.84	1.42%	38.21 ±1.17	35.38 ±1.17	30.00 ±1.21	12.08%
Devstral	123B	45.60 ±0.50	44.76 ±1.42	45.15 ±1.17	0.93%	38.96 ±1.14	35.66 ±1.28	31.23 ±0.96	10.99%
DeepSeek V2	236B	46.06 ±0.61	46.12 ±0.84	46.78 ±0.93	0.86%	41.60 ±1.40	37.08 ±0.79	33.68 ±0.52	10.61%
DeepSeek V2 Coder	236B	46.97 ±0.97	45.41 ±0.89	45.99 ±0.53	1.71%	40.09 ±1.60	35.66 ±1.21	31.60 ±0.41	11.87%
Llama3.1	405B	41.43 ±0.97	40.07 ±0.29	38.57 ±0.22	3.57%	44.43 ±1.13	41.98 ±2.15	**40.09 ±1.85**	5.16%
DeepSeek V3	671B	47.62 ±0.53	45.67 ±0.88	47.75 ±0.36	2.48%	41.04 ±1.10	37.83 ±0.76	35.28 ±0.64	7.59%
DeepSeek R1	671B	**54.33 ±0.53**	**53.16 ±1.52**	**55.11 ±0.34**	1.81%	43.02 ±1.05	40.75 ±0.98	39.15 ±0.59	4.75%
Claude 3.5 Sonnet	-	41.56 ±1.23	42.08 ±0.53	41.76 ±0.72	**0.63%**	37.36 ±0.96	36.79 ±1.31	33.58 ±0.49	5.67%
Gemini 1.5 Pro	-	45.47 ±0.97	44.56 ±0.44	44.89 ±0.92	1.02%	28.49 ±1.35	27.45 ±0.64	27.55 ±1.41	2.06%
ChatGPT	-	35.83 ±0.81	36.87 ±0.44	39.41 ±0.95	4.93%	31.70 ±1.46	32.08 ±1.24	31.98 ±0.76	**0.62%**
GPT-4o	-	45.60 ±1.25	44.36 ±0.88	46.32 ±1.26	2.18%	**45.94 ±0.98**	**43.02 ±1.05**	33.40 ±0.76	16.09%

in pass@1 and efficient@1 as LLM size increases. In both function-level and file-level code generation, we find that larger LLMs generally generate more correct solutions, as evidenced by higher Pass@1. However, larger LLMs do not always generate more efficient code solutions. For example, in function-level code generation, CodeLlama-34b achieves an efficient@1 of 40.64%, which is quite close to the efficient@1 of 40.44% achieved by Llama3-70b and 38.77% achieved by Llama3.1-405b, but Llama3.1-405b is more than 10× larger than CodeLlama-34b. In file-level code generation, Codestral achieves an efficient@1 of 32.07%, which is also quite close to the efficient@1 of 35.26% achieved by DeepSeek V2, and 35.10% achieved by DeepSeek V2 Coder, but DeepSeek V2 is 10× larger than Llama3-70b.

Finding 4: Larger LLMs generally perform better in terms of Pass@1 but do not significantly outperform smaller LLMs in terms of efficient@1, indicating larger parameter sizes of current LLMs do not contribute much to efficient code generation.

2) Performance on Test Cases with Different Input Scales: To evaluate the robustness of code generated by current LLMs, we run the code solutions on mutated test cases across three input levels, as shown in Table III. We show the efficient@1 of LLM-generated code solutions on mutated test cases in Table IX for both function-level and file-level problems. We also run the experiments 5 times and report the average values with 95% confidence intervals. We include the relative standard deviation (RSD) of efficient@1 on three input levels to indicate the robustness of LLM-generated code solutions on different input levels.

Function-level Code Generation vs. File-level Code Generation. We first analyze the RSD achieved by different LLMs in function-level and file-level code generation. As test cases

with different input scales share the same problems, we can exclude the impact of the correctness here. From Table IX, we observe that the performance of LLM-generated code solutions on function-level and file-level problems differs significantly. In function-level code generation, all LLMs achieve an RSD of only 0.63%-6.16%, indicating the high robustness of their generated code across different input scales. On the other hand, file-level code generation introduces much higher RSDs from 0.62% achieved by ChatGPT to 43.30% achieved by CodeLlama-13b. When comparing the performance of the same LLM on function-level code generation and file-level code generation, almost all LLMs achieve higher RSD in file-level code generation than in function-level code generation. DeepSeek V2 even achieves the largest 12× difference of RSD, with 10.61% in file-level code generation and 0.86% in function-level code generation. This suggests that correct code solutions generated by current LLMs for file-level problems are not robust enough to handle different input scales. Therefore, file-level problems pose great challenges in generating either efficient or robust code for current LLMs.

Finding 5: Current LLMs are capable of generating both correct and robust code for function-level problems by achieving an RSD of 0.63% - 6.16% on different input scales. However, current LLMs cannot generate robust code for file-level problems, with an RSD up to 12× higher.

Time Efficiency vs. Robustness. By comparing the efficient@1 and RSD achieved by different LLMs, we find that LLMs that achieve better efficient@1 generally obtain smaller RSD in function-level code generation. For example, DeepSeek R1 achieves the best efficient@1 across all input scale levels and a very small RSD of 1.81% in function-level code generation. However, this observation is overturned in file-level code generation. Although GPT-4o achieves the best efficient@1 across most input scale levels, many LLMs, such as ChatGPT, DeepSeek R1, Llama3.1, and Gemini 1.5 Pro, achieve smaller RSD than it. This suggests that LLM-generated code solutions with good time efficiency may not exhibit good robustness under different input scales in file-level code generation.

Finding 6: LLMs show consistent performance on time efficiency and robustness in function-level code generation, while LLMs that generate the most efficient code do not necessarily generate the most robust code in file-level code generation.

In summary, based on the experimental results of 19 popular LLMs on COFFE, we study the time efficiency and robustness of function-level and file-level code solutions generated by these LLMs across six aspects. We find that generating efficient and robust code is much more challenging for current LLMs than generating correct code, especially for file-level problems. We also find that larger LLMs do not always generate more efficient code.

E. RQ5: Effectiveness of LLM-based Code Optimization Approaches

To answer RQ5, we implement LLM-based code optimization approaches on two targets: code solutions generated by GPT-4o and ground truth solutions in COFFE. Performance on code solutions generated by GPT-4o reveals the practical usefulness of existing approaches for directly generating efficient code from problem descriptions, while performance on ground-truth solutions indicates the maximum effectiveness of existing code optimization approaches when given correct code solutions as input. We collect optimized code solutions from existing LLM-based code optimization approaches and evaluate their time efficiency on both stressful test cases and those with different input scales. For efficient@1 and speedup, we run all experiments five times and calculate the average values under 95% confidence intervals.

1) Performance on Stressful Test Cases: We show the performance of seven LLM-based code optimization approaches on stressful test cases in COFFE in Table X.

Overall Effectiveness. We first analyze the overall effectiveness of all approaches in both function-level and file-level code optimization by observing the efficient@1 and speedup. In function-level code optimization, we find that DeepSeek R1 achieves the best efficient@1 of 52.10% on GPT-4o code optimization, and EffiLearner achieves the best efficient@1 of 54.72% on ground-truth code optimization. This indicates that they are about 50% likely to generate more efficient code. DeepSeek R1 and SBLLM achieve the highest speedups of 16.47 and 28.79 on GPT-4o and ground-truth code optimization, respectively. In file-level code optimization, EffiLearner achieves the best efficient@1 of 59.08% and 53.18% on GPT-4o and ground-truth code optimization, respectively. This demonstrates the superior performance of EffiLearner in optimizing file-level code solutions. It also achieves the highest speedup of 1.43 in ground-truth code optimization, while DeepSeek R1 achieves the highest speedup of 2.24 in GPT-4o code optimization.

In both function-level and file-level code optimization, we observe that existing LLM-based code optimization approaches improve time efficiency, achieving an efficient@1 of about 50%. Although LLMs show significant differences in performance for function-level and file-level code generation, existing LLM-based code optimization approaches do not necessarily perform better for function-level code optimization. Approaches such as ICL, SBLLM, and EffiLearner even perform better in file-level code optimization.

Finding 7: Existing LLM-based code optimization approaches are capable of successfully optimizing 50% of function-level and file-level code. EffiLearner is the best approach, achieving an efficient@1 of 42.65%-59.08%.

Negative Impacts of Current Approaches. Code optimization approaches are double-edged swords. On the one hand, they could produce more efficient code. On the other hand, they could also introduce inefficient or even incorrect code. Therefore, the effectiveness of code optimization approaches could be affected by decreases in both correctness

TABLE X
THE CORRECTNESS AND TIME EFFICIENCY OF OPTIMIZED CODE BY DIFFERENT CODE OPTIMIZATION APPROACHES ON COFFE. EFFICIENCT@1 AND PASS@1 ARE CALCULATED UPON ALL INSTANCES, AND SPEEDUP IS CALCULATED ON CORRECT SOLUTIONS GENERATED BY APPROACHES. "GT" IN THE "TARGET" COLUMN INDICATES THE OPTIMIZATION TARGET IS GROUND TRUTH SOLUTIONS IN COFFE, AND "4O" INDICATES THE OPTIMIZATION TARGET IS THE CODE SOLUTIONS GENERATED BY GPT-4O IN TABLE VIII. WE DEFINE Δ_{cor} AS CORRECTNESS DECREASE, I.E., 100%-PASS@1. WE DEFINE Δ_{eff} EFFICIENCY DECREASE, I.E., PASS@1 - EFFICIENT@1.

Approach	Target	Function-level				File-level			
		Efficient@1	Speedup	Pass@1	$\Delta_{cor}/\Delta_{eff}$	Efficient@1	Speedup	Pass@1	$\Delta_{cor}/\Delta_{eff}$
CoT	4o	22.59 ±1.59	1.03 ±0.01	56.96	1.25	26.28 ±0.74	1.47 ±0.03	73.85	0.55
	GT	44.97 ±0.31	7.53 ±0.07	70.85	1.13	30.73 ±0.42	1.24 ±0.01	66.48	0.94
ICL	4o	38.38 ±1.12	1.06 ±0.01	69.58	0.97	41.91 ±0.68	1.21 ±0.06	84.62	0.36
	GT	45.33 ±0.61	16.53 ±0.19	66.83	1.54	46.65 ±0.88	1.18 ±0.01	80.73	0.57
SBLLM	4o	20.52 ±0.73	0.82 ±0.03	49.84	1.71	27.32 ±1.25	1.13 ±0.02	59.69	1.25
	GT	40.50 ±0.95	**28.79 ±0.81**	66.58	1.28	28.38 ±1.03	0.85 ±0.01	58.66	1.37
PIE	4o	35.34 ±1.15	0.64 ±0.02	71.84	0.77	39.75 ±1.58	1.09 ±0.04	81.23	0.45
	GT	48.04 ±0.51	2.93 ±0.03	74.62	0.95	39.66 ±0.74	0.67 ±0.29	76.54	0.64
Self-refine	4o	23.50 ±1.23	12.70 ±7.24	50.49	1.83	25.05 ±1.92	1.50 ±0.01	56.92	1.35
	GT	35.43 ±0.85	5.30 ±0.08	62.81	1.36	23.07 ±0.94	1.11 ±0.01	54.75	1.43
EffiLearner	4o	42.65 ±0.52	8.40 ±2.82	85.44	0.34	**59.08 ±1.18**	1.32 ±0.30	**93.85**	0.18
	GT	**54.72 ±0.89**	8.58 ±0.03	**85.18**	0.49	**53.18 ±0.76**	**1.43 ±0.03**	**88.83**	0.31
DeepSeek R1	4o	**52.10 ±0.40**	**16.47 ±0.70**	**89.00**	0.30	55.69 ±0.60	**2.24 ±0.03**	87.08	0.41
	GT	53.82 ±0.34	14.61 ±0.01	72.86	1.43	48.49 ±0.80	1.33 ±0.01	69.55	1.45

and efficiency. To illustrate which factor most impacts existing approaches, we compare Δ_{cor} and Δ_{eff} and present their ratio in Table X. We define Δ_{cor}=*100% - pass@1* as the correctness decrease and Δ_{eff}=*pass@1-efficient@1* as the efficiency decrease. In function-level code optimization, we find that most approaches obtain a $\Delta_{cor}/\Delta_{eff}$ ratio higher than 1.0. This reveals that the major threat to them in function-level code generation is reduced correctness. In file-level code optimization, however, most approaches yield a $\Delta_{cor}/\Delta_{eff}$ ratio below 1.0, indicating that the major threat to existing approaches is their ineffectiveness in further optimizing code time efficiency.

Furthermore, we find that approaches that achieved the best efficient@1 in both function-level and file-level code optimization, such as EffiLearner, have very low $\Delta_{cor}/\Delta_{eff}$ ratios, ranging from 0.18 to 0.49. On the contrary, approaches that achieved the worst efficient@1, such as Self-refine and SBLLM, have very high $\Delta_{cor}/\Delta_{eff}$ ratios, ranging from 1.25 to 1.83. This further demonstrates that preserving high correctness is essential in code optimization.

Finding 8: Correctness decrease is the major threat to existing approaches in function-level code optimization, while the capability to further optimize the time efficiency is the major challenge for existing approaches in file-level code optimization.

2) Performance on Test Cases with Different Input Scales: We evaluate the robustness of seven LLM-based code optimization approaches across test cases with different input scales in Table VII and present the results in Table XI.

Robustness on Different Input Scales. We analyze the robustness of current approaches by observing the RSD they achieved in three input levels. In function-level code optimization, we identify that PIE and DeepSeek R1 achieve the lowest RSD of 0.55% and 0.60% for GPT-4o code and ground-truth code, respectively. This indicates that the two approaches can generate optimized code with consistent performance under different input scales. In file-level code optimization, SBLLM achieves the lowest RSDs of 1.55% and 1.11% for the GPT-4o code and the ground-truth code, respectively.

Similar to the performance of current LLMs in code generation, we find that current LLM-based code optimization approaches are generally more robust in function-level code optimization. Most approaches, except SBLLM and EffiLearner, achieve higher RSD in file-level code optimization. This suggests that existing LLM-based code optimization approaches cannot address the challenge of generating robust and efficient file-level code. Furthermore, code optimized from GPT-4o code achieves a higher RSD than code optimized from ground-truth code. We attribute this to the robustness of the code generated by GPT-4o. As shown in Table IX, GPT-4o achieves an RSD of 16.09% for file-level code generation.

Finding 9: Current approaches are not capable of robust file-level code optimization. The robustness of the input code could also impact the robustness of the optimized code.

Time Efficiency vs. Robustness. Based on the comparison between the most effective approaches and the most robust approaches in Table XI, we find that the most effective approach is not necessarily the most robust approach in code optimization. For example, EffiLearner and DeepSeek R1 achieve the best efficient@1 in function-level code optimiza-

TABLE XI
THE TIME EFFICIENCY OF OPTIMIZED CODE BY DIFFERENT CODE OPTIMIZATION APPROACHES ON COFFE. WE ONLY LIST THE EFFICIENT@1 HERE. "L1", "L2" AND "L3" ARE THREE INPUT SCALE LEVELS STMUT GENERATES. EACH LEVEL HAS AT LEAST 10% CPU INSTRUCTION COST DIFFERENCE AND "STRESSFUL" > "L1" > "L2" > "L3" > CORRECTNESS. "RSD (-)" INDICATES THE RELATIVE STANDARD DEVIATION OF EFFICIENCT@1 ON THREE INPUT LEVELS. IT IS USED TO EVALUATE THE ROBUSTNESS OF CODE SOLUTIONS GENERATED BY DIFFERENT LLMS ON DIFFERENT INPUT SCALES. "GT" IN TARGET INDICATES THE OPTIMIZATION TARGET IS GROUND TRUTH SOLUTIONS IN COFFE, AND "4O" IN TARGET INDICATES THE OPTIMIZATION TARGET IS THE CODE SOLUTIONS GENERATED BY GPT-4O IN TABLE VIII.

Approach	Target	Function-level				File-level			
		L1	L2	L3	RSD (-)	L1	L2	L3	RSD (-)
CoT	4o	24.39 ±1.60	22.11 ±0.45	21.46 ±1.35	6.79%	37.92 ±1.03	31.58 ±0.74	29.18 ±1.32	13.73%
	GT	46.38 ±0.46	45.67 ±0.72	46.06 ±0.73	0.77%	44.06 ±0.67	40.19 ±1.62	35.38 ±0.59	10.90%
ICL	4o	40.49 ±1.05	40.41 ±1.16	40.00 ±1.22	0.65%	49.18 ±2.54	47.54 ±1.07	43.93 ±2.94	5.73%
	GT	46.78 ±0.68	46.38 ±0.73	44.95 ±0.76	2.09%	57.74 ±0.67	54.72 ±0.41	49.62 ±1.57	7.60%
SBLLM	4o	19.59 ±0.23	19.84 ±0.83	21.87 ±0.23	6.12%	26.89 ±0.88	26.99 ±2.18	26.23 ±1.07	**1.55%**
	GT	41.95 ±0.34	42.54 ±1.66	39.28 ±0.61	4.21%	27.45 ±0.76	26.89 ±1.49	26.98 ±1.05	**1.11%**
PIE	4o	31.38 ±0.83	31.46 ±0.45	31.71 ±0.80	**0.55%**	40.55 ±2.22	40.55 ±1.76	38.58 ±2.01	2.85%
	GT	51.14 ±1.11	50.10 ±0.66	51.60 ±0.46	1.51%	41.42 ±2.05	40.57 ±2.55	39.43 ±1.88	2.47%
Self-refine	4o	25.28 ±0.83	23.90 ±1.87	24.31 ±1.57	2.89%	23.83 ±0.77	22.51 ±2.11	23.39 ±1.30	2.89%
	GT	37.13 ±0.95	36.35 ±0.46	34.59 ±1.41	3.61%	25.38 ±1.52	25.00 ±0.93	23.30 ±0.52	4.51%
EffiLearner	4o	46.10 ±0.68	43.58 ±0.66	40.81 ±1.16	6.08%	52.90 ±2.42	**51.48 ±1.30**	50.27 ±0.83	2.55%
	GT	**57.85 ±0.54**	**56.48 ±0.79**	55.18 ±1.17	2.36%	54.81 ±0.49	55.94 ±0.98	50.09 ±1.27	5.79%
DeepSeek R1	4o	**51.95 ±1.97**	**51.38 ±0.45**	**48.54 ±1.16**	3.61%	**54.21 ±0.74**	51.04 ±0.77	**50.60 ±1.32**	3.79%
	GT	56.35 ±0.64	55.77 ±0.53	**55.77 ±0.44**	**0.60%**	**57.64 ±0.96**	**56.32 ±1.47**	**50.57 ±0.26**	6.85%

tion, but they achieve higher RSD than PIE. Results from file-level code optimization also reinforce this finding, as the most effective approach, DeepSeek R1, achieves a higher RSD than SBLLM. This reveals that current LLM-based code optimization approaches cannot generate both efficient and robust code.

Finding 10: Existing approaches cannot simultaneously achieve the highest effectiveness and the highest robustness in code optimization.

In summary, based on the experimental results of seven LLM-based code optimization approaches on COFFE, we study the effectiveness and robustness of code optimized by these approaches in four aspects. We find that current approaches can successfully optimize half of GPT-4o and the ground-truth code. However, correctness decreases, and limited robustness remains a major threat to current approaches to code optimization.

VI. IMPLICATIONS

Based on the findings that we conclude in Sec. V, we provide some implications for researchers who build LLMs and practitioners who use LLMs in software development.

LLM Researchers. We identify that there is a large gap between correct code generation and efficient code generation. This indicates that the current LLM-generated code is correct but sub-optimal, and generating efficient code remains a great challenge, especially for file-level code generation. This challenge cannot be effectively mitigated by just increasing the model size of current LLMs. Furthermore, how to generate efficient and robust code still remains a great challenge for code generation and code optimization. We recommend that LLM researchers consider the code structure and semantics when improving the time efficiency of LLM-generated code. Besides, LLM researchers should also focus more on file-level code generation since current LLMs perform much worse on it than function-level code generation.

Software Practitioners. As LLMs are gradually adopted in software development in product environments, software practitioners face the problem of choosing LLMs. In function-level and file-level code generation, generally, code solutions generated by DeepSeek V2 Coder and Llama3.1-405b obtain the best time efficiency, respectively. However, we also find that some LLMs with middle sizes, such as Llama3-70b and CodeLlama-34b, achieve competitive performance. We recommend that software practitioners adopt mid-sized LLMs to obtain similar performance on efficient code generation with much lower computational costs. LLM-based code optimization approaches also help improve the time efficiency of LLM-generated code. We recommend that software practitioners also adopt approaches such as EffiLearner to further improve the time efficiency and robustness of code generated by current LLMs.

VII. LEAKAGE ANALYSIS

COFFE is built on widely used code generation benchmarks such as HumanEval [9] and MBPP [5], so the data leakage risk of these benchmarks may affect the validity of findings described in this paper. To investigate this issue, we collect four training sets at HuggingFace and compare their similarity with COFFE. Specifically,

TABLE XII
THE TLIS BETWEEN COFFE AND FOUR TRAINING SETS.

Level	Evol-Instruct	OSS-Instruct	Evol-instruction	Self-OSS
Function	0.41	0.19	0.24	0.20
File	0.02	0.02	0.01	0.01

we collect two training sets for MagicCoder [109]: `Magicoder-Evol-Instruct-110K` [30] and `Magicoder-OSS-Instruct-75K` [31], the training set for CodeLlama [95]: `Evol-instruction-66k` [29], and the training set for StarCoder [58]: `self-oss-instruct-sc2-exec-filter-50k` [32]. Note that we are unable to find the training sets for all LLMs, as many are not publicly available. We implement both problem-level and solution-level analysis to identify potential leakages.

A. Problem-level Analysis

Similarity Comparison. We follow previous work [105] and use the Test Leakage Indicator (TLI) to measure potential data leakage. TLI is the similarity score $S(t_i, r_j)$ between a test sample t_i and a training sample r_j, which is calculated as the fraction of common n-grams over the total n-grams in the test samples. We calculate TLI on 4 grams and show the results in Table XII. From the table, we observe that the TLIs between the file-level data in COFFE and the four training sets are all below 0.02. This indicates that there is no data leakage risk between the file split of COFFE and the existing training sets. On the contrary, the TLIs between the function-level data in COFFE and four training sets are much higher. A possible factor is that the problem descriptions in function-level data are much shorter and can easily lead to larger TLIs. We further analyze the data instances in the function split of COFFE and identify 33 instances with a TLI larger than 0.8 with four training sets. This suggests an overlap rate of about 10% between the function split of COFFE with four training sets. This is also reported by previous work [105].

Effectiveness of LLMs on Cleaned Dataset. To further analyze the impact of overlapping instances in the function split of COFFE. We create a cleaned version by removing the 33 similar instances and re-evaluate the effectiveness of current LLMs. We report the results in Table XIII. For efficient@1 and speedup, we run all experiments 5 times and report the average values with 95% confidence intervals.

By comparing Table XIII and Table VIII, we find that the changes of all LLMs in terms of Pass@1 are very small, with an absolute increase or decrease within 1%. While the 33 similar instances account for about 10% instances in the function split of COFFE, they have a limited impact on the correctness of code generated by current LLMs. This case is also applicable to MagicCoder, StarCoder, and CodeLlama, whose training sets may contain similar instances. Surprisingly, we find that the drop in efficient@1 is greater than that in Pass@1. This demonstrates that current LLMs are proposing more efficient solutions than the ground-truth solutions for the 33 similar instances. Furthermore, there is no significant

TABLE XIII
THE CORRECTNESS AND TIME EFFICIENCY OF CODE SOLUTIONS GENERATED BY LLMS IN TABLE IV ON THE FUNCTION SPLIT OF COFFE, BY REMOVING THE 33 SIMILAR INSTANCES. MODELS HIGHLIGHTED IN GRAY ARE CLOSED-SOURCE MODELS. "Δ" INDICATES THE DIFFERENCE OF EFFICIENT@1 AND PASS@1 IN PERCENTAGE (100% - EFFICIENT@1 / PASS@1).

Model	Size	Efficient@1 (Δ)	Speedup	Pass@1
Phi3	3.8B	24.14 ±0.52 (45%)	2.37 ±0.04	43.72
MagicCoder	DS-6.7B	17.65 ±0.77 (41%)	3.87 ±0.15	30.05
	CL-7B	27.28 ±0.36 (40%)	3.89 ±0.28	45.63
CodeLlama	7B	23.59 ±0.83 (38%)	2.60 ±0.06	37.98
	13B	22.82 ±1.12 (45%)	2.34 ±1.02	41.80
	34B	36.36 ±0.44 (43%)	3.23 ±0.09	63.66
Llama3	8B	25.41 ±0.65 (40%)	3.56 ±0.18	42.62
	70B	36.77 ±0.62 (46%)	2.84 ±0.03	67.49
StarCoder	15B	33.38 ±1.00 (45%)	3.09 ±0.06	60.93
WizardCoder	15B	26.43 ±0.59 (47%)	1.93 ±0.06	49.45
Codestral	22B	42.57 ±0.65 (46%)	4.13 ±0.03	78.89
Mixtral	8×7B	24.19 ±0.41 (47%)	3.96 ±0.12	45.36
Qwen 2.5	72B	45.57 ±0.37 (43%)	4.87 ±0.04	79.65
Devstral	123B	42.57 ±1.03 (44%)	5.38 ±0.04	76.38
DeepSeek V2	236B	41.34 ±0.71 (47%)	2.34 ±0.10	77.60
DeepSeek V2 Coder	236B	42.04 ±0.70 (47%)	2.22 ±0.01	79.23
Llama3.1	405B	34.54 ±0.87 (48%)	1.10 ±1.15	66.67
DeepSeek V3	671B	45.14 ±0.65 (43%)	5.59 ±0.04	78.89
DeepSeek R1	671B	**53.99 ±0.51 (34%)**	**6.09 ±0.06**	**81.91**
Claude 3.5 Sonnet	-	38.70 ±1.06 (50%)	1.78 ±2.27	77.05
Gemini 1.5 Pro	-	40.62 ±0.80 (46%)	1.66 ±0.03	74.86
ChatGPT	-	32.52 ±1.40 (51%)	1.18 ±0.97	66.94
GPT-4o	-	39.36 ±0.85 (49%)	2.55 ±4.21	76.78

improvement of MagicCoder, StarCoder, and CodeLlama over other LLMs in terms of efficient@1. This suggests that the 33 similar instances have a very limited impact on the efficiency evaluation of COFFE and do not introduce bias in the paper's findings. By comparing the time efficiency of code solutions with ground-truth solutions to calculate efficient@1, COFFE can prevent the bias introduced by the potential data leakage from ground-truth solutions in code efficiency evaluation.

B. Solution-level Analysis

In solution-level analysis, we compare the solutions generated by LLMs and the solutions in the training sets. Specifically, for each generated solution that passes the test cases, we first compute its AST to the Python code extracted from the four training sets. A generated solution is considered verbatim only when its full AST, including node types, identifiers, and literal values, is identical to a training solution after stripping comments and docstrings. For the function split, we canonicalize function names and self-references because the benchmark harness renames functions to "*solution*". For the file split, we compare whole-module ASTs directly.

For the identified verbatim solutions generated by LLMs, we then compare the CPU instruction counts consumed by

TABLE XIV
SOLUTION-LEVEL LEAKAGE ANALYSIS RESULTS ON THE FUNCTION SPLIT OF COFFE. MODELS HIGHLIGHTED IN GRAY ARE CLOSED-SOURCE MODELS. EACH LEAKAGE COLUMN REPORTS THE NUMBER AND RATIO IN THE FORMAT "COUNT (RATIO)". "VERBATIM" INDICATES FULL-AST IDENTICAL GENERATED SOLUTIONS, "VERB. GT" DENOTES VERBATIM SOLUTIONS EQUIVALENT TO THE GROUND TRUTH, AND "VERB. FASTER" DENOTES VERBATIM SOLUTIONS FASTER THAN THE BEST GROUND-TRUTH SOLUTION.

Model	Size	Passed	Verbatim	Verb. GT	Verb. Faster
Phi3	3.8B	173	16 (4.02%)	3 (0.75%)	12 (3.02%)
MagicCoder	DS-6.7B	129	14 (3.52%)	3 (0.75%)	8 (2.01%)
	CL-7B	185	16 (4.02%)	1 (0.25%)	10 (2.51%)
CodeLlama	7B	154	11 (2.76%)	0 (0.00%)	7 (1.76%)
	13B	166	16 (4.02%)	3 (0.75%)	10 (2.51%)
	34B	257	28 (7.04%)	4 (1.01%)	17 (4.27%)
Llama3	8B	169	20 (5.03%)	3 (0.75%)	14 (3.52%)
	70B	269	26 (6.53%)	3 (0.75%)	16 (4.02%)
StarCoder	15B	244	31 (7.79%)	7 (1.76%)	19 (4.77%)
WizardCoder	15B	193	13 (3.27%)	2 (0.50%)	8 (2.01%)
Mixtral	8×7B	178	18 (4.52%)	0 (0.00%)	15 (3.77%)
DeepSeek V2	236B	312	26 (6.53%)	4 (1.01%)	17 (4.27%)
DeepSeek V2 Coder	236B	318	26 (6.53%)	4 (1.01%)	17 (4.27%)
Llama3.1	405B	268	20 (5.03%)	5 (1.26%)	11 (2.76%)
Claude 3.5 Sonnet	-	309	15 (3.77%)	3 (0.75%)	7 (1.76%)
Gemini 1.5 Pro	-	300	22 (5.53%)	4 (1.01%)	12 (3.02%)
ChatGPT	-	268	23 (5.78%)	3 (0.75%)	15 (3.77%)
GPT-4o	-	309	25 (6.28%)	3 (0.75%)	17 (4.27%)
Total		**4201**	**366 (5.11%)**	**55 (0.77%)**	**232 (3.24%)**

these solutions and the ground-truth solutions in COFFE. We show the results in Table XIV. Note that we do not identify any verbatim solutions generated by LLMs at the file split of COFFE, so we only show the results of the function split.

From the table, we observe that only 232 LLM-generated code solutions are both verbatim and faster than the ground truth, i.e., 3.24%. As the efficient@1 metric defined in this paper cannot be improved by generating the same ground truth solutions, the 232 solutions directly measure the maximum extent to which exact solution reproduction from the four training sets could affect efficient@1. While a 3.24% leakage does not impact the main findings concluded in this paper. Therefore, we claim that the solution-level leakage does not introduce significant bias in the paper's findings.

VIII. THREATS TO VALIDITY

Our research may face the following threats to the internal and external validity.

A. Threats to Internal Validity

Performance Measurement. The time efficiency measurement of code solutions generated by LLMs can introduce errors. We propose to use CPU instruction count instead of execution time to improve the stability of measurements. However, there still exist factors such as specific code optimization techniques that introduce measurement errors. To mitigate the threats posed by the errors in time efficiency measurements, we conduct all measurements in dockers [23] to ensure that only one single process is running at the same time. Furthermore, we run the measurements for each code solution 12 times and remove the highest and lowest measurements before calculating the average metric. This could further reduce the errors introduced in a single measurement.

Representativeness of CPU Instruction Count. We introduce CPU instruction counts as a more stable metric for evaluating code efficiency. Sometimes different CPU instructions may lead to different execution times, so CPU instruction count may not be completely linearly correlated with execution time. To mitigate this risk, we use only CPU instruction counts to compare the efficiency of different solutions to the same problem, since they are likely to share the same instruction distribution.

Baseline Implementation. Currently, there are no LLM-based stressful test case generation methods that could be compared with STGEN, so we modify three correctness test case generation methods as our baselines. However, such modifications may result in performance changes. To improve the validity of baselines, we run them on the most powerful and robust LLM GPT-4o [81]. Besides, we ask the baselines to generate 20 stressful test cases once and only choose the best 5 test cases for most evaluations except for accuracy. Therefore, we believe our implementations can represent the best performance of baselines.

B. Threats to External Validity

Adaptation to Different Programming Languages. While code generation is a general task for all programming languages, we mainly focus on the evaluation of Python code generation in this paper. The code generation performance of LLMs on other programming languages such as C++ and Java may be different from the experiment results we show in Sec. V, as it could be affected by the syntax and coding styles. This threatens the validity of our experiment results in other programming languages. However, Python is the top 2 most popular programming language at GitHub [36] and is the major programming language used to build the code generation benchmarks [5], [9], [41], [42], [49], [60], [66], [117]. Besides, our stressful test case generation method STGEN is language-agnostic and fully based on LLMs to generate stressful test cases, we believe it could be easily extended to build benchmarks for other programming languages.

IX. RELATED WORK

A. LLMs for Code Generation

As a critical task to automate the software development process, code generation has drawn a lot of attention in both the academia and industry. At the beginning, encoder-decoder models such as AlphaCode [60], CodeT5 [107], CodeRL [56], CodeT5+ [106] are directly trained on large code corpus and obtain good performance on code generation. Recently, decoder-only models such as Codex [10], CodeGen [77], [78], InCoder [28], CodeGeeX [119], SantaCoder [2], StarCoder [58], [67], WizardCoder [68], CodeLlama [95], MagicCoder [109], DeepSeek-Coder [40] show superior performance than encoder-decoder models on code generation. Besides,

some general LLMs trained on multiple types of data, such as Llama3 [70], Llama3.1 [71], GPT-3.5 [79], GPT-4 [80] also demonstrate competitive or even better performance compared with code LLMs.

B. Code Generation Benchmarks

Correctness Benchmarks. There are many benchmarks designed for the correctness evaluation of code generated by LLMs. They provide contexts that indicate the functionality of the generated code and several test cases to evaluate the correctness of the generated code. The benchmarks are initially built from scratch by skilled developers and researchers. HumanEval [9] is a benchmark that contains 164 Python programming problems with function signatures and docstrings. MBPP [5] is a benchmark consisting of 974 basic Python programming problems with short functionality descriptions. It also provides a sanitized version with verified ground truth solutions that have 427 problems. In order to comprehensively evaluate the performance of LLMs, some benchmarks are built from code competition problems. APPS [42] contains 10,000 Python problems with different difficulty levels and diversified ground truth solutions for each problem. Code Contests [60] is a multi-lingual benchmark built from various competition sources and includes both correct and incorrect human solutions for each problem. Apart from Code Contests, there are also other multi-lingual benchmarks such as xCodeEval [55] and HumanEval-X [119]. The above-mentioned benchmarks focus on the evaluation of function-level or file-level code generation. There are some research efforts, such as RepoEval [117], RepoBench [66], SWE-Bench [53], and CrossCodeEval [22], devoted to the evaluation of the repo-level code generation performance.

Time Efficiency Benchmarks. Despite the well-explored evaluation for the correctness of code generated by LLMs, the time efficiency of code generated by LLMs is under-explored. Effibench [49] is the first benchmark designed for evaluating the time and memory efficiency of code generation. It selects efficiency-critical problems tagged "LeetCode" and prompts GPT-3.5 to generate test cases with different input sizes and data distribution. However, the problems in this benchmark are too difficult, so most open-source models cannot even generate correct solutions. Besides, it adopts execution time as the performance metric, which is unreliable to distinguish the efficiency of different code solutions.

C. LLM-based Test Case Generation

Apart from the advances in code generation, LLMs have also been demonstrated to improve software testing [21]. A lot of work has comprehensively evaluated the ability of LLMs on test case generation [54], [57], [74], [86], [96]. Most recently, Chen *et al.* [12] propose ChatUniTest, a unit test generation framework based on LLM by utilizing innovative mechanisms such as adaptive focal context and generation-validation-repair mechanisms. Liu *et al.* [64] propose a novel LLM-powered test oracle generation approach that combines LLMs and differential testing. Hossain *et al.* [45] propose TOGLL, a fine-tuned LLM on designed instruction prompts to generate test oracle for Java projects. Wang *et al.* [103] propose TestEval to generate test cases that cover certain lines, branches, and paths of the code under test. Despite the effectiveness of previous approaches on correctness test case generation, there is no work on stressful test case generation that aims to generate large test inputs to evaluate the time efficiency of the code under test. In this paper, we propose a novel approach STGEN to generate stressful test cases for Python projects with high accuracy and coverage.

D. LLM-based Code Optimization

Code Optimization aims to improve the time efficiency of existing code. It is an important task to produce high-quality code in practice and is initially integrated into compilers such as gcc [27]. In the era of LLMs, researchers have started to use LLMs to optimize the code. Gong *et al.* [37] provide a survey that collects all recent LLM-based approaches on code optimization. Specifically, some researchers focus on prompt engineering techniques. For example, Self-Refine [69] proposes to refine the code based on self-feedback iteratively, while EffiLearner [47], ECCO [102], PerfCodeGen [89] propose to refine the code based on outside feedback from profilers and performance measurements. Gao *et al.* [33] propose to use search-based approaches to select appropriate examples and optimization patterns from existing practice in prompts to optimize the code. Garg *et al.* [34] propose to use RAG techniques when constructing the prompts for LLMs to fix inefficient code. Other researchers focus on model training techniques to directly let LLMs generate efficient code. Shypula *et al.* [99] and Ye *et al.* [114] build a dataset containing slow-fast code pairs to fine-tune the models. SwiftCoder [48] uses code solutions from different LLMs to generate the dataset for fine-tuning the models. Duan *et al.* [24], Gee *et al.* [35], and Nichols *et al.* [76] propose to use reinforcement learning to train the models. Ren *et al.* [93] propose an code editing approach for repo-level code optimization.

X. Conclusion

In this paper, we propose a new benchmark COFFE for the time efficiency evaluation of LLM-generated code. To address the challenges of existing correctness code generation benchmarks, we propose a novel stressful test case generation method STGEN that incorporates contracts and two test case formats to improve the accuracy. To comprehensively investigate the robustness of LLM-generated code and current code optimization approaches, we complement STGEN with STMUT to generate test cases with different input scales. For the evaluation, we introduce a new time efficiency metric *efficient@k* based on CPU instruction count that stably evaluates both the correctness and time efficiency of code. Based on COFFE, we evaluate 19 popular LLMs and 7 LLM-based code optimization approaches and identify 11 important findings. We provide implications based on the findings for LLM researchers and software practitioners.

REFERENCES

[1] Marah I Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, et al. Phi-3 technical report: A highly capable language model locally on your phone. *CoRR*, abs/2404.14219, 2024.

[2] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, et al. Santacoder: don't reach for the stars! *CoRR*, abs/2301.03988, 2023.

[3] Anthropic. Api reference provided by anthropic, 2024. https://docs.anthropic.com/en/api/getting-started.

[4] Anthropic. Claude 3.5 sonnet, 2024. https://www.anthropic.com/news/claude-3-5-sonnet.

[5] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021.

[6] Ned Batchelder. The coverage.py library, 2024. https://github.com/nedbat/coveragepy.

[7] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. Unit test generation using generative AI : A comparative performance analysis of autogeneration tools. *CoRR*, abs/2312.10622, 2023.

[8] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[11] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *CoRR*, abs/2304.05128, 2023.

[12] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. Chatunitest: A framework for llm-based test generation. In Marcelo d'Amorim, editor, *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, pages 572–576. ACM, 2024.

[13] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, et al. Scaling instruction-finetuned language models. *CoRR*, abs/2210.11416, 2022.

[14] The MITRE Corporation. Performance efficiency cwes, 2024. https://cwe.mitre.org/data/definitions/1132.html.

[15] Deepmind. Gemini 1.5 pro, 2024. https://deepmind.google/technologies/gemini/pro/.

[16] DeepSeek. Deepseek api, 2024. https://platform.deepseek.com/.

[17] DeepSeek-AI et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *CoRR*, abs/2405.04434, 2024.

[18] DeepSeek-AI et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[19] DeepSeek-AI et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[20] DeepSeek-AI et al. Deepseek-v3 technical report, 2025.

[21] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 423–435. ACM, 2023.

[22] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[23] Docker. Docker, 2024. https://www.docker.com/.

[24] Shukai Duan, Nikos Kanakaris, Xiongye Xiao, Heng Ping, Chenyu Zhou, Nesreen K. Ahmed, Guixiang Ma, Mihai Capota, Theodore L. Willke, Shahin Nazarian, and Paul Bogdan. Leveraging reinforcement learning and large language models for code optimization. *CoRR*, abs/2312.05657, 2023.

[25] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering*, 1(FSE):1889–1912, 2024.

[26] The Linux Foundation. The perf tool on linux, 2024. https://perf.wiki.kernel.org/index.php/Main_Page.

[27] Inc. Free Software Foundation. Gcc compiler, 2025. https://gcc.gnu.org/.

[28] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[29] Downloaded from HuggingFace in Apr 2026. The evol-instruction-66k dataset, 2026. https://huggingface.co/datasets/codefuse-ai/Evol-instruction-66k.

[30] Downloaded from HuggingFace in Apr 2026. The magicoder-evol-instruct-110k dataset, 2026. https://huggingface.co/datasets/ise-uiuc/Magicoder-Evol-Instruct-110K.

[31] Downloaded from HuggingFace in Apr 2026. The magicoder-oss-instruct-75k dataset, 2026. https://huggingface.co/datasets/ise-uiuc/Magicoder-OSS-Instruct-75K.

[32] Downloaded from HuggingFace in Apr 2026. The self-oss-instruct-sc2-exec-filter-50k dataset, 2026. https://huggingface.co/datasets/bigcode/self-oss-instruct-sc2-exec-filter-50k.

[33] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. Search-based llms for code optimization, 2024.

[34] Spandan Garg, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. Rapgen: An approach for fixing code inefficiencies in zero-shot, 2025.

[35] Leonidas Gee, Milan Gritta, Gerasimos Lampouras, and Ignacio Iacobacci. Code-optimise: Self-generated preference data for correctness and efficiency. *CoRR*, abs/2406.12502, 2024.

[36] Inc. GitHub. Github octoverse report on programming languages, 2022. https://octoverse.github.com/2022/top-programming-languages.

[37] Jingzhi Gong, Vardan Voskanyan, Paul Brookes, Fan Wu, Wei Jie, Jie Xu, Rafail Giavrimis, Mike Basios, Leslie Kanthan, and Zheng Wang. Language models for code optimization: Survey, challenges and future directions. *CoRR*, abs/2501.01277, 2025.

[38] Google. Sanitized version of mbpp benchmark released by google, 2023. https://huggingface.co/datasets/google-research-datasets/mbpp/viewer/sanitized/test.

[39] Google. Api reference provided by google, 2024. https://ai.google.dev/gemini-api/docs/models/gemini.

[40] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196, 2024.

[41] Nam Le Hai, Dung Manh Nguyen, and Nghi D. Q. Bui. REPOEXEC: evaluate code generation with a repository-level executable benchmark. *CoRR*, abs/2406.11927, 2024.

[42] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In Joaquin Vanschoren and Sai-Kit Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021.

[43] Samuel Holt, Max Ruiz Luyten, and Mihaela van der Schaar. L2MAC: large language model automatic computer for unbounded code generation. *CoRR*, abs/2310.02003, 2023.

[44] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, et al. Metagpt: Meta programming for A multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[45] Soneya Binta Hossain and Matthew B. Dwyer. TOGLL: correct and strong test oracle generation with llms. *CoRR*, abs/2405.03786, 2024.

[46] Dong Huang, Qingwen Bu, Jie M. Zhang, Michael Luck, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *CoRR*, abs/2312.13010, 2023.

[47] Dong Huang, Jianbo Dai, Han Weng, Puzhen Wu, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M. Zhang. Effilearner: Enhancing efficiency of generated code via self-optimization, 2024.

[48] Dong Huang, Guangtao Zeng, Jianbo Dai, Meng Luo, Han Weng, Yuhao Qing, Heming Cui, Zhijiang Guo, and Jie M. Zhang. Swiftcoder: Enhancing code generation in large language models through efficiency-aware fine-tuning, 2025.

[49] Dong Huang, Jie M. Zhang, Yuhao Qing, and Heming Cui. Effibench: Benchmarking the efficiency of automatically generated code. *CoRR*, abs/2402.02037, 2024.

[50] Deep Infra. Deep infra api, 2024. https://deepinfra.com/.

[51] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *CoRR*, abs/2401.04088, 2024.

[52] Shuyang Jiang, Yuhao Wang, and Yu Wang. Selfevolve: A code evolution framework via large language models. *CoRR*, abs/2306.02907, 2023.

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

[54] Rabimba Karanjai, Aftab Hussain, Md Rafiqul Islam Rabin, Lei Xu, Weidong Shi, and Mohammad Amin Alipour. Harnessing the power of llms: Automating unit test generation for high-performance computing, 2024.

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

[56] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[57] Kefan Li and Yuan Yuan. Large language models as test case generators: Performance evaluation and enhancement. *CoRR*, abs/2404.13340, 2024.

[58] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *CoRR*, abs/2305.06161, 2023.

[59] Yichen Li, Yun Peng, Yintong Huo, and Michael R. Lyu. Enhancing llm-based coding tools through native integration of ide-derived static context, 2024.

[60] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[61] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[62] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. The mbpp plus benchmark, 2023. https://github.com/evalplus/evalplus/releases/tag/v0.2.1.

[63] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Evalplus leaderboard, 2024. https://evalplus.github.io/leaderboard.html.

[64] Kaibo Liu, Yiyang Liu, Zhenpeng Chen, Jie M. Zhang, Yudong Han, Yun Ma, Ge Li, and Gang Huang. Llm-powered test case generation for detecting tricky bugs. *CoRR*, abs/2404.10304, 2024.

[65] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. Retrieval-augmented generation for code summarization via hybrid GNN. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[66] Tianyang Liu, Canwen Xu, and Julian J. McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *CoRR*, abs/2306.03091, 2023.

[67] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173, 2024.

[68] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *CoRR*, abs/2306.08568, 2023.

[69] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, et al. Self-refine: Iterative refinement with self-feedback, 2023.

[70] Meta. Llama3, 2024. https://ai.meta.com/blog/meta-llama-3/.

[71] Meta. Llama3.1, 2024. https://ai.meta.com/blog/meta-llama-3-1/.

[72] Mistral. Codestral blog, 2024. https://mistral.ai/news/codestral.

[73] Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[74] Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin T. Vechev. Code agents are state of the art software testers. *CoRR*, abs/2406.12952, 2024.

[75] Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida I. Wang, and Xi Victoria Lin. LEVER: learning to verify language-to-code generation with execution. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 26106–26128. PMLR, 2023.

[76] Daniel Nichols, Pranav Polasam, Harshitha Menon, Aniruddha Marathe, Todd Gamblin, and Abhinav Bhatele. Performance-aligned llms for generating fast code. *CoRR*, abs/2404.18864, 2024.

[77] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. Codegen2: Lessons for training llms on programming and natural languages. *CoRR*, abs/2305.02309, 2023.

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

[79] OpenAI. Chatgpt, 2022. https://openai.com/blog/chatgpt.

[80] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[81] OpenAI. Gpt-4o, 2024. https://openai.com/index/hello-gpt-4o/.

[82] OpenAI. Openai api, 2024. https://openai.com/api/.

[83] Openrouter. Openrouter, 2026. https://openrouter.ai/.

[84] The opensource community. Line profiler, 2025. https://github.com/pyutils/line_profiler.

[85] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, et al. Training language models to follow instructions with human feedback. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[86] Wendkûuni C. Ouédraogo, Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. Large-scale, independent and comprehensive study of the power of llms for test case generation, 2024.

[87] Md. Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Retrieval augmented code generation and summarization. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Findings of the Association for Computational Linguistics: EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 16-20 November, 2021*, pages 2719–2734. Association for Computational Linguistics, 2021.

[88] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012.

[89] Yun Peng, Akhilesh Deepak Gotmare, Michael Lyu, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. Perfcodegen: Improving performance of llm generated code with execution feedback, 2024.

[90] Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. Coffe: A code efficiency benchmark for code generation, 2025.

[91] Qwen. Qwen2.5 technical report, 2025.

[92] Abhinav Rastogi, Adam Yang, Albert Q. Jiang, Alexander H. Liu, Alexandre Sablayrolles, et al. Devstral: Fine-tuning language models for coding agent applications, 2025.

[93] Xiaoxue Ren, Jun Wan, Yun Peng, Zhongxin Liu, Ming Liang, Dajun Chen, Wei Jiang, and Yong Li. Peace: Towards efficient project-level efficiency optimization via hybrid code editing. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 1831–1843. IEEE Press, 2025.

[94] Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *CoRR*, abs/2401.08500, 2024.

[95] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023.

[96] Malik Abdul Sami, Zeeshan Rasheed, Muhammad Waseem, Zheying Zhang, Tomas Herda, and Pekka Abrahamsson. A tool for test case scenarios generation using large language models. *CoRR*, abs/2406.07021, 2024.

[97] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2024.

[98] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[99] Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations*, 2024.

[100] Matt Stuchlik, Bruno P. Kinoshita, and Donald Lee. The cirron library, 2024. https://github.com/s7nfo/Cirron.

[101] Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. ARKS: active retrieval in knowledge soup for code generation. *CoRR*, abs/2402.12317, 2024.

[102] Siddhant Waghjale, Vishruth Veerendranath, Zhiruo Wang, and Daniel Fried. ECCO: can we improve model-generated code efficiency without sacrificing functional correctness? In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pages 15362–15376. Association for Computational Linguistics, 2024.

[103] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. TESTEVAL: benchmarking large language models for test case generation. *CoRR*, abs/2406.04531, 2024.

[104] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. *CoRR*, abs/2402.01030, 2024.

[105] Yejie Wang, Keqing He, Dayuan Fu, Zhuoma Gongque, Heyang Xu, Yanxu Chen, Zhexu Wang, Yujia Fu, Guanting Dong, Muxi Diao, Jingang Wang, Mengdi Zhang, Xunliang Cai, and Weiran Xu. How do your code llms perform? empowering code instruction tuning with high-quality data, 2024.

[106] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 1069–1088. Association for Computational Linguistics, 2023.

[107] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 8696–8708. Association for Computational Linguistics, 2021.

[108] 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. Emergent abilities of large language models. *Trans. Mach. Learn. Res.*, 2022, 2022.

[109] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Source code is all you need. *CoRR*, abs/2312.02120, 2023.

[110] Papers with Code. The leaderboard of apps benchmark on papers with code, 2024. https://paperswithcode.com/sota/code-generation-on-apps.

[111] Papers with Code. The leaderboard of the code contests benchmark, 2024. https://paperswithcode.com/sota/code-generation-on-codecontests.

[112] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. Wizardlm: Empowering large language models to follow complex instructions. *CoRR*, abs/2304.12244, 2023.

[113] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *CoRR*, abs/2405.15793, 2024.

[114] Tong Ye, Tengfei Ma, Lingfei Wu, Xuhong Zhang, Shouling Ji, and Wenhai Wang. Iterative or innovative? A problem-oriented perspective for code optimization. *CoRR*, abs/2406.11935, 2024.

[115] Xiao Yu, Lei Liu, Xing Hu, Jacky Wai Keung, Jin Liu, and Xin Xia. Where are large language models for code generation on github?, 2024.

[116] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 2471–2484. Association for Computational Linguistics, 2023.

[117] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 2471–2484. Association for Computational Linguistics, 2023.

[118] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. *CoRR*, abs/2404.05427, 2024.

[119] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *CoRR*, abs/2303.17568, 2023.

[120] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. *CoRR*, abs/2310.04406, 2023.

[121] Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[122] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.