Rethinking the Influence of Source Code on Test Case Generation

DONG HUANG, University of Hong Kong
JIE M. ZHANG, King's College London
MINGZHE DU, National University of Singapore
MARK HARMAN, University College London
HEMING CUI, University of Hong Kong

Large language models (LLMs) have been widely applied to assist test generation with the source code under test provided as the context. This paper aims to answer the question: **If the source code under test is incorrect, will LLMs be misguided when generating tests?** The effectiveness of test cases is measured by their accuracy, coverage, and bug detection effectiveness. Our evaluation results with **five** open- and **six** closed-source LLMs on four datasets demonstrate that incorrect code can significantly mislead LLMs in generating correct, high-coverage, and bug-revealing tests. For instance, in the HumanEval dataset, LLMs achieve 80.45% test accuracy when provided with task descriptions and correct code, but only 57.12% when given task descriptions and incorrect code. For the APPS dataset, prompts with correct code yield tests that detect 39.85% of the bugs, while prompts with incorrect code detect only 19.61%. These findings have important implications for the deployment of LLM-based testing — **using it on mature code may help protect against future regression, but on early-stage immature code, it may simply bake in errors**. Our findings also underscore the need for further research to improve LLMs' resilience against incorrect code in generating reliable and bug-revealing tests.

ACM Reference Format:
Dong Huang, Jie M. Zhang, Mingzhe Du, Mark Harman, and Heming Cui. 2024. Rethinking the Influence of Source Code on Test Case Generation. 1, 1 (September 2024), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Automatic test case generation is a crucial part of the software development process, enriching the effectiveness of test cases and ensuring that the software under development adheres to the specified requirements and operates as intended [4, 61]. Recently, many research works have harnessed the capabilities of large language models (LLMs) to generate test cases automatically[9, 14, 16, 24, 25, 27, 49, 55, 65, 66, 69, 72]. The information provided with LLMs typically contains two aspects: the source code under test, and/or the task description of the code. For example, FuzzGPT [16], TitanFuzz [14], KernelGPT [67], and CodaMOSA [34] provide LLMs with the source code under test only for LLMs to generate tests automatically. CodeCoT [25] uses both task description and source code under test. AgentCoder [27] and MetaGPT [24] directly provide the task description to LLMs without source code.

Authors' addresses: Dong Huang, dhuang@cs.hku.hk, University of Hong Kong; Jie M. Zhang, jie.zhang@kcl.ac.uk, King's College London; Mingzhe Du, mingzhe@nus.edu.sg, National University of Singapore; Mark Harman, mark.harman@ucl.ac.uk, University College London; Heming Cui, heming@cs.hku.hk, University of Hong Kong.

XXXX-XXXX/2024/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Although generating tests with LLMs based on the source code under test is a common practice, it poses a significant challenge that is often overlooked: if the source code under test contains bugs, the tests generated by LLMs may inherit the flawed logic or assumptions from the code, resulting in ineffective or incorrect tests. The relationship between the correctness of the source code and the effectiveness of the generated test cases, however, remains largely unexplored.

To fill this gap, in this paper, we present the first systematic empirical study on how the correctness of the code under test impacts the effectiveness of the LLM-generated test cases. We evaluate the effectiveness of test cases by measuring their accuracy[1] and coverage in the correct code provided by the dataset. We also check their bug detection ratio in our collected bug set.

We first conduct experiments using 5 open-source and 6 closed-source LLMs on three widely-studied code generation datasets (i.e., HumanEval [48], MBPP [6], and APPS [22]). For each code generation task, we prompt each LLM to generate test cases based on five different prompts: (1) task description only, (2) task description with correct code, (3) task description with incorrect code, (4) correct code only, and (5) incorrect code only. We then evaluate the effectiveness of LLM-generated test cases in three dimensions: accuracy, coverage, and bug detection ratio. We also examine whether LLMs are more prone to being misled by the code they generate themselves. Finally, we evaluate LLMs with incorrect code from the real-world library BugsinPy [64] to check whether our observations hold for real-world scenarios.

Our results demonstrate that incorrect code under test can significantly impact the ability of LLMs to generate effective tests. For example, for the HumanEval dataset, test cases generated by LLMs achieve an accuracy of 80.45%, a coverage of 98.43%, and a bug detection ratio of 87.38% with both task descriptions and correct code under test in the prompt. However, when the code under test is incorrect, these results drop to 57.12%, 91.72%, and 74.97%, respectively. We also observe that LLMs are less likely to be misguided by the code they generate by themselves. Finally, our experiments with real-world tasks demonstrate the same conclusions as those of widely adopted benchmarks, although the accuracy, coverage, and bug detection ratio are much lower than on the three simpler benchmarks. In particular, for the bug detection ratio, LLMs with correct code under test detect 5.45% of the bugs on average, but LLMs with incorrect code under test detect only 0.91% on average.

In conclusion, this paper makes the following contributions:

- We present the first systematic study on the influence of source code on test case generation.
- Our evaluation results demonstrate that providing task descriptions with correct code yields SOTA performance in test case generation. For instance, in the HumanEval dataset, LLM-generated test cases achieve an average accuracy of 80.45% on average for all models when providing task descriptions and correct code. Conversely, when provided with task descriptions and incorrect code, the average accuracy declines substantially to 57.12%.
- We provide implications for developers and researchers on using LLMs for generating tests automatically based on our observations. In particular, our finding indicates that **LLM-based testing will be more effective at generating tests to protect mature code from regression errors. However, if used in the early stage of software development on relatively immature code, it will be more likely to "bake in" errors.** We also call for more research to improve LLMs' resilience against incorrect code in generating reliable and bug-revealing tests.

[1]Both "accuracy" and "correctness" are widely used in the literature to refer to the ratio of the test cases that pass correct code against the total number of generated test cases [9, 30, 36, 38, 40, 61, 69]. We use the term of accuracy in our paper.

2 BACKGROUND AND RELATED WORK

2.1 LLMs for Source Code Generation

LLMs have seen boosting adoption in code generation, driven by the availability of extensive open-source code repositories and the demand for enhanced developer productivity. Pioneering works focused on generating functionally correct code from natural language instructions exclusively, including CodeT5 [62], AlphaCode [39], CodeGen [47], InCoder [18], StarCoder [37], SantaCoder [3], and DeepSeek Coder [13]. With the rapid scale expansion of LLMs, subsequent advancements have produced models such as Codex [10] and CodeLLaMA [53]. These models are fine-tuned from foundational LLMs [8, 58] and are proficient in a variety of tasks, including code generation [10, 12], program repair [20, 28], automated testing [15, 34], code translation [1, 54], type prediction [46, 63], and code summarization [2, 21]. Among these, the model performance on the code generation task has emerged as a pivotal benchmark for evaluating the LLM holistic coding capability.

To enhance the functional correctness of generated source code, feedback-based refinement techniques have been employed. These methods mimic the human learning process, where individuals enhance their knowledge through trial and error [7, 45]. Initial efforts revolved around human feedback for model evaluation and refinement [31, 51]. To reduce human intervention, automated feedback approaches have been explored, utilizing signals from the diverse aspects, including LLM self-reflection [26, 43], dedicated verification models [42], external tools [25, 27], external knowledge sources [19], and model generation distribution [68]. For example, Self-Edit [70] and Self-Evolve [29] execute the initially generated program on canonical test cases and provide the execution results as feedback to prompt the LLM to refine the code. Furthermore, Self-Debug [11] incorporates multiple feedback sources, including program explanations, unit tests, and program interpreters. Notably, ALGO [71] takes a more detailed approach to generate a reference oracle program via an exhaustive search.

2.2 Debugging and Improving Source Code with Test Cases

In the current code evaluation paradigm [24, 25, 57, 60], an LLM starts by tentatively generating a source code based on the given task description and then validating the code functionality through a set of pre-defined test cases. These test cases are executed and expected to identify any code errors and inconsistencies between the generated code and the given task description. Consequently, developing appropriate test cases is vital for accurately assessing code generation tasks. However, high-effectiveness public test cases are not always available. To address this, researchers have harnessed LLMs to generate test cases [9, 24, 25, 27, 49, 55, 72]. Tools like CodeT [9] generate test cases directly for the source code, minimizing human effort and expanding test scenario coverage. CodeChain [33] enhances this by devising prompt templates to format the generated test cases. CodeCoT [25] advances further by generating both source code and test cases simultaneously. AgentCoder [27] and MetaGPT [24] decompose the software development process into multiple stages, with each stage managed by specialized agents. Test designer agents, for example, are proficient in generating reliable test cases based on the task description.

2.3 Improving Effectiveness of Test Case Generation

Low-effectiveness test cases can mislead the debugging process, resulting in incorrect conclusions and sub-optimal code refinement [9, 24, 27]. One potential issue arises when the generated test cases are misaligned with the problem instruction. In the code debugging process, even if the generated code is correct, it may fail to pass erroneous tests, leading the LLM to unnecessarily rectify the code and potentially introduce new errors. Similarly, in software testing, the developed software may raise errors when incorrect test cases are used to analyze its correctness. The errors raised by incorrect

code may also cause developers to revise the source code and introduce new errors. Another concern is the coverage of the generated test cases [9, 25]. If the test cases only cover a limited range of common behaviors and fail to account for edge cases or specific task requirements, the generated code may pass all tests while still being incomplete or incorrect. This can give a false sense of confidence in the code's correctness, as it has not been thoroughly validated against all relevant scenarios. To enhance test case effectiveness, several prompt engineering techniques are employed, which involve using source code-guided and non-source code-guided approaches. Frameworks like CodeT [9], AgentCoder [27], MetaGPT [24], LATS [73], and Reflexion [56] generate test cases based solely on task descriptions. In contrast, CodeCoT [25], ATHENATEST [59], EvalPlus [41], and CodaMOSA [34] leverage existing source code to generate test cases. Though these methods show promise, the impact of incorporating source code on test case effectiveness is not comprehensively understood. This paper aims to empirically study whether source code inclusion consistently enhances the effectiveness of LLM-generated test cases, compared to using task descriptions alone.

3 METHOD

This section introduces our method for generating, extracting, and executing tests, as well as our measurements on the effectiveness of tests.

3.1 Prompt Construction

The first step in our study is prompt construction. In our experiments, we have five prompts for each task that requires LLMs to generate code (See Tab. 1). The first prompt (`P_T`) is the **T**ask description. For this prompt, we follow the setup of existing works [10, 50], and directly ask LLMs to generate test cases for each task based on the task description with zero-shot prompting. The second prompt (`P_T_CC`) in our experiments is **T**ask description + **C**orrect code. For the HumanEval, MBPP, and APPS datasets, we directly use the correct code provided by each dataset to represent the correct code in our experiments. For BugsInPy, we utilize the patched code as the correct code in our experiments. The third prompt (`P_T_IC`) in our experiments is the **T**ask description + **I**ncorrect code. For the incorrect code, we first require LLMs evaluated in our experiments to generate code with zero-shot prompting for the HumanEval, MBPP, and APPS datasets, and then collect incorrect pieces of code for each task[2] in our evaluated dataset and then randomly select an incorrect code that would be used in all models as the `P_T_IC`'s incorrect code part. For the BugsInPy dataset, we directly use the pre-patch source code as the incorrect code. For the fourth prompt (`P_CC`), we utilize **C**orrect code without task description. The fifth prompt (`P_IC`) is directly used as an **I**ncorrect solution without a task description. In our experiments, the correct source code for `P_T_CC` and `P_CC` is the same for each task, and the incorrect source code for `P_T_IC` and `P_IC` is also the same for each task.

Table 1. The five prompts used in our empirical study for generating test cases with LLMs.

Prompt	Template
`P_T`	**T**ask description
`P_T_CC`	**T**ask description + **C**orrect **C**ode
`P_T_IC`	**T**ask description + **I**ncorrect **C**ode
`P_CC`	**C**orrect **C**ode
`P_IC`	**I**ncorrect **C**ode

Finally, to make sure the test cases generated by LLMs follow the test case format rather than pure natural languages in the experiments, we also provide the test case template `assert function_name(input_parameters) == output` before the task description so that the test cases generated by LLMs can follow the same format and can be directly used in our experiments.

[2]Since some tasks would be addressed by all LLMs, we then filter these tasks from our evaluation tasks.

3.2 Tests Extraction and Script Writing

To make sure the test cases can be extracted from the LLMs' response, we constraint LLMs generate test cases in the ``` python[test_case] ``` so that we can directly extract test cases from ```python and ``` that can remove the natural language in the test cases[3]. After extracting tests from the LLM-generated response, we utilize the HumanEval provided script to automatically write the source code (e.g., correct code for the accuracy and coverage evaluation) and LLM-generated tests in the script. For the required libraries for each task, we directly import them based on the datasets (e.g., HumanEval) setup, which can avoid errors raised due to the script's lack of necessary libraries in the experiments.

3.3 Source Code Execution

For accuracy and coverage, we conduct experiments in each dataset-provided correct code. For bug detection experiments, we execute LLM-generated test cases in the constructed bug detection source code. During the code execution process, we set the timeout value as 5 seconds for all tasks to make sure the code can be executed with all test cases and does not require much time. To speed up the testing process, we utilize concurrency in our accuracy and bug detection experiments and set the maximum number of workers as 20, which can reduce the overhead of the testing process. Since we employ `coverage.py` library [4] for the coverage experiments, which can not support the concurrency setting, we opt to execute all tests using a single-threaded script instead.

3.4 Effectiveness Measurement

We measure the effectiveness of LLM-generated test cases from three metrics, i.e., the accuracy of LLM-generated test cases (Accuracy), code line coverage of LLM-generated test cases in the correct code (Coverage), and bug detection effectiveness of LLM-generated test cases (Bug Detection).

3.4.1 Accuracy. We measure the accuracy of LLM-generated test cases by calculating the number of test cases generated by LLM that can pass the correct code provided by the dataset[5]. If a test case generated by an LLM passes the correct code, we treat it as correct, i.e., when we feed the input of the test case into the correct code, the correct code returns the same output as the test case output. We analyze two levels of effectiveness in our experiments: test level and task level.

At the **test level**, we analyze the accuracy of LLM-generated test cases for the same task individually. For example, if GPT-3.5-turbo generated test cases for Task 1 provided by HumanEval have ten test cases, where seven of the test cases are correct and three test cases are incorrect, we then calculate the test level accuracy as 70% (7/10) for Task 1. The test level accuracy is calculated as:

At the **task level**, we consider LLM-generated test cases to be correct only if all test cases can pass the correct code. In the previous example, even though 70% of the test cases for Task 1 can pass the correct code, the task level accuracy would be 0% because not three of the test cases can not pass the correct code.

3.4.2 Coverage. We use the `coverage.py` package to calculate the line-level coverage of the test cases on the correct code provided by the dataset. To calculate the coverage of LLM-generated test cases, we consider two different scenarios based on the accuracy result, i.e., coverage for correct tests at the test level and coverage for correct tests at the task level. The former measures the

[3]Sometimes LLMs generate test cases with some natural language explanations [24, 25].
[4]`coverage.py` Library: https://github.com/nedbat/coveragepy
[5]For the HumanEval, MBPP, and APPS datasets, we use the "canonical solution" provided by the dataset as the correct code in our experiments. For BugsInPy, we use the patched code as the correct code.

percentage of code lines in the correct code executed by all correct tests at the test level. The latter measures the percentage of code lines in the correct code executed by correct tests at the task level.

3.4.3 Bug Detection. To measure the bug detection efficacy of the LLM-generated test cases, we first construct a bug set for each dataset (more details in Sec. 4.2). We then analyze whether the LLM-generated test cases can discover bugs in our constructed bug set.

Similar to the coverage measurement, we consider two different scenarios: (1) bug detection for correct tests at the test level and (2) bug detection for correct tests at the task level. Bug detection for correct tests at the test level measures the percentage of bug code in our constructed code detected by LLM-generated correct tests at the test level. Bug detection for correct tests at the task level measures the percentage of bugs in our constructed code solutions that can be detected by the correct test cases at the task level.

4 EXPERIMENT DESIGN

4.1 Research Questions

This study answers the following questions:

- **RQ1: How do the source code in prompts affect LLMs in test generation?** This RQ investigates the effectiveness of LLM-generate test cases in terms of test case accuracy, coverage, and bug detection effectiveness among the five test case generation prompts. There are three sub-RQs:
  - *RQ1.1 What is the **accuracy** of LLM-generated test cases for the five different prompts?*
  - *RQ1.2: What is the code **coverage** of LLM-generated test cases for the five different prompts?*
  - *RQ1.3: What is the **bug detection effectiveness** of LLM-generated test cases in our constructed pieces for the five different prompts?*
- **RQ2: How does the source of the code influence the LLMs in test generation?** This RQ investigates whether LLM-generated tests are more likely misguided by LLM-generated code rather than our constructed `P_T_CC` and `P_T_IC`.
- **RQ3: To what extent are LLMs misguided by the incorrect code in the prompts in test generation?** This RQ analyzes the percentage of LLM-generated test cases that can pass the incorrect code provided in `P_IC`.
- **RQ4: Do our observations hold for real-world code?** This RQ investigates the effectiveness of LLM-generated test cases based on the source code of real-world tasks.

4.2 Datasets

In our experiments, we first use HumanEval, MBPP, and APPS datasets, which are widely used in LLM-based code generation [9, 24, 27, 33, 72] and LLM-based test case generation [9, 11, 17, 32]. To facilitate a consistent evaluation of test case generation effectiveness across datasets, we convert the prompt format of APPS and MBPP into HumanEval's function-level format for both task description and solutions, which is more easily to evaluate compared to the line-level code script [44]. This conversion constrains the LLMs to generate test cases in a standardized unit test case format, simplifying the evaluation process of the generated test cases. Next, we evaluate the effectiveness of the generated test cases on the BugsInPy dataset, which contains real-world Python programs with known bugs and allows us to analyze how the source code of real-world programs affects the performance of LLM-generated test cases in detecting bugs.

HumanEval. Chen et al. [10] proposes the first code generation dataset that utilizes *pass@k* to analyze the code generation effectiveness of LLMs. HumanEval contains 164 code generation tasks in its original version. In our experiments, some tasks are correctly addressed by all LLMs, which then do not contain incorrect code as `P_T_IC` in our setup. Then we remove these tasks in

Table 2. Code generation datasets used in the experiments. The tokens are calculated based on tiktoken with GPT-4 encoding.

Dataset	Mean Token P_T	Mean Token P_T_CC	Mean Token P_T_IC	Mean Token P_CC	Mean Token P_IC	No. of Problems	No. of Bug code
humaneval	117.19	164.21	198.69	58.85	59.81	85	85
mbpp	122.97	162.79	191.46	51.04	55.89	213	213
apps	486.25	571.12	541.68	94.78	56.11	172	172
BugsInPy	-	-	-	1092.20	904.00	10	10

our experiments. Finally, we collect 85 tasks from the original HumanEval dataset to measure the effectiveness of LLM-generated test cases.

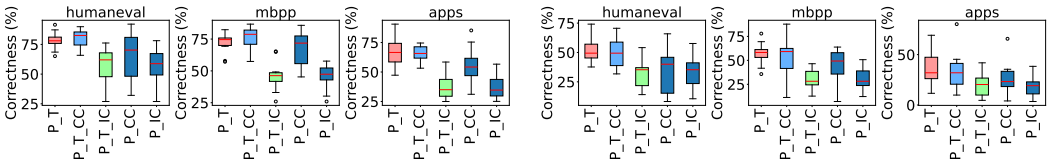
MBPP. [5] contains 974 code generation tasks. In the experiments, we follow the current existing works and utilize the 399 tasks in the MBPP-EvalPlus [41] version to measure the effectiveness of the LLM-generated test cases. Prior to conducting the experiments, we convert the task descriptions of MBPP into the HumanEval function format. Since the correct code provided by MBPP is already at the function level, we directly incorporate the original task prompt into the function. Subsequently, we feed the converted tasks into the evaluated LLMs to generate solutions. For each task, we select an incorrect code from the generated solutions to construct `P_T_IC` and `P_IC`. However, since some tasks do not have incorrect code, we ultimately collect 213 tasks for our experiments.

APPS. [23] contains 5,000 code generation tasks with three levels of difficulty (including 1,000 introductory tasks, 2,000 interview tasks, and 1,000 competition tasks). Prior to conducting the experiments, we first convert the task descriptions into the HumanEval format. Since the correct code provided by APPS is not at the function level, we use GPT-3.5-turbo to convert the correct code into function-level code, filter out incorrect converted functions, and incorporate the original task prompt into the function. After this process, we collect 405 tasks for our experiments. Next, we feed the converted tasks into the evaluated LLMs to generate solutions. For each task, we select an incorrect code from the generated solutions to construct `P_T_IC` and `P_IC`. However, since some tasks do not have incorrect code, we ultimately collect 172 tasks for our experiments.

BugsInPy. [64] contains 493 real bugs from 17 real-world Python programs, including popular libraries such as matplotlib, numpy, pandas, and fastapi. Since tasks in BugsInPy does not exist a predefined task descriptions, we directly use the patched and original code as `P_CC` and `P_IC`, respectively. We choose BugsInPy because the patched code of BugsInPy is primarily focused on in-file functions that do not require calling functions from other files and are not at the class level, which is more suitable for our experiments than other benchmarks such as SWE-bench.

In our experiments, we initially conduct experiments on all 493 tasks from the BugsInPy dataset. However, we observe that for most of the tasks, the generated tests for both prompts are incorrect at both the test level and task level. The primary reason for this is that most of the tasks require 13,000+ input tokens, and this long-context information impairs LLMs' reasoning ability and causes LLMs to struggle in generating useful test cases for the testing process. These tasks are useless for us to investigate the influence of correct code and incorrect code, we therefore remove them and focus on only the remaining 10 tasks from 8 programs, which have correct test cases produced from either correct code or incorrect code.

Our constructed bug set. To measure the bug detection effectiveness of LLM-generated test cases, we construct a bug set with incorrect solutions from the tasks in each dataset. For HumanEval, MBPP, and APPS, we first require our evaluated LLMs to generate code for each task description. Then, we randomly select an incorrect code for each task to construct the bug set. Since some tasks do not have incorrect code, we filter out these tasks during the dataset construction process. Finally,



(a) Test level. (b) Task level.

Fig. 1. RQ1.1: Accuracy of LLM-generated test cases across HumanEval, MBPP, and APPS datasets using different prompts at test level and task level.

we obtain 85, 213, and 172 incorrect code samples for HumanEval, MBPP, and APPS, respectively. For BugsInPy, we directly use the original incorrect code as the bug code.

4.3 Evaluation LLMs

Five open-source LLMs and six closed-source LLMs are used in our experiments. The experiments are conducted on an 8 * H100 server.

4.3.1 Open-Source Models. For open-source LLMs, we evaluate **Meta-Llama-3-8B, CodeLlama-7B-Python-hf, DeepSeek-Coder-6.7B-Instruct, StarCoder2-7B, and Codestral-22B-v0.1** in our experiments. We select these open-source LLMs since they achieve SOTA performance in code generation tasks (e.g., evalplus) with low parameters and then can be conducted with an 8 * H100 server.

4.3.2 Closed-Source Models. We conducted an evaluation of six state-of-the-art closed-source LLMs: **GPT-3.5-turbo, GPT-3.5-turbo-1106, GPT-4-turbo, GPT-4, Claude-3-haiku, and Claude-3-sonnet**. These models exemplify the latest advancements in LLM architecture[6].

4.4 Inference Configuration of LLMs

In our experiments, four parameters affect the LLM response: Temperature, Top-p, Top-K, and max_new_tokens. To ensure consistency in the test cases generated by LLMs across different executions, we set Temperature to 0, Top-p to 1.0, Top-K to 0, and max_new_tokens to 1024. These settings guarantee that the generation process follows a greedy decoding approach[7].

5 RESULTS AND FINDINGS

This section shows the experiment results and the analysis for our RQs.

5.1 RQ1: How do the source code in prompts affect LLMs in test generation?

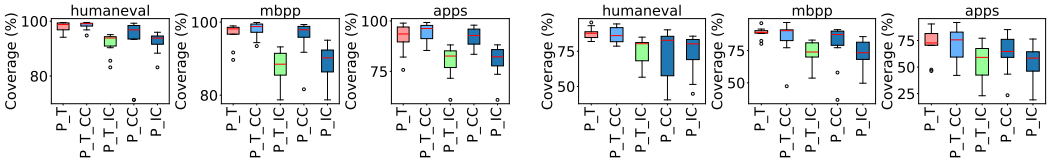
*5.1.1 RQ1.1 What is the **accuracy** of LLM-generated test cases for the five different prompts?*

Test Level Accuracy. Figure 1a presents test level accuracy results across three datasets (HumanEval, MBPP, and APPS) using different prompts for all LLMs. We observe that `P_T`- and `P_T_CC`-guided test generation achieve SOTA performance compared to other prompts. For example, as shown in Tab. 3 *Test level*, for the HumanEval dataset, `P_T_CC` and `P_T` achieve 80.45% and 78.30% test case accuracy on average for all models, while other prompt-guided test generation only achieves 64.05% accuracy. For the MBPP and APPS datasets, we can still observe that `P_T_CC` and

[6]GPT-3.5-turbo and GPT-3.5-turbo-1106 are variants within the GPT-3.5 series. "GPT-3.5-turbo-1106" indicates a release date of June 11, 2023, whereas "GPT-3.5-turbo" refers to a more recent iteration released on January 25, 2024.
[7]We also provide the CodeBLEU scores for five consecutive generations in Sec. 6.2.

Table 3. RQ1.1: Accuracy of LLM-generated test case across HumanEval, MBPP, and APPS datasets

Model	HumanEval					MBPP					APPS				
	P_T	P_T_CC	P_T_IC	P_CC	P_IC	P_T	P_T_CC	P_T_IC	P_CC	P_IC	P_T	P_T_CC	P_T_IC	P_CC	P_IC
Test level															
Meta-Llama-3-8B	68.53	**74.52**	26.93	70.25	26.93	**75.18**	69.89	25.95	54.64	25.95	**66.59**	61.96	28.44	58.04	28.44
CodeLlama-7b-Python-hf	65.13	**65.84**	34.26	47.44	40.51	57.16	**78.74**	33.03	45.28	30.2	**90.60**	74.5	24.97	31.1	25.08
deepseek-coder-6.7b-instruct	**79.85**	71.81	54.85	59.63	58.02	**74.23**	62.3	47.93	56.75	46.53	**72.34**	53.24	29.07	41.16	30.26
starcoder2-7b	**76.88**	74.49	40.78	48.97	38.54	**70.66**	57.43	40.61	52.22	39.27	**76.28**	65.74	29.79	35.23	29.11
Codestral-22B-v0.1	75.45	**82.55**	61.27	77.11	65.02	74.59	71.89	47.81	**78.39**	46.85	47.28	**60.36**	36.8	52.66	36.06
GPT-3.5-turbo	79.95	**82.39**	62.0	32.11	58.28	75.07	**81.07**	46.26	77.03	51.25	56.76	**61.56**	33.58	52.78	34.68
GPT-3.5-turbo-1106	78.05	**85.23**	64.86	33.71	58.79	76.53	**83.06**	45.85	77.79	52.12	60.26	**69.31**	34.99	57.03	33.61
GPT-4-turbo-preview	87.25	**89.92**	72.77	86.57	72.44	82.35	**86.73**	65.36	85.79	57.69	68.4	73.46	50.49	**75.39**	56.65
GPT-4	82.35	**88.86**	76.19	84.77	69.57	77.4	**86.47**	64.89	77.12	57.36	65.44	**74.58**	53.65	65.23	51.36
Claude-3-sonnet	76.25	**83.55**	63.38	71.95	63.33	69.32	**78.82**	49.17	71.82	47.39	53.47	**63.21**	36.95	54.23	35.84
Claude-3-haiku	91.57	85.75	70.98	**92.08**	78.14	57.9	**73.54**	42.88	66.19	52.46	80.89	67.8	58.45	**85.25**	52.76
Overall	78.3	**80.45**	57.12	64.05	57.23	71.85	**75.45**	46.34	67.55	46.1	**67.12**	65.97	37.93	55.28	37.62
Task level															
Meta-Llama-3-8B	**47.06**	43.53	23.53	40.0	23.53	**62.91**	49.77	28.17	38.5	28.17	**36.63**	35.47	22.09	33.14	22.09
CodeLlama-7b-Python-hf	**43.53**	31.76	16.47	8.24	11.76	59.62	**87.79**	21.6	41.31	17.37	61.63	**80.23**	20.35	18.02	19.19
deepseek-coder-6.7b-instruct	**48.24**	38.82	20.0	14.12	23.53	**58.22**	11.27	16.9	7.04	21.6	**31.98**	9.88	6.4	4.07	6.4
starcoder2-7b	**72.94**	70.59	14.12	10.59	10.59	**77.93**	64.79	12.68	15.96	12.21	**69.19**	45.35	4.65	9.3	3.49
Codestral-22B-v0.1	38.82	35.29	36.47	**40.00**	35.29	41.78	31.92	42.72	**49.30**	26.76	11.63	13.95	**40.70**	19.19	38.37
GPT-3.5-turbo	**49.41**	**49.41**	37.65	16.47	37.65	53.99	**60.56**	27.23	57.75	36.15	25.58	**31.98**	10.47	23.26	11.05
GPT-3.5-turbo-1106	51.76	**60.00**	41.18	17.65	40.0	58.22	**64.32**	27.23	58.22	31.92	27.33	**42.44**	9.3	25.0	11.05
GPT-4-turbo-preview	49.41	**60.00**	36.47	51.76	48.24	52.11	**59.15**	38.97	56.81	50.7	26.74	31.98	22.09	**33.72**	19.77
GPT-4	**62.35**	57.65	35.29	52.94	42.35	54.93	60.09	34.74	**61.50**	41.78	37.79	**40.12**	16.86	35.47	18.02
Claude-3-sonnet	37.65	38.82	29.41	**40.00**	30.59	35.68	**38.50**	38.03	32.86	25.82	12.79	13.37	**31.40**	18.6	23.84
Claude-3-haiku	**74.12**	50.59	54.12	65.88	57.65	**69.48**	44.6	46.48	63.85	47.89	56.98	27.33	41.86	**65.70**	34.3
Overall	**52.3**	48.77	31.34	32.51	32.83	**56.81**	52.07	30.43	43.92	30.94	**36.21**	33.83	20.56	25.95	18.87



(a) RQ1.2a: Test level. (b) RQ1.2b: Task level.

Fig. 2. RQ1.2: Coverage of LLM-generated test cases for different prompts.

P_T achieve SOTA test case accuracy compared to other prompts. Furthermore, we observe that without task description guided test case generation, P_CC also achieves SOTA accuracy compared to P_IC. For example, P_CC achieves 64.05% accuracy while P_IC only achieves 57.23% accuracy on average for all models.

Task Level Accuracy. Figure 1b presents the task level accuracy results for different prompts. We can observe that similar to test level accuracy, P_T and P_T_CC still obtain SOTA performance compared to other prompts. For example, For example, as shown in Tab. 3 *Task level*, P_T and P_T_CC achieve 52.30% and 48.77% test case accuracy for the HumanEval dataset, while other prompts only achieve 32.83% accuracy on average for all models.

> ***Answer to RQ1.1****: Incorrect code can significantly impact the ability of LLMs to generate correct tests. For instance, in the HumanEval dataset, LLMs achieve 80.45% test accuracy when provided with* P_T_CC*, but only 57.12% with* P_T_IC*.*

*5.1.2 RQ1.2 What is the code **coverage** of LLM-generated test cases for the five different prompts?*

Table 4. RQ1.2: Code line coverage of LLM-generated correct test case, where we evaluate whether these test cases can cover lines in the correct code in each dataset.

Model	HumanEval					MBPP					APPS				
	P_T	P_T_CC	P_T_IC	P_CC	P_IC	P_T	P_T_CC	P_T_IC	P_CC	P_IC	P_T	P_T_CC	P_T_IC	P_CC	P_IC
Test Level															
Meta-Llama-3-8B	**95.36**	94.77	83.15	93.6	83.15	91.68	**94.38**	78.72	91.74	78.72	82.1	**85.37**	60.87	84.05	60.87
CodeLlama-7b-Python-hf	95.79	**99.07**	90.32	96.83	89.94	89.66	**96.37**	84.03	81.61	83.58	75.74	**88.81**	75.79	83.47	77.31
deepseek-coder-6.7b-instruct	**99.50**	98.24	91.99	97.2	93.23	**98.15**	98.05	86.93	96.94	89.13	89.15	**93.67**	78.04	90.85	78.13
starcoder2-7b	**97.90**	96.82	85.58	93.37	88.34	**96.70**	93.44	79.63	94.74	81.82	**90.06**	87.69	71.58	86.15	73.56
Codestral-22B-v0.1	98.79	**99.16**	95.09	98.29	95.95	98.59	98.36	91.99	**98.82**	93.33	93.56	93.79	88.13	**95.32**	87.94
GPT-3.5-turbo	98.79	**99.00**	93.87	71.32	93.83	98.24	**98.78**	88.48	97.9	90.72	95.68	**96.28**	82.61	93.38	82.24
GPT-3.5-turbo-1106	98.92	**99.13**	94.68	71.32	93.64	97.89	**99.04**	88.48	98.3	90.21	95.95	**97.17**	81.71	92.76	79.17
GPT-4-turbo-preview	99.34	**99.51**	94.15	99.19	94.44	98.96	**99.90**	91.57	98.84	90.05	98.94	**99.26**	88.08	97.92	86.29
GPT-4	**99.48**	99.3	94.03	99.24	94.73	98.96	**99.82**	90.24	99.33	91.4	**98.79**	98.78	83.59	97.52	85.34
Claude-3-sonnet	99.37	**99.52**	94.53	98.55	95.44	98.56	**99.62**	93.21	98.87	94.6	98.1	**98.43**	86.44	96.63	88.16
Claude-3-haiku	94.14	**98.17**	91.52	94.73	94.31	96.67	**99.32**	91.34	97.64	95.04	93.46	**96.73**	83.94	92.77	83.16
Overall	97.94	**98.43**	91.72	92.15	92.45	96.73	**97.92**	87.69	95.89	88.96	91.96	**94.18**	80.07	91.89	80.2
Task level															
Meta-Llama-3-8B	**86.03**	85.0	69.63	84.04	69.63	**91.72**	87.33	73.35	82.56	73.35	77.31	**78.51**	63.75	75.69	63.75
CodeLlama-7b-Python-hf	**83.40**	78.59	59.41	40.43	51.41	89.45	**96.85**	66.64	80.99	59.77	83.25	**92.06**	59.14	58.68	58.45
deepseek-coder-6.7b-instruct	**87.45**	81.64	66.75	55.86	68.42	**88.77**	47.34	58.67	36.75	66.15	**71.50**	42.16	28.94	23.43	29.42
starcoder2-7b	**95.41**	94.38	56.53	46.0	44.73	**96.29**	92.61	53.66	57.96	49.68	**90.66**	83.71	22.95	43.32	19.07
Codestral-22B-v0.1	81.91	79.97	80.33	**82.90**	80.7	82.55	77.24	83.17	**87.50**	73.16	46.26	50.18	**77.21**	62.95	76.43
GPT-3.5-turbo	87.75	**87.96**	81.18	59.25	80.24	89.12	**90.47**	73.47	90.34	80.58	70.4	**75.67**	43.81	64.68	45.44
GPT-3.5-turbo-1106	88.62	**91.73**	83.74	61.0	82.9	90.32	**91.53**	73.91	90.43	77.71	72.76	**83.20**	41.05	66.38	46.8
GPT-4-turbo-preview	88.76	**92.46**	81.09	88.21	85.68	88.34	**90.73**	82.07	89.22	85.87	70.78	74.38	63.02	**76.61**	60.63
GPT-4	**93.13**	91.75	80.97	90.19	83.76	88.92	90.57	80.03	**91.30**	82.46	80.74	**82.77**	57.07	77.75	57.57
Claude-3-sonnet	**83.03**	81.86	73.79	82.76	74.91	80.25	**81.95**	79.63	77.25	70.18	47.6	48.07	**71.28**	59.63	65.15
Claude-3-haiku	**88.91**	86.07	84.98	87.14	85.26	**90.65**	83.59	83.07	89.97	83.73	82.5	68.63	76.83	**85.34**	72.71
Overall	**87.67**	86.49	74.4	70.71	73.42	**88.76**	84.56	73.42	79.48	72.97	**72.16**	70.85	55.0	63.13	54.13

Coverage of Correct Tests at the Test Level. The evaluation results are shown in Figure 2a, where we observe that the code line coverage of both P_T and P_T_CC is higher than other prompt-generated tests. In most models and datasets, P_T_CC also achieves better results compared to P_T, indicating that providing task descriptions and correct code examples to LLMs can guide them in generating test cases that cover more lines of the correct code. For example, as shown in Tab. 4, the code line coverage of P_T_CC and P_T achieves 98.43% and 97.94% in the HumanEval dataset, while P_T_IC only achieves 91.72% code line coverage. Next, we can also observe that in most of the experiments, the code line coverage of P_CC is also higher than P_IC. For example, the code line coverage of P_CC achieves 95.89% while P_IC only achieves 88.96% code line coverage in the MBPP dataset for all models on average. These evaluation results demonstrate that for the code line coverage of LLM-generated correct tests at the test level, test cases generated by the guidance of incorrect code (i.e., P_T_IC and P_IC) have lower coverage than those guided by correct code and even those generated with only the task descriptions provided.

Coverage of Correct Tests at the Task Level. The evaluation results are shown in Figure 2b, where we observe that similar to the code line coverage of correct tests at the test level, P_T and P_T_CC generated test cases achieve higher code line coverage compared to P_T_IC. Moreover, P_T generated tests have competitive code line coverage compared to P_T_CC. For example, as shown in Tab. 4, the overall code line coverage of P_T and P_T_CC is 87.67% and 86.49% in the HumanEval dataset, while P_T_IC only achieves 74.40% coverage. Furthermore, we observe that in the MBPP dataset, providing P_CC to guide test case generation achieves 79.48% code line coverage, which is higher than P_IC with 72.97% coverage. These evaluation results demonstrate that for the code line coverage of most of the LLM-generated test cases, those generated by the guidance of incorrect code (i.e., P_T_IC and P_IC) have lower coverage than those guided by correct code and even those generated with only the task descriptions provided.

Table 5. RQ1.3: Bug detection rate of LLM-generated test cases in our constructed bug set.

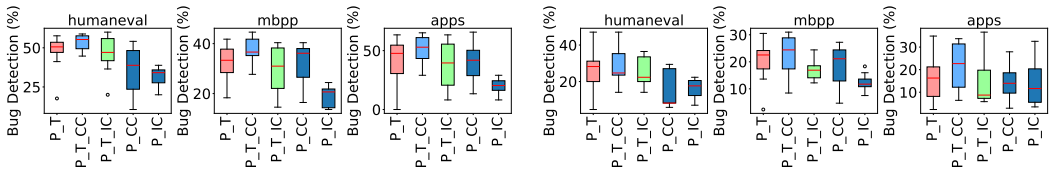
Model	HumanEval					MBPP					APPS				
	P_T	P_T_CC	P_T_IC	P_CC	P_IC	P_T	P_T_CC	P_T_IC	P_CC	P_IC	P_T	P_T_CC	P_T_IC	P_CC	P_IC
Test level															
Meta-Llama-3-8B	41.18	**44.71**	20.0	38.82	20.0	24.41	**29.58**	14.55	26.29	14.55	28.49	**35.47**	8.14	30.23	8.14
CodeLlama-7b-Python-hf	44.71	**57.65**	45.88	35.29	29.41	23.0	**35.21**	19.72	16.43	14.08	0.0	**35.47**	12.79	27.33	17.44
deepseek-coder-6.7b-instruct	**57.65**	50.59	55.29	42.35	35.29	**39.44**	36.15	37.09	31.46	20.66	30.23	**58.14**	48.26	36.63	20.93
starcoder2-7b	**51.76**	48.24	47.06	34.12	25.88	**32.86**	27.7	26.29	26.76	13.62	**32.56**	29.07	31.4	25.58	15.12
Codestral-22B-v0.1	51.76	**57.65**	52.94	48.24	36.47	35.21	36.62	34.74	**40.38**	24.41	50.0	**52.91**	45.93	47.67	23.84
GPT-3.5-turbo	50.59	**52.94**	36.47	12.94	34.12	32.39	**38.97**	22.54	36.15	22.54	47.67	**51.16**	20.93	43.02	20.35
GPT-3.5-turbo-1106	49.41	**55.29**	38.82	11.76	32.94	33.33	**40.38**	21.6	36.15	20.66	51.74	**57.56**	20.35	41.86	19.19
GPT-4-turbo-preview	57.65	58.82	**60.00**	54.12	36.47	41.78	**44.13**	39.44	38.97	19.25	63.37	65.12	63.37	**65.70**	26.16
GPT-4	55.29	**57.65**	56.47	51.76	38.82	40.85	**43.19**	40.38	38.97	24.41	63.37	**63.95**	62.79	54.07	25.0
Claude-3-sonnet	50.59	**57.65**	56.47	48.24	35.29	36.15	**44.60**	39.44	37.09	21.13	57.56	**64.53**	62.79	52.91	29.07
Claude-3-haiku	17.65	**47.06**	44.71	10.59	20.0	18.31	**35.21**	30.99	16.9	14.08	30.81	**52.33**	39.53	13.37	10.47
Overall	48.02	**53.48**	46.74	35.29	31.34	32.52	**37.43**	29.71	31.41	19.04	41.44	**51.43**	37.84	39.85	19.61
Task level															
Meta-Llama-3-8B	28.24	**34.12**	20.0	29.41	20.0	23.47	**24.41**	12.21	21.13	12.21	27.91	**33.72**	32.56	27.91	32.56
CodeLlama-7b-Python-hf	18.82	23.53	**36.47**	5.88	8.24	13.62	**30.99**	16.9	16.43	11.27	13.95	**33.72**	16.86	13.95	19.77
deepseek-coder-6.7b-instruct	**29.41**	23.53	14.12	8.24	16.47	**22.54**	8.45	15.02	4.69	11.27	**9.30**	6.4	8.72	2.91	3.49
starcoder2-7b	**47.06**	**47.06**	34.12	8.24	14.12	**30.52**	24.88	21.13	7.51	12.21	34.88	33.72	**36.63**	4.65	20.35
Codestral-22B-v0.1	21.18	22.35	22.35	**27.06**	21.18	17.37	13.15	17.37	**22.54**	11.74	6.98	9.88	6.4	**10.47**	4.65
GPT-3.5-turbo	**25.88**	24.71	20.0	8.24	17.65	19.72	23.94	14.08	**25.82**	15.96	16.86	**22.09**	5.81	14.53	6.4
GPT-3.5-turbo-1106	28.24	**35.29**	23.53	8.24	20.0	23.94	**29.11**	14.08	26.29	15.02	16.28	**29.07**	5.81	16.86	7.56
GPT-4-turbo-preview	32.94	**37.65**	34.12	28.24	22.35	24.41	**29.11**	19.72	23.0	10.33	17.44	22.67	16.28	**23.26**	20.35
GPT-4	**35.29**	**35.29**	32.94	27.06	22.35	25.82	**28.64**	24.41	27.23	18.31	25.0	28.49	22.67	20.35	**31.98**
Claude-3-sonnet	18.82	**23.53**	18.82	21.18	10.59	17.37	**18.78**	17.37	15.96	7.51	6.98	9.3	8.14	**12.21**	4.65
Claude-3-haiku	4.71	14.12	**21.18**	7.06	7.06	2.35	**15.96**	13.15	9.86	7.51	2.33	**14.53**	8.72	8.72	11.63
Overall	26.42	**29.2**	25.24	16.26	16.36	20.1	**22.49**	16.86	18.22	12.12	16.17	**22.15**	15.33	14.16	14.85

Table 6. RQ1.3: Bug detection rate of LLM-generated test cases on the incorrect code provided with the prompt (P_T_IC).

Model	HumanEval					MBPP					APPS				
	P_T	P_T_CC	P_T_IC	P_CC	P_IC	P_T	P_T_CC	P_T_IC	P_CC	P_IC	P_T	P_T_CC	P_T_IC	P_CC	P_IC
Test level															
Meta-Llama-3-8B	65.88	**67.06**	28.24	63.53	28.24	47.42	**60.56**	26.76	51.17	26.76	37.21	**47.67**	16.86	45.93	16.86
CodeLlama-7b-Python-hf	72.94	**92.94**	60.0	76.47	48.24	45.54	**65.73**	41.78	31.46	31.92	1.74	**47.67**	18.02	44.19	30.81
deepseek-coder-6.7b-instruct	**95.29**	85.88	88.24	81.18	57.65	**76.53**	71.83	69.01	69.95	38.5	37.79	**77.33**	62.79	61.63	33.14
starcoder2-7b	**80.00**	75.29	74.12	63.53	43.53	**65.73**	55.87	56.34	57.28	30.99	45.35	38.95	41.28	**45.93**	28.49
Codestral-22B-v0.1	91.76	**97.65**	92.94	88.24	65.88	77.46	**83.57**	77.0	82.16	48.36	70.35	**76.16**	62.21	72.09	37.79
GPT-3.5-turbo	89.41	**90.59**	60.0	24.71	61.18	76.53	**84.04**	42.25	77.46	46.01	74.42	**75.00**	40.12	70.35	38.37
GPT-3.5-turbo-1106	89.41	**92.94**	64.71	24.71	62.35	72.77	**81.69**	40.85	78.4	44.13	75.0	**77.91**	37.21	66.28	34.3
GPT-4-turbo-preview	95.29	**97.65**	**97.65**	90.59	63.53	83.1	**89.67**	76.53	81.69	41.78	87.79	**90.70**	87.79	**90.70**	43.6
GPT-4	**95.29**	**95.29**	92.94	91.76	64.71	79.81	**85.92**	79.34	84.98	49.3	87.79	**89.53**	**89.53**	82.56	44.19
Claude-3-sonnet	92.94	**97.65**	**97.65**	85.88	62.35	79.81	**85.92**	81.69	81.22	46.01	84.88	**89.53**	87.79	77.33	48.84
Claude-3-haiku	25.88	**68.24**	**68.24**	18.82	28.24	32.86	**69.95**	68.08	30.99	23.47	41.86	**75.58**	59.88	21.51	19.19
Overall	81.28	**87.38**	74.97	64.49	53.26	67.05	**75.89**	59.97	66.07	38.84	58.56	**71.46**	54.86	61.68	34.14
Task level															
Meta-Llama-3-8B	50.59	**52.94**	31.76	49.41	31.76	47.42	**50.70**	21.6	39.91	21.6	38.37	**43.60**	41.28	41.86	41.28
CodeLlama-7b-Python-hf	35.29	41.18	**50.59**	9.41	12.94	30.05	**58.69**	35.21	29.11	17.37	20.93	**43.02**	20.93	20.35	26.16
deepseek-coder-6.7b-instruct	**48.24**	37.65	29.41	15.29	21.18	**47.89**	14.08	28.17	7.98	20.19	**11.05**	8.14	9.3	5.23	4.65
starcoder2-7b	**70.59**	67.06	56.47	12.94	25.88	**63.38**	53.52	46.95	13.15	23.94	**47.09**	43.6	**47.09**	8.14	28.49
Codestral-22B-v0.1	34.12	36.47	36.47	**40.00**	29.41	36.62	30.05	33.33	**46.48**	20.66	11.05	12.79	9.88	**18.02**	8.72
GPT-3.5-turbo	**45.88**	**45.88**	32.94	16.47	30.59	49.3	**55.40**	24.88	52.11	32.39	23.26	**30.23**	9.3	22.67	9.3
GPT-3.5-turbo-1106	48.24	**57.65**	37.65	16.47	35.29	48.83	**57.28**	24.41	52.58	28.64	25.58	**41.28**	8.72	23.84	10.47
GPT-4-turbo-preview	47.06	**57.65**	50.59	47.06	32.94	46.48	**55.40**	37.09	45.07	21.6	24.42	30.81	22.67	**31.98**	26.74
GPT-4	**57.65**	55.29	52.94	50.59	37.65	47.89	53.05	47.42	**56.34**	36.15	36.05	36.05	31.4	33.14	**43.60**
Claude-3-sonnet	34.12	**38.82**	28.24	36.47	21.18	33.8	**34.74**	30.99	28.64	13.15	11.63	12.79	12.79	**17.44**	8.72
Claude-3-haiku	8.24	20.0	**29.41**	10.59	9.41	4.69	26.29	**27.23**	15.02	13.15	6.4	**22.09**	14.53	12.21	13.95
Overall	43.64	**46.42**	39.68	27.7	26.2	41.49	**44.47**	32.48	35.13	22.62	23.26	**29.49**	20.72	21.35	20.19

***Answer to RQ1.2**: Incorrect code also affects the ability of LLMs to generate high-coverage tests. For instance, for the APPS dataset, LLMs achieve 94.18% test-level coverage on average when provided with task descriptions and correct code, but only 80.07% when given task descriptions and incorrect code.*

*5.1.3 RQ1.3: What is the **bug detection effectiveness** of LLM-generated test cases in our constructed pieces for the five different prompts?* To investigate whether LLM-generated test cases can



(a) RQ1.3a: Test level.

(b) RQ1.3b: Task level.

Fig. 3. RQ1.3: Bug detection rate of LLM-generated test cases with different prompts.

detect errors in incorrect code, we evaluated the effectiveness of LLM-generated test cases in our constructed solutions and P_T_IC provided incorrect code for correct test cases at both the test level and the task level.

Bug Detection in Constructed Solutions for the of Correct Tests at the Test Level. The bug detection results of correct test cases are shown in Figure 3a, where we can observe that P_T_CC achieves the SOTA bug detection performance in three datasets, while sometimes the bug detection effectiveness of P_T generated tests competitive results for P_T_CC. For example, as shown in Tab. 5, we can observe that in the HumanEval dataset, on average for all evaluated LLMs, 53.48% bug source code was detected by test cases generated by P_T_CC. While only 48.02% bug source code was detected by P_T generated test cases. Then, we can also observe that in the MBPP and APPS datasets, 37.43% and 51.43% bug source code was detected by P_T_CC generated test cases. While other prompts only detect 32.52% and 41.44% bug source code. Next, we can also observe that for the P_CC and P_IC, tests generated by P_CC also achieve SOTA bug detection effectiveness compared to incorrect code guided test case generation. For example, in the HumanEval dataset,35.29% bug source code was detected by P_CC generated test cases, while only 31.34% bug source code was detected by P_IC generated source code.

Bug Detection in Constructed Solutions for the of Correct Tests at the Task Level. As shown in Figure 3b, we can observe that P_T_CC achieves SOTA performance compared with other prompt-guided test generation methods in most of the experiments. Similar to other metrics, test cases generated by the guidance of P_T also achieve competitive results with P_T_CC in some experiments. As shown in Tab. 5, we can observe that in the HumanEval dataset, tests generated by the guidance of P_T_CC detect 29.20% bug source code, on average for all models, while baselines only detect 26.42% bug source code. In the MBPP and APPS datasets, we can observe that 22.49% and 22.15% bug source codes were also detected by P_T_CC generated tests. However, we can observe that the baselines only detected 20.10% and 16.17% bug source code.

Bug Detection in P_T_IC *solutions for the of Correct Tests at the Test Level.* Tab. 6 *Test level* presents the test level bug detection results for different prompts across the three datasets. We observe that for all datasets, the test cases generated based on P_T_CC achieve the highest bug detection effectiveness compared to other prompts, on average. For example, in the HumanEval dataset, the bug detection effectiveness of P_T_CC-generated test cases is 87.38% on average for all models, while the bug detection effectiveness of P_T_IC-generated test cases is only 74.97%. This indicates that test cases generated based on task description + correct code are more effective in detecting bugs in incorrect code than those generated based on incorrect code.

Bug Detection in P_T_IC *solutions for the of Correct Tests at the Task Level.* Tab. 6 *Task level* presents the task level bug detection results for different prompts across the three datasets. Similar to the test level results, we observe that for all datasets, the test cases generated based on P_T_CC

Table 7. RQ2: Accuracy differences between the test cases generated by the correct code and incorrect code for different sources of code. We calculate the diff_absolute as the difference between the accuracy of P_T_CC and P_T_IC, and the diff_relative as the diff_absolute divided by the accuracy of P_T_CC. Others refer to the results based on code provided by other sources, while Own refers to the results based on the LLM's own generated code.

Model	HumanEval				MBPP				APPS			
	diff_absolute		diff_relative		diff_absolute		diff_relative		diff_absolute		diff_relative	
	Others	Own	Others	Own	Others	Own	Others	Own	Others	Own	Others	Own
Test level												
CodeLlama-7b-Python-hf	**-24.93**	-42.39	**-51.37**	-109.82	**36.19**	-27.80	**45.71**	-144.57	**28.85**	19.55	**31.81**	40.13
deepseek-coder-6.7b-instruct	**9.56**	-2.90	**12.43**	-4.24	13.03	**15.68**	19.24	**21.84**	0.21	**13.05**	0.37	**23.37**
starcoder2-7b	18.15	**41.05**	18.15	**41.05**	**-12.51**	-28.39	**-30.86**	-93.57	-8.47	**21.00**	-16.94	**24.00**
Codestral-22B-v0.1	2.37	**5.88**	2.86	**7.90**	18.42	**20.53**	24.69	**29.24**	**22.52**	17.96	**35.65**	31.00
GPT-3.5-turbo	**26.37**	16.79	**31.86**	20.81	**47.15**	23.45	**55.67**	28.67	**47.64**	13.33	**75.01**	21.34
GPT-3.5-turbo-1106	10.82	**21.09**	12.96	**24.76**	**49.75**	27.99	**57.96**	33.19	**53.40**	17.46	**75.30**	27.43
GPT-4-turbo-preview	**5.55**	-1.97	**6.19**	-2.24	**16.83**	14.35	**19.41**	16.53	**19.02**	12.04	**25.52**	16.07
GPT-4	-6.57	**16.12**	-7.44	**18.31**	**13.13**	12.74	**14.89**	15.57	**23.33**	11.79	**30.62**	16.30
Claude-3-sonnet	-0.55	**-0.50**	-0.67	**-0.65**	**15.48**	12.18	**19.05**	15.85	**21.49**	4.68	**28.70**	7.83
Claude-3-haiku	-1.90	**0.24**	-2.23	**0.30**	20.54	**23.05**	27.64	**29.89**	3.43	**10.67**	4.75	**15.47**
Overall	**10.46**	7.71	**12.52**	9.64	**23.23**	9.60	**31.05**	14.22	**17.49**	8.16	**27.80**	13.77
Task level												
Meta-Llama-3-8B	**83.33**	77.38	**83.33**	77.38	**46.54**	11.15	**77.57**	27.87	-3.49	**-15.12**	**0.00**	**0.00**
CodeLlama-7b-Python-hf	**17.50**	16.25	**29.17**	**40.62**	**21.04**	-15.12	**33.66**	**0.00**	**19.30**	13.93	**36.67**	**44.11**
deepseek-coder-6.7b-instruct	**27.95**	-1.47	**70.39**	-9.09	-17.17	**-3.08**	-103.87	**-23.30**	**0.40**	-1.81	**3.20**	-28.96
starcoder2-7b	40.96	**48.19**	40.96	**48.19**	**17.16**	-2.04	**24.02**	-3.17	-16.37	**14.01**	-49.11	**21.01**
Codestral-22B-v0.1	-4.42	**-5.88**	-12.02	**-19.99**	**13.27**	17.23	**35.98**	**63.93**	**13.99**	8.98	**78.20**	**77.55**
GPT-3.5-turbo	**28.62**	30.07	**53.38**	**54.60**	**47.76**	27.75	**72.59**	43.48	**33.50**	27.90	**90.37**	63.45
GPT-3.5-turbo-1106	13.77	**24.88**	23.66	**42.74**	**55.59**	36.30	**78.75**	53.41	**39.23**	15.77	**87.18**	45.06
GPT-4-turbo-preview	**8.02**	5.56	**13.82**	10.01	**36.59**	30.36	**59.81**	50.71	**23.06**	15.68	**72.33**	47.04
GPT-4	-15.02	**10.99**	-26.63	**20.41**	**22.73**	22.06	**35.52**	34.83	**24.72**	10.23	**58.91**	24.79
Claude-3-sonnet	**11.48**	-1.29	**28.39**	-4.66	**12.86**	16.78	**32.43**	47.48	**21.41**	15.20	**64.24**	63.84
Claude-3-haiku	**9.41**	-7.26	**18.26**	-20.83	**22.64**	27.86	**55.46**	67.13	-3.90	**15.03**	-11.70	**46.90**
Overall	**20.15**	17.95	**29.34**	21.76	**25.36**	15.39	**36.54**	32.94	**13.80**	10.89	**39.12**	36.80

generally achieve the highest bug detection effectiveness compared to other prompts. For example, in the HumanEval dataset, the task level bug detection effectiveness of P_T_CC-generated test cases is 46.42% on average for all models, while the task level bug detection effectiveness of P_T_IC-generated test cases is only 39.68%.

> ***Answer to RQ1.3****: LLM-generated test cases based on* P_T_CC *(task description + correct code) achieve the highest bug detection ratio across all datasets. For example, in the HumanEval dataset, the test level bug detection effectiveness of* P_T_CC*-generated test cases is 87.38% on average for all models, while the bug detection effectiveness of* P_T_IC*-generated test cases is 74.97%.*

5.1.4 Case Analysis. Here we provide a case illustration to demonstrate why the effectiveness of P_T_IC generated tests is lower than P_T_CC. As shown in Figure 4 P_T_CC, we can observe that the P_T_CC provided solution is correct, where the code ensures that the values between a and b are digits (0-9). Then, for the test case `assert generate_intergers(10,14)` its output is `[]` since no digits between 10 and 14. However, as shown in Figure 4 P_T_IC, we can observe that the solution in P_T_IC does not consider digits ranging from 0 to 9, then generates test case `assert generate_intergers(28, 36) == [28,30,32,34,36]`.

5.2 RQ2: How does the source of the code influence the LLMs in test generation?

To explore whether LLMs are more easily misled by the code they generate themselves (Own) compared to directly using source code produced elsewhere (Others), for each LLM, we compare the

P_T_CC

```
def generate_integers(a, b):
    """
    Given two positive integers a and b,
    ↪  return the even digits between a
    and b, in ascending order.

    For example:
    generate_integers(2, 8) => [2, 4, 6, 8]
    generate_integers(8, 2) => [2, 4, 6, 8]
    generate_integers(10, 14) => []
    """

    lower = max(2, min(a, b))
    upper = min(8, max(a, b))

    return [i for i in range(lower, upper+1)
    ↪  if i % 2 == 0]
```

P_T_IC

```
def generate_integers(a, b):
    """
    Given two positive integers a and b,
    ↪  return the even digits between a
    and b, in ascending order.

    For example:
    generate_integers(2, 8) => [2, 4, 6, 8]
    generate_integers(8, 2) => [2, 4, 6, 8]
    generate_integers(10, 14) => []
    """
    # Ensure a is less than or equal to b
    if a > b:
        a, b = b, a

    return [x for x in range(a, b + 1) if x %
    ↪  2 == 0]
```

Test Case (P_T_CC)

```
assert generate_integers(2, 8) == [2, 4, 6, 8]
assert generate_integers(8, 2) == [2, 4, 6, 8]
assert generate_integers(10, 14) == []
assert generate_integers(3, 7) == [4, 6]
assert generate_integers(5, 5) == []
assert generate_integers(0, 0) == []
assert generate_integers(0, 2) == [2]
assert generate_integers(-2, 2) == [2]
```

Test Case (P_T_IC)

```
assert generate_integers(2, 8) == [2, 4, 6, 8]
assert generate_integers(1, 7) == [2, 4, 6]
assert generate_integers(4, 4) == [4]
assert generate_integers(5, 5) == []
assert generate_integers(8, 2) == [2, 4, 6, 8]
assert generate_integers(10, 14) == []
assert generate_integers(-4, 2) == [-4, -2, 0,
↪  2]
assert generate_integers(28, 36) == [28, 30,
↪  32, 34, 36]
```

Fig. 4. Example of prompt-based and completed-code-based test case generation by GPT-4-turbo. The prompt provides a function skeleton to generate even integers between two given numbers. When generating test cases directly from the prompt, the GPT-4-turbo correctly focuses on the digits (0-9). In contrast, when generating test cases from the completed code, GPT-4-turbo considers numbers beyond the digit range (e.g., 28 and 36), illustrating that the incorrect code affects the test case accuracy when we feed the completed code into the GPT-4-turbo.

accuracy of test cases generated with 1) P_T_CC with correct code produced elsewhere; 2) P_T_CC with correct code generated by itself; 3) P_T_IC with incorrect code produced elsewhere; 4) P_T_IC with incorrect code generated by its own. Then we report the evaluation results by calculating the **diff_absolute** between the accuracy of P_T_CC - the accuracy of P_T_IC, and **diff_relative**, i.e., diff_absolute/accuracy of P_T_CC. The comparison is based on identical coding tasks.

Test level. Tab. 7 *Test level* presents the test level results of diff_absolute and diff_relative across the three datasets. We can observe that with Others provided solutions, the diff_absolute and diff_relative would be larger than the results based on the LLM itself generated source codes. For example, the diff_absolute of Others achieves 10.46% in the HumanEval dataset on average for all models, while the diff_absolute of Own only achieves 7.71%. In addition, we can observe that the diff_relative of the Others is also higher than the diff_relative of Own on average for all models. For example, the diff_relative of Others achieves 12.52% in the HumanEval dataset on average for all models, while the diff_relative of Own achieves 9.64%.

Task level. Tab. 7 *Task level* present the task level results of diff_absolute and diff_relative across the three datasets. We can observe that similar to the results of *Test level*, the diff_absolute of the

Table 8. RQ3: Pass rate of the LLM-generated test cases in the incorrect code provided by P_IC.

Model	HumanEval				MBPP				APPS			
	Test Level		Task Level		Test Level		Task Level		Test Level		Task Level	
	P_CC	P_IC	P_CC	P_IC	P_CC	P_IC	P_CC	P_IC	P_CC	P_IC	P_CC	P_IC
Meta-Llama-3-8B	9.57	**19.60**	15.29	**36.47**	17.1	**20.36**	21.6	**39.44**	3.53	**19.15**	15.7	**61.05**
CodeLlama-7b-Python-hf	27.81	**30.74**	5.88	**18.82**	16.23	**20.93**	**58.22**	27.23	12.86	**19.68**	29.65	**43.60**
deepseek-coder-6.7b-instruct	30.92	**39.15**	10.59	**23.53**	21.87	**26.61**	7.51	**23.94**	18.71	**21.58**	5.81	**10.47**
starcoder2-7b	22.93	**23.80**	10.59	**23.53**	21.94	**27.56**	10.33	**23.00**	14.45	**23.62**	5.23	**28.49**
Codestral-22B-v0.1	45.5	**46.64**	20.0	**22.35**	28.56	**29.82**	**18.31**	17.84	23.23	**30.89**	3.49	**18.60**
GPT-3.5-turbo	8.45	**41.20**	5.88	**24.71**	27.51	**31.57**	20.19	**23.47**	21.8	**28.31**	7.56	**14.53**
GPT-3.5-turbo-1106	10.03	**39.73**	8.24	**27.06**	27.05	**31.55**	19.25	**24.41**	**25.80**	24.52	9.88	**13.95**
GPT-4-turbo-preview	44.33	**58.86**	27.06	**45.88**	31.47	**42.82**	23.47	**43.19**	29.56	**55.47**	7.56	**39.53**
GPT-4	43.63	**55.86**	23.53	**38.82**	28.14	**40.13**	22.07	**38.03**	26.09	**49.70**	12.79	**51.74**
Claude-3-sonnet	39.63	**46.18**	20.0	**22.35**	24.16	**31.48**	11.27	**17.37**	21.48	**35.79**	4.07	**25.00**
Claude-3-haiku	54.72	**54.84**	**42.35**	35.29	34.43	**35.10**	**31.46**	29.11	**53.44**	46.45	**42.44**	40.12
Overall	30.68	**41.51**	17.22	**28.98**	25.31	**30.72**	22.15	**27.91**	22.81	**32.29**	13.11	**31.55**

Others is also higher than the diff_absolute of the Own on average for all models across three datasets. For example, the diff_absolute of the Others achieves 20.15% on average for the HumanEval, while the diff_absolute of the Own achieves 17.95%. Furthermore, we can also observe that the diff_relative of Others is also higher than the diff_relative of Own on average for all models across three datasets. For example, the diff_relative of Others achieves 29.34% in the HumanEval dataset on average for all models, while the diff_relative of Own achieves 21.76%.

> ***Answer to RQ2****: LLMs are less likely to be misguided by their own-generated code. For example, the diff_absolute of Own is only 7.71% for the test in the HumanEval dataset on average across all models, while the diff_absolute of Others achieve 10.46%.*

5.3 RQ3: To what extent are LLMs misguided by the incorrect code in the prompts in test generation?

To investigate whether LLMs align with incorrect code and generate test cases that incorrectly pass, we evaluated the pass rate of test cases generated by P_CC and P_IC on the incorrect code provided by P_IC[8]. The evaluation results, presented in Tab. 8, demonstrate that LLMs tend to align with the incorrect code and generate test cases that inappropriately pass the incorrect implementations. For instance, in the HumanEval dataset, the average pass rate of P_IC-generated test cases across all LLMs at the test level is 41.51%, while P_CC achieves only 30.68%. In addition, the average pass rate for all LLMs of P_IC-generated test cases achieves 28.98% in the HumanEval dataset at the task level, but P_CC-generated test cases only have 17.22%, which further demonstrates that LLMs tend to generate incorrect test cases to pass the provided incorrect code.

> ***Answer to RQ3****: Incorrect code misleads LLMs into generating more test cases that pass the incorrect code. For example, in the HumanEval dataset, the test pass rate on incorrect code is 41.51% with tests generated with* P_IC*, but is only 30.68% with tests generated with* P_CC*.*

5.4 RQ4: Do our observations hold for real-world code?

To analyze whether the prompts affect the test case effectiveness in real-world tasks, we conducted experiments on 10 tasks selected from BugsInPy [64]. Since the collected functions can only be

[8]We do not use accuracy as a measure for the passed tests since the source code being evaluated is incorrect.

Table 9. RQ4: Effectiveness of test cases generated by LLMs using different source code-guided test case generation in the BugsInPy dataset.

Model	Accuracy				Coverage				Bug Detection			
	Test Level		Task Level		Test Level		Task Level		Test Level		Task Level	
	P_CC	P_IC	P_CC	P_IC	P_CC	P_IC	P_CC	P_IC	P_CC	P_IC	P_CC	P_IC
Meta-Llama-3-8B	6.98	**8.57**	**0.00**	0.00	**22.04**	22.04	**0.00**	0.00	**0.00**	0.00	**0.00**	0.0
CodeLlama-7b-Python-hf	**7.84**	4.17	**0.00**	0.00	**22.10**	21.98	**0.00**	0.00	**10.00**	0.0	**0.00**	0.00
deepseek-coder-6.7b-instruct	15.56	**24.53**	**0.00**	0.00	**22.04**	21.98	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00
starcoder2-7b	0.0	**2.13**	**0.00**	0.00	21.73	**21.98**	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00
Codestral-22B-v0.1	**34.62**	32.0	**0.00**	0.00	**22.10**	22.04	**0.00**	0.00	**10.00**	0.0	**0.00**	0.00
GPT-3.5-turbo	**23.33**	15.15	**0.00**	0.00	**22.10**	22.04	**0.00**	0.00	**10.00**	0.0	**0.00**	0.00
GPT-3.5-turbo-1106	18.75	**20.00**	**0.00**	0.00	**22.10**	22.04	**0.00**	0.00	**10.00**	0.0	**0.00**	0.0
GPT-4-turbo-preview	**22.22**	21.43	**0.00**	0.00	22.04	**22.36**	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00
GPT-4	**23.25**	21.37	**0.00**	0.00	**22.31**	20.16	**0.00**	0.00	**10.00**	0.0	**0.00**	0.00
Claude-3-sonnet	16.44	**19.35**	**0.00**	0.00	**22.10**	22.04	**0.00**	0.00	**10.00**	0.0	**0.00**	0.00
Claude-3-haiku	**13.00**	11.0	**0.00**	0.00	**21.73**	19.14	**0.00**	0.00	0.0	**10.00**	**0.00**	0.00
Overall	**16.54**	16.34	**0.00**	0.00	**22.01**	21.79	**0.00**	0.00	**5.45**	0.91	**0.00**	0.0

classified into P_CC and P_IC, we then report the evaluation results for P_CC and P_IC of the effectiveness of the test case.

The evaluation results are shown in Tab. 9, where we can observe that first, for the test level evaluation, the effectiveness of the test cases generated by P_CC is higher than P_IC for most of the experiments. For example, we can observe that the accuracy of test cases generated by P_CC achieves 16.54%, while test cases generated by P_IC only achieve 16.34% on average for all models. Next, the code line coverage of test cases generated by P_CC achieves 22.01%, while test cases generated by P_IC only achieve 21.79% on average for all models. Furthermore, we can also observe that the bug detection effectiveness of test cases generated by P_CC achieves 5.45%, while P_IC only achieves 0.91% on average for all models for the accuracy of task level.

Although the average accuracy and coverage of test cases generated by P_CC are higher than those generated by P_IC, the difference between the two prompts is minimal. For instance, the difference in accuracy and coverage between P_CC and P_IC is only 0.20% and 0.22%, respectively. This can be attributed to the large number of input tokens in BugsInPy's tasks. The average input tokens for P_CC and P_IC are 1092.2 and 904.0, respectively, which is significantly higher than the average input token usage in HumanEval, which requires only 58.85 and 59.81 input token usage, respectively. As demonstrated by Levy et al. [35], longer input tokens can negatively impact the reasoning effectiveness of LLMs. Consequently, LLMs struggle to generate high-effectiveness test cases for both prompts, resulting in similar evaluation results.

***Answer to RQ4**: Incorrect code impacts the ability of LLMs to generate more effective test cases compared to the correct code. For example, providing correct code in the prompt yields 5.45% bug detection effectiveness on average for all LLMs in the test level, while providing incorrect code only achieves 0.91% bug detection results.*

6 DISCUSSION

6.1 Correlation between the code generation capability of LLMs and their ease of being misled during test generation

We provide the correlation between the code accuracy (%) and the test accuracy (%) generated by LLMs in Tab. 10. We can observe that there exists a negative correlation for the code accuracy (%) and the test accuracy (%) generated by LLMs for P_IC, where the r ranges from -0.49 to -0.92 for

Table 10. Correlation between the code generation capability of LLMs and how easily their test generation can be misguided. The results are presented in correlation (p-value) format.

Prompt	Pearson			Spearman			Kendall's tau		
	HumanEval	MBPP	APPS	HumanEval	MBPP	APPS	HumanEval	MBPP	APPS
P_T	**-0.75 (0.01)**	-0.48 (0.14)	0.29 (0.39)	**-0.65 (0.03)**	**-0.75 (0.01)**	0.26 (0.43)	**-0.51 (0.03)**	**-0.59 (0.01)**	0.13 (0.65)
P_T_CC	**-0.77 (0.01)**	-0.57 (0.07)	-0.53 (0.09)	**-0.59 (0.05)**	**-0.72 (0.01)**	-0.55 (0.08)	-0.44 (0.06)	**-0.56 (0.02)**	-0.45 (0.06)
P_T_IC	-0.45 (0.16)	-0.59 (0.06)	-0.08 (0.82)	-0.36 (0.27)	-0.25 (0.46)	-0.26 (0.43)	-0.22 (0.35)	-0.22 (0.34)	-0.13 (0.65)
P_CC	-0.21 (0.53)	**-0.88 (0.00)**	-0.57 (0.06)	-0.26 (0.44)	**-0.80 (0.00)**	-0.55 (0.08)	-0.18 (0.43)	**-0.64 (0.01)**	-0.38 (0.12)
P_IC	**-0.89 (0.00)**	**-0.92 (0.00)**	**-0.62 (0.04)**	**-0.66 (0.03)**	**-0.87 (0.00)**	**-0.65 (0.03)**	**-0.55 (0.02)**	**-0.75 (0.00)**	**-0.49 (0.04)**

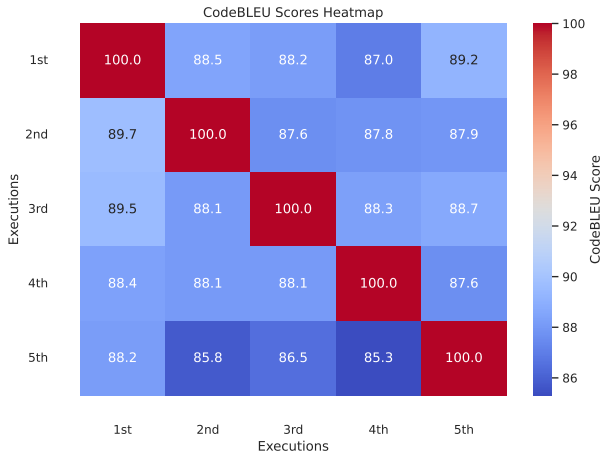


Fig. 5. CodeBLEU scores of GPT-3.5-turbo generated test cases across five executions.

three correlations, and the p-value ranges from 0.00 to 0.04 in our correlation experiments. Next, we can also observe that the p-value of other experiments of other prompts and datasets is always larger than 0.05, indicating that there is no significant correlation between the code accuracy (%) and the test accuracy (%) for these prompts and datasets.

6.2 Randomness of LLM-generated test cases

LLMs are non-deterministic for constrained inputs, which means that the response to the same input may vary across different executions. In our study, we attempt to utilize greedy decoding to constrain the response of the LLMs for the same input to produce identical results. We set the temperature to 0, Top K to 1, and Top P to 1. In this section, we analyze whether greedy decoding can ensure consistent results by calculating the CodeBLEU scores of GPT-3.5-turbo generated tests across five different execution times. The evaluation results are presented in Figure 5. We can observe that the CodeBLEU scores of GPT-3.5-turbo for five different executions are consistently above 85.3% for each pairwise comparison. However, the scores do not reach 100% between any two execution times, indicating that there is still some variation in the generated tests despite the use of greedy decoding.

6.3 Implications for researchers and developers

Based on our findings, we present implications for researchers and developers utilizing LLMs for test case generation. Most importantly, our findings indicate that LLM-based testing is more effective at generating tests that protect mature code from regression errors. However, when applied during

the early stages of software development on relatively immature code, it is more likely to reinforce existing errors.

Prioritizing correct code and task descriptions is crucial, as our results demonstrate that providing both to LLMs yields the most effective test cases. However, if the correctness of the source code cannot be guaranteed, providing only the task description can still lead to better results than providing incorrect code.

In addition, it is essential to be cognizant of LLM limitations when working with real-world code, as the effectiveness of LLM-generated test cases is significantly lower in complex, real-world scenarios compared to simpler benchmark datasets (e.g., longer context, function call, and class level tasks), highlighting the need for further research to improve the effectiveness of LLMs in generating test cases for long-context, real-world tasks.

7 THREAT TO VALIDITY

The threat to internal validity lies in the implementation of the empirical study and the analysis of the evaluation results. To reduce the first threat, the authors carefully checked the code twice during the implementation stage and experiment result analysis stage. To reduce the second threat, the two authors independently analyzed the experiment results and drew experimental conclusions separately. In cases where the conclusions differed, a third, more senior author was consulted to discuss the findings and determine the final result.

The threat to external validity lies in the datasets and the measure tool used in our study. To reduce the threat in our study, we select the three most widely used datasets (i.e., HumanEval, MBPP, and APPS,) and one real-world dataset (i.e., BugsInPy) in code generation tasks to measure the effectiveness of LLM-generated test cases. The evaluated subset for each dataset is checked by analyzing whether each task has an incorrect code in all LLM-generated code that can be used for `P_T_IC`. To measure the accuracy of LLM-generated tests, we also use the evaluation tool of HumanEval to ensure the results are correct. Besides, we also use `coverage.py` to measure the code line coverage of LLM-generated test cases in the correct code, where `coverage.py` is also widely used by developers and can be relied upon to provide accurate results.

The threat to construction validity lies in the randomness of LLM-generated responses. Since LLMs are non-determinized for their generated response in several different executions with the same input [52]. To reduce the randomness of LLM-generated response that would be used to measure the effectiveness of test cases. We use greedy decoding in all of the steps where LLMs would used to generate the response. Moreover, we provided the CodeBLEU results of five different executions of generated tests to demonstrate that our results can reduce the randomness in our experiments, enhancing the overall reliability of our findings.

8 CONCLUSION

In this paper, we present the first empirical study on how source code affects the effectiveness of LLM-generated test cases in code generation tasks. We evaluate the effectiveness of test cases by measuring their accuracy, coverage, and bug detection effectiveness across three widely studied code generation datasets, (i.e., HumanEval, MBPP, and APPS), and one real-world GitHub patching dataset (i.e., BugsInPy). Our evaluation results in five open-source and six closed-source models demonstrate that the effectiveness of LLM-generated test cases is highly affected by the prompts used. Providing task descriptions with correct code in the prompt generally leads to higher test case accuracy, better code coverage, and higher bug detection rates compared to other prompts. For example, `P_T_CC` achieves 80.45% test case accuracy in the HumanEval dataset on average for all LLMs in the test level but other prompts only achieve 64.05% accuracy in the HumanEval dataset. Next, we can also observe that `P_T_CC` also has higher code line coverage compared to

other prompts. For example, the average code line coverage for all models of P_T_CC achieves 94.18% in the APPS dataset for the test level, while the average code line coverage of other prompts only achieves 91.96%. Additionally, the bug detection effectiveness of LLM-generated test cases has a similar trend for accuracy and code line coverage. For example, the average bug detection effectiveness of P_T_CC achieves 51.43% in the APPS dataset, while other prompts only achieve 41.44% bug detection effectiveness.

9 DATA AVAILABILITY

We release our source code, datasets, and results in https://anonymous.4open.science/r/FSE2025-360/.

REFERENCES

[1] Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2023. AVATAR: A Parallel Corpus for Java-Python Program Translation. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 2268–2281. https://doi.org/10.18653/V1/2023.FINDINGS-ACL.143

[2] Toufique Ahmed and Premkumar T. Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 177:1–177:5. https://doi.org/10.1145/3551349.3559555

[3] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, and Leandro von Werra et.al. 2023. SantaCoder: don't reach for the stars! *CoRR* abs/2301.03988 (2023). https://doi.org/10.48550/ARXIV.2301.03988 arXiv:2301.03988

[4] Andrea Arcuri. 2018. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering* 23 (2018), 1959–1981.

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

[6] 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. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021). arXiv:2108.07732 https://arxiv.org/abs/2108.07732

[7] Evelyn M Boyd and Ann W Fales. 1983. Reflective learning: Key to learning from experience. *Journal of humanistic psychology* 23, 2 (1983), 99–117.

[8] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, and Dario Amodei et.al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[9] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

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

[11] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. *CoRR* abs/2304.05128 (2023). https://doi.org/10.48550/ARXIV.2304.05128 arXiv:2304.05128

[12] Jianbo Dai, Jianqiao Lu, Yunlong Feng, Rongju Ruan, Ming Cheng, Haochen Tan, and Zhijiang Guo. 2024. MHPP: Exploring the Capabilities and Limitations of Language Models Beyond Basic Code Generation. https://api.semanticscholar.org/CorpusID:269922079

[13] DeepSeekAI. 2023. DeepSeek Coder: Let the Code Write Itself. https://deepseekcoder.github.io/

[14] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[15] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large Language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via FuzzGPT. *CoRR* abs/2304.02014 (2023). https://doi.org/10.48550/ARXIV.2304.02014 arXiv:2304.02014

[16] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[17] Zhiyu Fan, Haifeng Ruan, Sergey Mechtaev, and Abhik Roychoudhury. 2018. Oracle-guided Program Selection from Large Language Models. (2018).

[18] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. 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. https://openreview.net/pdf?id=hQwb-lbM6EL

[19] Luyu Gao, Zhuyun Dai, Panupong Pasupat, Anthony Chen, Arun Tejasvi Chaganty, Yicheng Fan, Vincent Y. Zhao, Ni Lao, Hongrae Lee, Da-Cheng Juan, and Kelvin Guu. 2023. RARR: Researching and Revising What Language Models Say, Using Language Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 16477–16508. https://doi.org/10.18653/V1/2023.ACL-LONG.910

[20] Md. Mahim Anjum Haque, Wasi Uddin Ahmad, Ismini Lourentzou, and Chris Brown. 2022. FixEval: Execution-based Evaluation of Program Fixes for Competitive Programming Problems. *CoRR* abs/2206.07796 (2022). https://doi.org/10.48550/ARXIV.2206.07796 arXiv:2206.07796

[21] Masum Hasan, Tanveer Muttaqueen, Abdullah Al Ishtiaq, Kazi Sajeed Mehrab, Md. Mahim Anjum Haque, Tahmid Hasan, Wasi Uddin Ahmad, Anindya Iqbal, and Rifat Shahriyar. 2021. CoDesc: A Large Code-Description Parallel Dataset. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021 (Findings of ACL, Vol. ACL/IJCNLP 2021)*, Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (Eds.). Association for Computational Linguistics, 210–218. https://doi.org/10.18653/V1/2021.FINDINGS-ACL.18

[22] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html

[23] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. *NeurIPS* (2021).

[24] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352* (2023).

[25] Dong Huang, Qi Bu, Yuhao Qing, and Heming Cui. 2023. CodeCoT: Tackling Code Syntax Errors in CoT Reasoning for Code Generation. https://api.semanticscholar.org/CorpusID:261030533

[26] Dong Huang, Qi Bu, J Zhang, Xiaofei Xie, Junjie Chen, and Heming Cui. 2023. Bias Testing and Mitigation in LLM-based Code Generation. https://api.semanticscholar.org/CorpusID:262824773

[27] Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation. *arXiv preprint arXiv:2312.13010* (2023).

[28] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1430–1442. https://doi.org/10.1109/ICSE48619.2023.00125

[29] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. *CoRR* abs/2306.02907 (2023). https://doi.org/10.48550/ARXIV.2306.02907 arXiv:2306.02907

[30] Yazhuo Jin. 2024. Generating syntactically and semantically valid test cases for fuzzing JavaScript engines. In *Fifth International Conference on Computer Communication and Network Security (CCNS 2024)*, Vol. 13228. SPIE, 210–215.

[31] Julia Kreutzer, Shahram Khadivi, Evgeny Matusov, and Stefan Riezler. 2018. Can Neural Machine Translation be Improved with User Feedback?. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 3 (Industry Papers)*, Srinivas Bangalore, Jennifer Chu-Carroll, and Yunyao Li (Eds.). Association for Computational Linguistics, 92–105. https://doi.org/10.18653/V1/N18-3012

[32] Shuvendu K Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, et al. 2022. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950* (2022).

[33] Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2023. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *arXiv preprint arXiv:2310.08992*

(2023).

[34] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[35] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same task, more tokens: the impact of input length on the reasoning performance of large language models. *arXiv preprint arXiv:2402.14848* (2024).

[36] Kefan Li and Yuan Yuan. 2024. Large Language Models as Test Case Generators: Performance Evaluation and Enhancement. *arXiv preprint arXiv:2404.13340* (2024).

[37] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, and Harm de Vries et.al. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023). https://doi.org/10.48550/ARXIV.2305.06161 arXiv:2305.06161

[38] Vincent Li and Nick Doiron. 2023. Prompting code interpreter to write better unit tests on quixbugs functions. *arXiv preprint arXiv:2310.00483* (2023).

[39] Yujia Li, David H. 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. 2022. Competition-Level Code Generation with AlphaCode. *CoRR* abs/2203.07814 (2022). https://doi.org/10.48550/ARXIV.2203.07814 arXiv:2203.07814

[40] Yuchao Liao, Tosiron Adegbija, and Roman Lysecky. 2024. Are LLMs Any Good for High-Level Synthesis? *arXiv preprint arXiv:2408.10428* (2024).

[41] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[42] Jianqiao Lu, Zhiyang Dou, Hongru Wang, Zeyu Cao, Jianbo Dai, Yingjia Wan, Yinya Huang, and Zhijiang Guo. 2024. AutoCV: Empowering Reasoning with Automated Process Labeling via Confidence Variation. https://api.semanticscholar.org/CorpusID:270063532

[43] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. 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*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html

[44] Noble Saji Mathews and Meiyappan Nagappan. 2024. Test-Driven Development for Code Generation. *arXiv preprint arXiv:2402.13521* (2024).

[45] Janet Metcalfe. 2017. Learning from errors. *Annual review of psychology* 68 (2017), 465–489.

[46] Amir M. Mir, Evaldas Latoskinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2241–2252. https://doi.org/10.1145/3510003.3510124

[47] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. 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. https://openreview.net/pdf?id=iaYcJKpY2B_

[48] Changan Niu, Ting Zhang, Chuanyi Li, Bin Luo, and Vincent Ng. 2024. On Evaluating the Efficiency of Source Code Generated by LLMs. *arXiv preprint arXiv:2404.06041* (2024).

[49] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is Self-Repair a Silver Bullet for Code Generation?. In *The Twelfth International Conference on Learning Representations*.

[50] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/arXiv.2303.08774 arXiv:2303.08774

[51] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *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*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (Eds.). http://papers.nips.cc/paper_

files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html

[52] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2023. LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. *arXiv preprint arXiv:2308.02828* (2023).

[53] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). https://doi.org/10.48550/ARXIV.2308.12950 arXiv:2308.12950

[54] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html

[55] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[56] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems* 36 (2024).

[57] Qiushi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, et al. 2024. A survey of neural code intelligence: Paradigms, advances and beyond. *arXiv preprint arXiv:2403.14734* (2024).

[58] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, and Thomas Scialom et.al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023). https://doi.org/10.48550/ARXIV.2307.09288 arXiv:2307.09288

[59] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).

[60] Jianxun Wang and Yixiang Chen. 2023. A Review on Code Generation with LLMs: Application and Evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 284–289.

[61] Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2024. TESTEVAL: Benchmarking Large Language Models for Test Case Generation. *arXiv preprint arXiv:2406.04531* (2024).

[62] Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, 8696–8708. https://doi.org/10.18653/V1/2021.EMNLP-MAIN.685

[63] Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. TypeT5: Seq2seq Type Inference using Static Analysis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=4TyNEhI2GdN

[64] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1556–1560.

[65] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. 2024. Enhancing LLM-based Test Generation for Hard-to-Cover Branches via Program Analysis. *arXiv preprint arXiv:2404.04966* (2024).

[66] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2023. White-box compiler fuzzing empowered by large language models. *arXiv preprint arXiv:2310.15991* (2023).

[67] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2023. Kernelgpt: Enhanced kernel fuzzing via large language models. *arXiv preprint arXiv:2401.00563* (2023).

[68] Yuxuan Yao, Han Wu, Zhijiang Guo, Biyan Zhou, Jiahui Gao, Sichun Luo, Hanxu Hou, Xiaojin Fu, and Linqi Song. 2024. Learning From Correctness Without Prompting Makes LLM Efficient Reasoner. arXiv:2403.19094 [cs.CL]

[69] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726.

[70] Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-Edit: Fault-Aware Code Editor for Code Generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 769–787. https://doi.org/10.18653/V1/2023.ACL-LONG.45

[71] Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023. ALGO: Synthesizing Algorithmic Programs with Generated Oracle Verifiers. 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*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.). http://papers.nips.cc/paper_files/paper/2023/hash/abe1eb21ceb046209c96a0f5e7544ccc-Abstract-Conference.html

[72] Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906* (2024).

[73] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406* (2023).

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009