Oracle-Guided Program Selection from Large Language Models

Zhiyu Fan*
National University of Singapore
zhiyufan@comp.nus.edu.sg

Haifeng Ruan*
National University of Singapore
hruan@comp.nus.edu.sg

Sergey Mechtaev
Peking University
mechtaev@gmail.com

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

Abstract

While large language models (LLMs) have shown significant advancements in code generation, their susceptibility to producing incorrect code poses a significant challenge to the adoption of LLM-generated programs. This issue largely stems from the reliance on natural language descriptions as informal oracles in code generation. Current strategies to mitigate this involve selecting the best program from multiple LLM-generated alternatives, judged by criteria like the consistency of their execution results on an LLM-generated test suite. However, this approach has crucial limitations: (1) LLMs often generate redundant tests or tests that cannot distinguish between correct and incorrect solutions, (2) the used consistency criteria, such as the majority vote, fail to foster developer trust due to the absence of transparent rationale behind the made choices. In this work, we propose a new perspective on increasing the quality of LLM-generated code via program selection using the LLM as a test oracle. Our method is based on our experimentally confirmed observation that LLMs serve more effectively as oracles when tasked with selecting the correct output from multiple choices. Leveraging this insight, we first generate distinguishing inputs that capture semantic discrepancies of programs sampled from an LLM, and record outputs produced by the programs on these inputs. An LLM then selects the most likely to be correct output from these, guided by the natural language problem description. We implemented this idea in a tool LLMCodeChoice and evaluated its accuracy in generating and selecting standalone programs. Our experiments demonstrated its effectiveness in improving *pass@1* by 3.6-7% on HumanEval and MBPP benchmarks compared to the state-of-art codeT. Most interestingly, the selected input-output specifications helped us to uncover incompleteness and ambiguities in task descriptions and also identify incorrect ground-truth implementations in the benchmarks.

CCS Concepts

• **Software and its engineering** → **Automatic programming**; **Software testing and debugging**; • **Computing methodologies** → **Natural language processing**.

*Joint first authors.

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3680308

Keywords

large language model, code generation, oracle inference, differential testing

ACM Reference Format:
Zhiyu Fan, Haifeng Ruan, Sergey Mechtaev, and Abhik Roychoudhury. 2024. Oracle-Guided Program Selection from Large Language Models. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680308

1 Introduction

Program synthesis [4, 15–17] aims to automatically generate programs that satisfy developers' intentions, which reduces the manual burden and lowers the barrier to programming. Trained on massive amounts of data, LLMs [11, 24, 28] have propelled the generation of programs from natural language descriptions to an unprecedented level. LLM-based AI programming assistants like Github Copilot, and Amazon CodeWhisperer [1, 31, 42] automatically generate multiple code suggestions based on developers' natural language query, whereas developers navigate between these code suggestions until they find one that best aligns with their objectives. Consequently, the effectiveness of AI programming assistants is contingent upon their ability to promptly propose the most accurate code solutions, thereby fostering a sense of trust in the LLM-generated code suggestions and enhancing the overall user experience.

A key limitation of the reliance on LLMs and natural language specification is the generation of imprecise solutions. Recent studies [14] show that many LLM-generated programs often do not align with the requirements described in natural language. One common approach to mitigating this problem is selecting programs among multiple sampled LLM outputs based on the consistency of these outputs, e.g. via the majority vote [10, 24]. Techniques following this paradigm first group LLM-generated programs into clusters so that the programs in each cluster share the same execution results on a given test suite. Then, they use the size of a program cluster as a proxy for the likelihood of being correct. Specifically, AlphaCode [24] trains a separate LLM for natural language based test-input generation and group programs with the same output together. codeT [10] prompts an LLM to generate a test suite and applies a so-called dual-execution agreement to assign correctness scores for each program cluster. Although these approaches have shown promising improvements over the default random selection strategy in LLMs, there are certain limitations. One of the key limitations is the low quality of the LLM-generated tests. These tests are often incorrect, contain many duplicates, or cannot reveal the discrepancies between various generated programs.

To address the above limitations, we propose a new perspective on increasing the quality of LLM-generated code via program selection using the LLM as a test oracle [8]. Our method is based on our experimentally confirmed observation that LLMs serve more effectively as oracles when tasked with selecting the correct output from multiple choices, rather than predicting the right outcome for a given input from scratch. Relying on this observation, we first find *distinguishing inputs* that capture the semantic discrepancies between LLM-generated programs through differential testing instead of randomly sampling test cases with LLMs. Then, we group programs that produce the same results on distinguishing inputs into program clusters. We also record outputs produced by the programs on these inputs for each cluster. An LLM is then prompted to select the output that is most likely to be correct, guided by the natural language problem description. We also apply chain-of-thought (CoT) prompting [36] to increase the chance of selecting the right output. The selected outputs, together with the distinguishing inputs, form a new specification. Finally, the LLM-generated programs that satisfy this specification are returned to the user.

We conducted experiments on widely-used code generation benchmarks, HumanEval and MBPP [5], focused on the generation of standalone programs, i.e. self-contained programs that invoke only built-in functions and standard libraries [40]. The experiments show that the oracle-guided selection strategy improves the accuracy of program selection compared to the state-of-the-art CODET. Specifically, the *pass@1* increases from 80.8% to 87.7% on the HumanEval benchmark, and from 75.0% to 78.6% on the MBPP benchmark. LLMCODECHOICE is also 60% more efficient than CODET w.r.t. the token cost. In addition, using LLMCODECHOICE's distinguishing input-output specifications, we identified incorrect ground-truth implementations in the benchmarks and showed that the specifications help uncover incompleteness and ambiguities in task descriptions, which may improve developers' confidence in using AI programming assistants in the future.

- We propose the idea of oracle-guided program selection for LLM-generated programs, which finds distinguishing inputs and leverages the natural language descriptions to select the correct outputs of distinguishing inputs to form a high-quality distinguishing test suite as specifications.
- In our experiments on HumanEval+ and MBPP+, we show that our approach, implemented in a tool LLMCODECHOICE, enhances the accuracy of code generation compared with the state-of-the-art program selection techniques.
- We demonstrate that the inferred distinguishing input-output specifications are useful for detecting incompleteness and ambiguities in the problem descriptions.

2 Related Work

Code Generation. Code generation (program synthesis) is the task of automatically generating programs that satisfy some form of user specification. In existing works, the specification can take the form of a logical formula [4] or input-output examples [16, 29]. In recent years, neural networks have been trained to generate programs from a natural language description of the desired program [34, 39]. Most recently, a large number of LLMs have been developed, e.g., CodeX [11], AlphaCode [24], ChatGPT [3], and GPT-4 [28]. They have been pretrained on massive amounts of data and can generalize to various tasks given an appropriate prompt, without fine-tuning. In particular, LLMs go beyond previous techniques by generating programs from natural language descriptions. In our work, we take advantage of ChatGPT to generate initial program samples.

Prompting for Code Generation. A widely used prompting technique for code generation is the few-shot prompting introduced with GPT-3 [9]. In a few-shot prompt, the actual problem to be solved is preceded by several example question-answer pairs. These examples result in improved quality of generated code. Several other prompting techniques have been inspired by few-shot prompting, e.g., chain-of-thought [36] and structured chain-of-thought [23]. In this paper, we study the problem of selecting correct LLM-generated code, instead of generating code. Our selection technique can be applied on top of programs generated with various prompting techniques, to enhance user trust in the generated programs.

Program Selection from LLMs. LLMs are non-deterministic [30], and many generation attempts may result in the generation of incorrect programs. To tackle this issue, several techniques have been proposed to select the correct program from multiple samples. AlphaCode [24] divides programs into clusters based on program output on a set of LLM-generated test inputs. Programs with the same outputs are put in the same cluster, and larger clusters are prioritized for selection. Speculyzer [20] and CodeT [10] perform a similar clustering, where programs passing the same set of LLM-generated tests are clustered. Apart from the cluster size, Speculyzer and CodeT also take into account the number of passed test cases when doing the selection. In our work, we cluster programs by differential testing. This produces distinguishing inputs that better expose differences between programs than LLM-generated test cases. Moreover, instead of selecting the largest cluster, we select the cluster that LLM chooses as an oracle.

Program Differentiation. An important part of program selection is to expose the semantic differences between programs. One way to do so is *differential testing*, which aims to generate concrete inputs on which two programs exhibit different behavior (distinguishing inputs). The inputs can be generated via a biased random search under the guidance of some feedback (e.g., program coverage [13]) or generated by symbolic execution [32]. Besides differential testing, *regression verification* [6, 7] can also be used to expose the inequivalence between programs. In our work, we use differential testing for program differentiation, because the generated distinguishing inputs enable the LLM to be used more effectively as a test oracle.

Test Generation with LLMs. AthenaTest [35] generates unit tests for Java programs with a transformer. It uses the program-under-test as the oracle. ChatUniTest [38] prompts ChatGPT to generate Java unit tests with context information, including method name, class name, etc. ChatTester [41] queries ChatGPT about the "intention" of the method-under-test, which is then used to prompt ChatGPT to generate tests. CodaMosa [21] improves coverage of search-based testing by leveraging an LLM to generate example test cases for under-covered functions. Fuzz4All [37] automatically prompts LLMs to generate fuzzing inputs. Libro [19] prompts an LLM to generate test cases from a bug report for the purpose of bug reproduction. Contrary to these works we do not use LLMs

to generate tests. We use the LLM as partial guidance to infer the intended behavior of test inputs generated via differential testing.

Repair of Code from LLMs. Another possible mitigation to the lack of correctness guarantee is to repair LLM-generated code. The repair process can be aided with different artifacts, e.g., LLM-generated feedback on the code [27], error information produced by code execution [12, 14, 18]. In this paper, we focus on program selection which complements repair of LLM-generated code.

3 Motivating Example

In this section, we illustrate LLMCodeChoice with task 44 of the HumanEval benchmark. We compare our technique with codeT [24], the state-of-the-art program selection technique, as well as AlphaCode [24]. Both codeT and AlphaCode are based on majority vote and can improve the accuracy of program selection. However, they failed to select a correct program for this task, while LLMCodeChoice made a correct selection.

As shown at the top of Figure 1, the "change_base" task involves implementing a function that returns the string representation of the input number x after changing its numerical base to *base*, where *base* is less than 10, and we generated ten program samples with ChatGPT. We observe that six of the programs are equivalent, denoted with "Program Cluster 1" in Figure 1. Samples in program cluster 1 iteratively calculate the digits of x in the specified *base* and then concatenate the digits to get the result string. Meanwhile, the other four samples take another solution, forming "Program Cluster 2" in Figure 1. This solution is largely the same as that of program cluster 1, except that it correctly returns the string '0' in case $x = 0$. Because the representation of the number 0 would still be 0 for any base. On the other hand, since program cluster 1 ignores this edge case, it would return the wrong result of an empty string when $x = 0$. This confusion is likely due to the incompleteness of the natural language specification of the 'change_base' task. The task description in Figure 1 specifically mentions that *base* is less than 10, but leaves the range of the input x unspecified, which leads to two different understandings in the LLM-generated programs. The surprising fact is that the canonical solution of this task provided by experienced programmers in HumanEval also has this mistake, which underlines the importance of providing supporting evidence for LLM-generated programs to developers.

AlphaCode and codeT. AlphaCode and codeT perform majority vote based on program execution results on a set of LLM-generated test cases. For this example, we generated 100 test cases with GPT-4, using the problem description as the prompt. We notice that the test cases include a distinguishing input $x = 0, base = 2$, which would allow AlphaCode and codeT to divide the programs into two clusters. There is also a test case $x = 10, base = 16$, though the problem description specifies that *base* should be less than 10. Both clusters would fail this test due to the unexpected *base* value.

AlphaCode executes all the programs with the LLM-generated test inputs and groups programs that produce the same outputs into one cluster, and then randomly selects one program from the largest program cluster as the correct program sample. In the example of Figure 1, AlphaCode selects a sample from program cluster 1, because this cluster is the larger one. Similar to AlphaCode, codeT tracks the passing tests for each LLM-generated program regarding the LLM-generated test suite, and classifies programs that pass the same tests into one program cluster. codeT then assigns a correctness score defined as (size of a program cluster) × (number of passing tests of a program cluster) to each program cluster and selects programs from the cluster with the highest score. In Figure 1, cluster 1 passes 98 test cases (failing (0,2) and (10,16)) and has size 6, so its correctness score is calculated as $98 \times 6 = 588$. Cluster 2 does not fail the $x = 0$ test case and has a score of $99 \times 4 = 396$. Since cluster 1 has a higher score, codeT also selects program cluster 1 as correct. Therefore, both AlphaCode and codeT made the wrong selection for this example.

Our Approach. Instead of solely relying on LLMs to generate test cases based on incomplete natural language specification, we propose to search for a small yet precise distinguishing test suite. The small size of the test suite makes it affordable to be shown to developers as correctness evidence for selected LLM-generated programs. We illustrate the construction of the distinguishing test suite through the example in Figure 1. We set a postcondition for all program samples from LLM that their outputs must be equivalent for any given input. Taking the postcondition as a guard and leveraging testing techniques, we can easily find one counter-example $x = 0, base = 2$ that violates the postcondition as a distinguishing input and thus produces two program clusters. This counter-example shows that the natural language specification is incomplete for the test input $x = 0, base = 2$. We provide insight into the possible outputs by executing the two program variants of 'change_base' and get two possible specifications.

`change_base(0,2)=="0"` *OR* `change_base(0,2)==""`.

Then we leverage LLM to select one correct output from the two specification options based on the problem description. Experimentally, we show that LLM is more accurate in selecting the output from several options rather than predicting the output from scratch, which we believe explains a better success rate of our approach compared to codeT. Combining the distinguishing input and the selected output, we curate an input-output specification, which not only improves program selection accuracy but may also enhance the trust of developers in LLM-generated code, since input-output examples are easy to interpret for humans.

4 Methodology

LLMCodeChoice accepts a programming task description in natural language, which must satisfy the following assumptions:

- the description must contain information on how the outputs of the program relate to its inputs,
- there must be a method of testing the generated code to find distinguishing inputs and the corresponding outputs,
- it must be possible to represent the input and the output as text that multiple outputs for distinguishing inputs fit into the LLM's context window.

LLMCodeChoice first samples a set S of programs from an LLM (step ① in Figure 2). Then, it performs differential testing to obtain *distinguishing inputs* witnessing behavioral differences among samples (step ② in Figure 2). Specifically, we say an input i is a distinguishing input, if there exists two samples s_1, s_2 such that

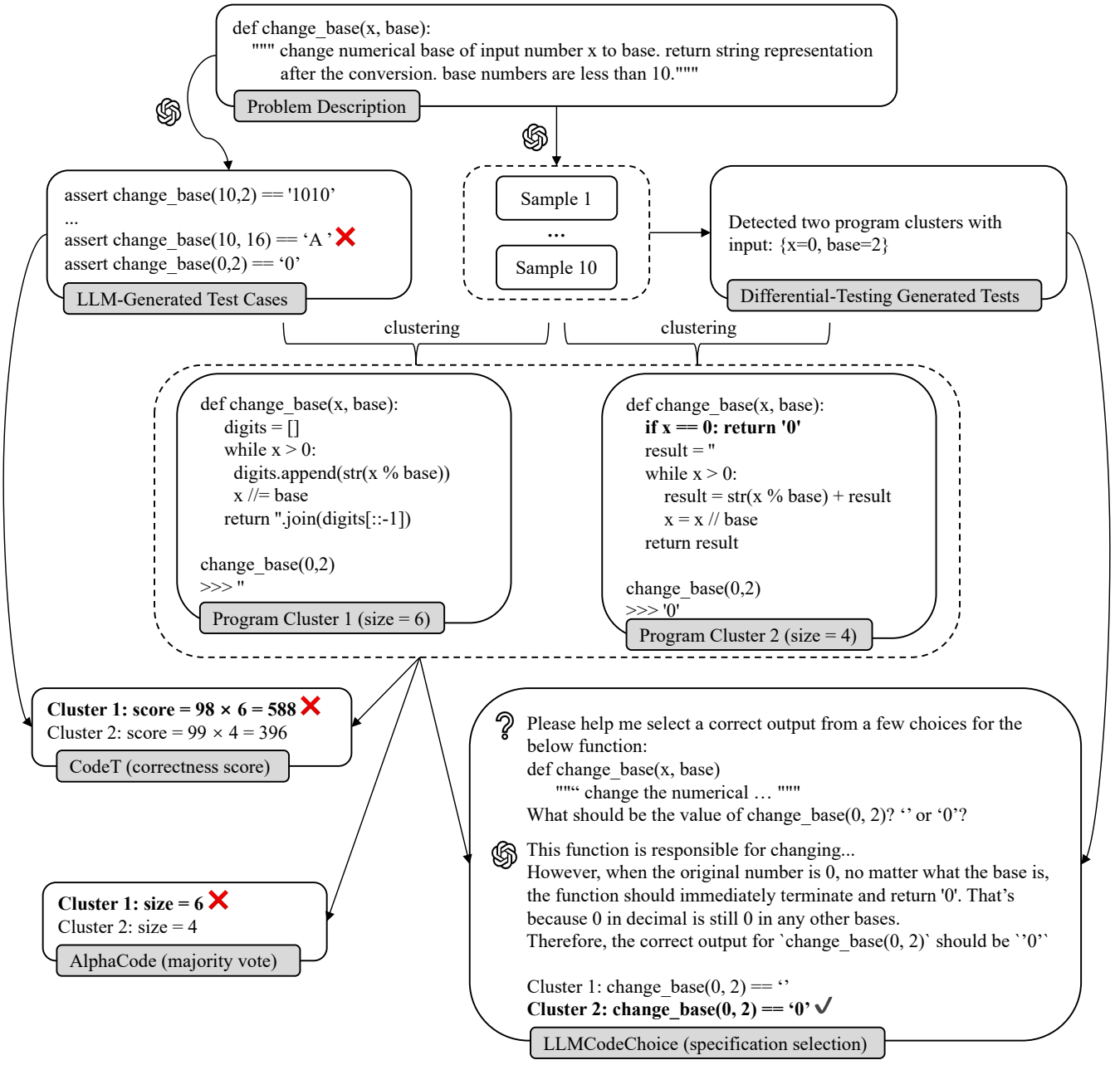

Figure 1: The workflows and selection results of LLMCodeChoice, AlphaCode, and codeT on task 44 of HumanEval. Selection results are marked in bold. While AlphaCode and codeT use LLM-generated tests to do program clustering and some form of majority vote, LLMCodeChoice obtains distinguishing inputs by differential testing and prompts LLM to select the correct outputs for these inputs based on natural language problem description.

$s_1(i) \neq s_2(i)$, where $s_1(i)$ and $s_2(i)$ represent the respective outputs of both samples on i. The set of distinguishing inputs, which we write $\mathcal{I}$, partitions the samples into clusters. Two samples fall into the same cluster if they produce the same output on every input.

With the distinguishing inputs and the program outputs on these inputs, we try to infer an input-output oracle for the problem. For every distinguishing input, we query the LLM about the corresponding correct output and then rule out all clusters incorrect on this input (step ③ in Figure 2). A query takes the form of a multiple choice question, choices consisting of all different outputs of the samples that have not been ruled out. In this way, we are able to

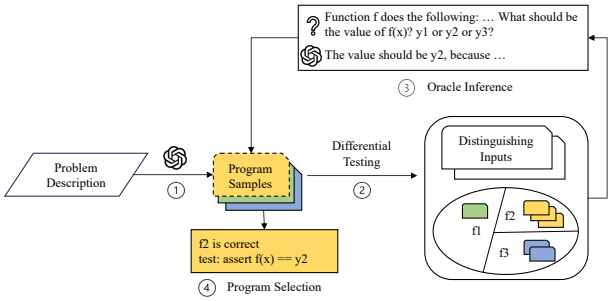

Figure 2: Overview of Oracle-Guided Program Selection

gradually zoom in on and finally select the correct equivalence class (step ④ in Figure 2).

4.1 Differential Testing of Program Samples

We employ differential testing to generate distinguishing inputs. The generated distinguishing inputs would allow us to select the correct program more accurately, with less computational cost, and also facilitate the maintenance of the LLM-generated program in the long run if the program is integrated into a codebase.

The distinguishing input can help increase the accuracy of our program selection. This is because a distinguishing input clearly indicates when two programs would have different behavior. To make a choice between two different programs, it is enough to get the expected output of a distinguishing input and compare the actual outputs of both programs with the expected output. Intuitively, it is often an easier task to find out the one expected output than to compare the full semantics of two programs.

The distinguishing input also makes program selection more cost-effective, as the number of distinguishing inputs is often small. As shown in our experiment, ten LLM-generated programs on average represent no more than three kinds of different semantics, so two distinguishing inputs would be enough to differentiate the programs. Therefore, only a modest number of tokens need to be used to predict the expected outputs of the inputs using an LLM. In comparison, CODET requires sampling many test cases from the LLM and thus incurs high computational cost.

An additional benefit of distinguishing inputs is that they facilitate the understanding and maintenance of the generated program. The inputs, together with their corresponding outputs, effectively form a test suite that specifies the desired program behavior. In case the generated program is integrated into a codebase, this test suite can be integrated for quality assurance as well. We will further illustrate the usefulness of the distinguishing inputs with concrete examples in a case study (Section 6.5), where these inputs have helped identify ambiguous descriptions or even wrong ground truths in the evaluation datasets.

These benefits motivate us to aid program selection with distinguishing inputs. There is more than one way to generate these inputs. An obvious option is to use the LLM to generate test cases [10]. However, as discussed in the example in Section 3, distinguishing inputs tend to be rare in LLM-generated test cases. Also, the queries to LLM would incur significant computational costs. In light of these observations, we have opted to generate distinguishing inputs using differential testing. Specifically, we employ two test generation techniques, fuzzing and symbolic execution for differential testing. To differentiate two program samples, we search for an input violating an assertion we make that the program samples yield the same output on every input. First, we use a property-based fuzz testing tool to find tests that produce different outputs in the two programs. This is done by differential fuzzing the two programs $P1, P2$ where for any input we check the outputs in $P1, P2$. If fuzzing fails to produce a distinguishing input within a time bound T, we employ a more systematic symbolic execution tool to find distinguishing inputs within the same time budget T. The code for both programs $P1, P2$ are renamed apart from input variables[1] and composed into a new program $Q_{P1,P2}$ which also contains the assertion $\varphi_{P1,P2}$ about equivalence of outputs of $P1, P2$

$$\varphi_{P1,P2} \equiv \text{out}(P1) == \text{out}(P2)$$

where $\text{out}(P1), \text{out}(P2)$ denotes the output variable for $P1, P2$ respectively. The symbolic execution engine analyzes $Q_{P1,P2}$ with the goal of finding an input violating $\varphi_{P1,P2}$.

We show our program selection algorithm in Algorithm 1, where lines 1—5 correspond to the differential testing process. The process of differentiating two samples is represented with the procedure **Diff Test** in line 4, which returns a singleton set of distinguishing inputs for two samples (or an empty set if not found). To distinguish between more than two samples, we perform this pairwise differentiation for the multiple pairs made up from the samples (lines 3–5 of Algorithm 1). Although one can also differentiate all samples at once by making an assertion about all of them, pairwise comparison makes LLMCODECHOICE compatible with any differential testing technique. The process of differential testing across samples produces a set of distinguishing program inputs. These distinguishing inputs effectively define an equivalence relation over the samples, where two samples are equivalent if they produce the same output on all the distinguishing inputs. We denote the equivalence relation with $\sim$ in line 5 of Algorithm 1; the resulting set of program clusters is the quotient set $C = S/\sim$. We then conduct oracle inference to select the correct program cluster from C.

4.2 Specification Selection with LLM

In this section, we explain the key specification selection component of LLMCODECHOICE, which uses the LLM to select the correct program specification for distinguishing inputs found in Section 4.1.

Given the distinguishing inputs and corresponding outputs from program clusters, our goal is to select the correct output for all distinguishing inputs and form the input-output program specification, thereby selecting the correct program cluster. Algorithm 1 details the specification selection and cluster selection process. LLMCODECHOICE first selects the distinguishing input i having the highest entropy (line 6). The entropy of a distinguishing input i is defined as $-\sum_k \frac{n_k}{N} \log \frac{n_k}{N}$, where n_k is the size of the k-th program cluster, and N is the total number of programs. The intuition is that by using the input-output specification of a high-entropy distinguishing input to filter the programs, more programs can be ruled out with a

[1]The only common variables across the two programs $P1, P2$ are the input variables, the rest are renamed apart.

Algorithm 1: Oracle-Guided Program Sample Selection

Input: Task description *task*, LLM $\mathcal{L}$, program samples S
Output: One selected cluster of program samples

1 $\mathcal{I} \leftarrow \{\ \}$
2 **for** (s_1, s_2) in $S \times S$ **do**
3 **if** $\neg\exists\, i \in \mathcal{I}.\, s_1(i) \neq s_2(i)$ **then**
4 $\mathcal{I} \leftarrow \mathcal{I} \cup$ **diffTest**(s_1, s_2)
5 $C \leftarrow S/\sim$, where $s_1 \sim s_2 \iff \forall\, i \in \mathcal{I}.\, s_1(i) = s_2(i)$
6 **for** i in $\mathcal{I}$ **do**
7 $outputs \leftarrow \{\, c(i) \mid c \in C \,\}$
8 $o \leftarrow$ **oracleInference**$(\mathcal{L}, task, i, outputs)$
9 $C \leftarrow \{\, c \in C \mid c(i) = o \,\}$
10 **if** $|C| = 1$ **then**
11 **break**
12 **return** the only remaining cluster in C

```
# Specification selection as a multi-choice question
Please select a correct output from a few choices for
below function.
# function signature and problem description
def foo(args):
  """problem description"""
# multi-choice to select correct output
What should be the value of `foo(i)`? `output 1` or `
output 2`?
# Explain the choice
Please explain step by step and answer.
```

Figure 3: Prompt for Specification Selection

single input, so that the number of queries made to the LLM can be minimized. For the selected input i, LLMCodeChoice then collects the different outputs $c(i)$ of each program cluster c (line 7), which have been recorded during differential testing. With the output options, we invoke the LLM to select the correct output (line 8) with the specification selection prompt shown in Figure 3. The prompt has three parts. First, the prompt frames the specification selection task for LLM as a multiple-choice question. We provide the function signature and problem description for the LLM to have an initial understanding of the programming problem. Second, the prompt takes a distinguishing input and asks LLM to select the correct answer from the recorded output options. Our intuition is that the provided output options prune the search space of LLM, so selecting a correct output from multiple choices should be more effective than predicting the right execution result for a given input from scratch. This intuition was confirmed by our experiment. Moreover, we also integrate the popular chain-of-thought prompting [36] which has been shown effective in improving LLM's accuracy through a detailed reasoning step, by adding an "explain step by step" request at the end of the prompt. This step enables the LLM to explain the possible reasons that produce each output for a distinguishing input before the LLM selects answers. Afterward, we keep only the clusters that produce the expected output and filter out the other clusters (line 9). We iteratively perform the specification selection and cluster filtering with each distinguishing input, until only one cluster satisfying all the selected input-output specifications is left (lines 10—11). This one cluster is then considered the most likely to be correct and proposed to the user (line 12).

5 Experiment Setup

We implement the idea of oracle-guided program selection in a tool LLMCodeChoice and demonstrate its effectiveness by addressing the following research questions:

RQ1: How effective are distinguishing inputs in finding semantic discrepancies of LLM-generated programs?
RQ2: How effective is specification selection in LLMCodeChoice?
RQ3: How effective is LLMCodeChoice in the overall program selection?
RQ4: What are the reasons for failure of LLMCodeChoice?

In RQ1, we first compare the effectiveness of distinguishing inputs generated by LLMCodeChoice against the LLM-based test generation used in AlphaCode and codeT on HumanEval and MBPP subjects. In RQ2, we evaluate LLMCodeChoice's specification selection accuracy for the generated distinguishing inputs to show the effectiveness of inferring specification with LLM. We then demonstrate the overall improvement of pass@1 (probability of LLM-generated program being correct with only one sample) by applying LLMCodeChoice in selecting correct LLM-generated programs on HumanEval and MBPP benchmarks in RQ3. Finally, we discuss the reasons for incorrect specification selection in our method through more detailed analysis (RQ4). Additionally, we discuss the use of our input-output specifications in enhancing the trust of developers in LLM-generated programs (Section 6.5).

Benchmarks. We evaluate LLMCodeChoice on two widely-used code generation benchmarks HumanEval [11] and MBPP [5]. HumanEval and MBPP are manually curated datasets that consist of 164 and 399 basic Python programming tasks. We removed the example input-output test cases in the problem descriptions of HumanEval, because the examples overlap with some test cases in the validation test suite. We modified the prompt of MBPP to the same format as HumanEval. Each prompt consists of a function signature and a natural language problem description in the docstring. However, a recent study [25] shows that the validation test suite of HumanEval and MBPP benchmarks are relatively weak. This may cause the acceptance of plausible LLM-generated programs, which pass the original test suite but are actually incorrect. To mitigate such risks, we also evaluate LLMCodeChoice on the enhanced versions of HumanEval+ and MBPP+ [25] equipped with a much more comprehensive test suite.

Implementation. We use ChatGPT (snapshot gpt-3.5-turbo-0613) to generate solutions for all tasks in HumanEval and MBPP benchmarks. Specifically, we set the `max_token` parameter to 1024 which provides sufficient space for ChatGPT to complete all tasks in benchmarks, and we keep other parameters as default (e.g., temperature=1). Our differential testing engine (refer to Section 4.1) is built upon two Python automated testing frameworks Hypothesis [26] and CrossHair [2]. Hypothesis is a property-based testing tool using a random testing strategy to find counterexamples for post-conditions, whereas CrossHair attempts to find counterexamples by exploring feasible execution paths with symbolic inputs. We set

a 3-minute time bound for both tools. For the specification selection step, we use the GPT-4 (snapshot gpt-4-0613) model from OpenAI.

Metrics and Baselines. We use the pass@1 metric [11] to evaluate program selection accuracy. For each task, we use LLMCodeChoice to select one program cluster from all program samples. We then randomly select one program from the selected program cluster. The pass@1 of LLMCodeChoice on this task is the probability that the selected program passes all test cases in the benchmark. Suppose the selected cluster contains N programs, of which n programs pass all test cases, the pass@1 is then calculated as n/N. For both benchmarks, we report the average pass@1 across all tasks. We compare LLMCodeChoice against four baselines, (1) random selection which randomly selects one program from all LLM-generated programs, (2) AlphaCode [24], (3) AlphaCode with LLM-based specification inference, and (4) codeT [10]. The pass@1 of the baselines are calculated similarly. Since Codex (code-davinci-002) model used in the original codeT has been deprecated by OpenAI, we replicated the test case generation process in AlphaCode and codeT with the most powerful GPT-4 model by reusing codeT's artifact and reported the latest result to ensure a fair comparison.

6 Evaluation

In this section, we first separately demonstrate the effectiveness of both components of LLMCodeChoice. We then show our improvement over the baselines in the overall selection accuracy. Finally, we discuss the limitations and unique advantages of LLMCodeChoice.

6.1 RQ1: Effectiveness of Test Generation

We measure the effectiveness of our test-generation component with the number of different clusters it can identify from the samples. The number of test inputs generated is also measured. It is desirable that the component can partition the samples into more clusters with fewer distinguishing inputs. This would mean that each input is of higher quality, and would make it easier for a human to examine these inputs and the resulting clusters.

We compare these two metrics of LLMCodeChoice with AlphaCode and codeT. Note that, although AlphaCode and codeT use the same set of LLM-generated tests, they can produce different clusters due to their different clustering strategies (see Section 3). Moreover, we compare with the "ground truth" clusters obtained by executing the validation test suites in HumanEval+ and MBPP+. Since the validation test suites contain a large number of carefully generated tests, one would expect them to find the most clusters.

Table 1 shows the metrics of LLMCodeChoice and the baselines. It can be seen that LLMCodeChoice outperforms all baselines on the MBPP benchmark. In particular, LLMCodeChoice finds 24.3% more program clusters than codeT. On HumanEval+, the average number of program clusters identified by all tools is comparable. LLMCodeChoice performs slightly better than codeT, but slightly worse than AlphaCode. Specifically, for 23 of the 164 tasks, LLMCodeChoice finds fewer clusters than AlphaCode. We have manually investigated these tasks and found that these tasks have strict input constraints. For example, task 19 requires the input string to be space-delimited numerals ("zero" to "nine"). Such constraints are difficult for our differential testing engines based on random mutation and symbolic execution. However, there are also 21 tasks for which LLMCodeChoice found more clusters than AlphaCode. This implies that the LLM and traditional differential testing techniques can complement each other, e.g., seeding random mutation with an LLM-generated input. We leave the exploration of combining both techniques to future work.

Table 1: Average number of identified program clusters and average number of generated test cases of each tool on HumanEval+ and MBPP+.

	HumanEval+		MBPP+	
	avg clusters	avg tests	avg clusters	avg tests
Ground Truth	2.54	740.9	2.18	108.6
AlphaCode	2.56	89.2	2.56	85.3
codeT	2.31	89.2	2.11	85.3
LLMCodeChoice	2.48	1.2	2.71	1.3

A noteworthy special situation during the testing is when no distinguishing input can be found. In this case, even if two programs actually have different semantics, LLMCodeChoice would place them in the same cluster and select them with equal probability. In the 563 tasks in both benchmarks (164 from HumanEval+ and 399 from MBPP+), the "ground truth" identifies 251 tasks as having semantically different programs, but LLMCodeChoice failed to find a distinguishing input for 35 of these. On the other hand, among the other 312 (=563-251) tasks whose programs are all deemed equivalent by the "ground truth", LLMCodeChoice managed to find a distinguishing input for as many as 119 tasks. This finding indicates that even a carefully constructed test suite may fail to differentiate semantically different samples.

With regard to the number of test cases, LLMCodeChoice uses no more than 1.3 test cases for each task, which is about 60x fewer than AlphaCode and codeT. With fewer test cases, LLMCodeChoice is able to identify a similar or larger number of program clusters, highlighting the usefulness of our test cases. The small number of distinguishing inputs allows them to be presented to developers as evidence before developers accept LLM-generated code, which we discuss further in Section 6.5.

6.2 RQ2: Effectiveness of Specification Selection

Specification selection is a key component of LLMCodeChoice, deciding whether the correct program cluster can be chosen. We measure its effectiveness by the frequency with which it makes a correct choice throughout our experiment. In total, our experiment includes 448 specification selection attempts made on 327 tasks of HumanEval and MBPP. Note that for the other 236 tasks from both benchmarks, as LLMCodeChoice has identified only one program cluster, there is no selection to be made. From the 448 selections, we count out the ones without a correct option. The correct option can be missing, either because all program samples are wrong, or because the correct programs have been excluded from the options due to a previous wrong selection. This leaves 381 selections with the correct option available, and in these, we count the number of times the correct option is actually chosen. The correct options have been obtained by executing the ground-truth program from the benchmarks on the inputs.

Table 2: Accuracy of Oracle Inference Strategies: specification selection via LLMCodeChoice, specification inference from scratch from LLM, and majority vote.

	LLMCodeChoice	Direct Inference	Majority Vote
HumanEval+	87.6%	77.0%	71.3%
MBPP+	82.7%	68.2%	66.8%
Total	84.3%	70.3%	68.2%

We compare our specification selection with two baseline selection strategies, by applying the strategies to the test inputs involved in the above-mentioned 381 selections. One baseline is a majority vote, i.e., selecting the option that is the output of the majority of the samples. This is similar to AlphaCode, but AlphaCode takes all inputs into account to identify the majority, while here we only consider a single input at a time. The other baseline, denoted with direct inference, is where we remove the options from the prompts of LLMCodeChoice and use the LLM to predict the output directly. By comparing with direct inference, we demonstrate the benefit of hinting the LLM with program outputs.

We present the result in Table 2. As shown in the table, LLMCodeChoice has selected the correct output in 321 (84.3%) of the 387 attempts, representing the highest accuracy among all the strategies. This is 14.0% higher than direct inference, highlighting the benefit of providing options for the LLM. The majority vote has an even lower accuracy than direct inference. The fact that the majority is often incorrect shows the necessity of oracle inference beyond a simple majority vote.

We further investigate the reason LLMCodeChoice outperforms direct inference. It turns out that the LLM struggles to handle complex inputs, especially in math problems, while including possible options in the prompt mitigates this problem. For example, direct inference did not correctly solve the problem in Figure 4, while LLMCodeChoice managed to do so.

Please help me find the correct output for the below function.
Please help me select a correct output from a few choices for the below function
def largest_prime_factor(n: int):
"""Return the largest prime factor of n. Assume n > 1 and is not a prime."""
What should be the value of `largest_prime_factor(25191)`? `311` or `933`?

Please explain step by step and answer.

----------Response of Direct Specification Inference ----------
Let's apply this to `largest_prime_factor(25191)`:
25191 is not divisible by 2
25191 is divisible by 3 --> 8397 remains (25191/3 = 8397)
8397 is not divisible by any of the prime numbers 2, 3, 5, 7, 11...
So, the output should be: `8397`

---------- Response of Specification Selection ----------
The process to find out the largest prime factor of a number `n` is as follows:
1. ... 2. ... 3. ...
Consider `n=25191`, ... will find that the prime factors of 25191 are 3, and 311.
Among them, 311 is the largest prime factor.
So, the function `largest_prime_factor(25191)` should return `311`.

Figure 4: Comparison between direct specification inference (marked in green) and specification selection (marked in blue) on an example from HumanEval.

Table 3: Overall pass@1 (%) and average token spent on the HumanEval+, HumanEval, MBPP+, and MBPP benchmarks with GPT-4 for all tools, excluding all tasks that cannot make correct program selection. The numbers $X/(Y)$ in the pass@1 (%) columns represent the pass@1 result for enhanced / (original) benchmarks respectively.

	HumanEval+ (HumanEval)		MBPP+ (MBPP)	
	pass@1 (%)	avg tokens	pass@1 (%)	avg tokens
Random Selection	57.0 (63.1)	-	58.1 (69.9)	-
AlphaCode	75.6 (80.3)	6319	71.4 (81.6)	5507
codeT	80.8 (86.8)	6319	75.0 (83.3)	5507
LLMCodeChoice-Vote	73.1 (84.7)	-	72.0 (81.2)	-
LLMCodeChoice-Infer	79.6 (82.5)	-	76.9 (81.9)	-
LLMCodeChoice	87.7 (89.2)	1533	78.6 (83.2)	1331

6.3 RQ3: Overall Effectiveness

We have evaluated the two components of LLMCodeChoice in RQ1 and RQ2 separately. In this section, we focus on the overall program selection accuracy of LLMCodeChoice, measured by the pass@1 metric. This pass rate has been calculated both on HumanEval and MBPP, and on their enhanced versions HumanEval+ and MBPP+. We count out the tasks for which the LLM-generated programs are all correct or all wrong, in which case the program selection technique makes no difference. This leaves 89 tasks in HumanEval and 162 tasks in MBPP. Apart from pass@1, we also count the number of LLM tokens used and measure the time usage to show the cost-effectiveness of different program selection techniques. The baselines we compare against are codeT and AlphaCode. We present the results in Table 3. Note that in the two pass@1 columns, the results are of the form X/(Y), where X represents the pass@1 on the enhanced benchmarks and Y represents the pass@1 on the original benchmarks.

Pass@1. As shown in Table 3, LLMCodeChoice achieves pass@1 of 87.7% and 78.6% on HumanEval+ and MBPP+ respectively, making it the most accurate of the techniques. This is followed by codeT, and AlphaCode has the lowest accuracy. We further analyzed the reason AlphaCode and codeT are not as accurate as LLMCodeChoice. For AlphaCode, this is due to the limited ability of the LLM to generate correct programs. Specifically, the majority of the LLM-generated programs is often incorrect. The quality of the initial program samples also has an impact on the "dual-agreement selection" [10] of codeT. This stresses the importance of a better program selection strategy beyond majority vote. With regard to codeT, an equally important factor is the quality of the LLM-generated test cases. To evaluate the correctness of these tests, we execute the test cases with the ground-truth programs provided in the HumanEval+ and MBPP+ benchmarks. If a ground-truth program fails a test case, this indicates that the test case is wrong. The execution results show that on both benchmarks, over 80% of the tasks have at least one wrong test case. On average, each task has more than 23 wrong test cases. Remember from Table 1 that there are about 85 LLM-generated test cases for each task, which means nearly 25% of the LLM-generated test cases are wrong. This shows that the LLM is prone to generate incorrect test cases, which would

impact the accuracy of the dual-agreement selection of CODET. Different from CODET, LLMCODECHOICE obtains the expected output of the test inputs by specification selection.

Ablation Study. We identify the respective contribution of both components in LLMCODECHOICE through an ablation study. Specifically, we evaluate the following two variants of LLMCODECHOICE:

- LLMCODECHOICE-Vote: selecting the largest program cluster.
- LLMCODECHOICE-Infer: replacing specification selection with direct inference, i.e., removing the output options from the prompt.

We can see that partitioning the programs into clusters brings an accuracy gain of 16.1% on HumanEval+ and 13.9% on MBPP+. On top of the partitioning, comparing LLMCODECHOICE-Infer and LLMCODECHOICE-Vote indicates that specification inference increases the accuracy by 6.5% and 4.9%. Finally, further providing the LLM with possible outputs increases the accuracy by 8.1% and 1.7% on the benchmarks.

Trustworthiness. Overfitting is a common challenge in many research areas, such as machine learning and automated program repair. It happens when a neural network or a program performs well on a small number of test cases but fails to generalize to more tests. A similar case can happen to LLM-generated programs when the developer only has time to perform a few tests before accepting an actually incorrect program. To evaluate the extent to which program selection techniques can mitigate this risk, we simulate the developer test cases with the original HumanEval and MBPP benchmarks. We say an LLM-generated program is overfitting if it passes the original benchmark but fails the enhanced benchmark. For example, according to Table 3, ALPHACODE has a pass@1 of 80.3% on HumanEval+ and 75.6% on HumanEval, indicating that ALPHACODE selected an overfitting program for 4.7% (80.3%-75.6%) of the tasks. In contrast, this is only 1.5% (89.2%-87.7%) for LLMCODECHOICE, showing that LLMCODECHOICE is less prone to the overfitting problem. LLMCODECHOICE is also the least overfitting on MBPP. This means that the developer can have greater trust for a program selected by LLMCODECHOICE when only limited resources are available for validating program correctness.

Token Consumption. To apply the program selection techniques in practice, it is also important to understand their computational cost. In particular, we count the number of LLM tokens each technique consumes in the selection process. For LLMCODECHOICE, the tokens are used to perform specification selection, while CODET consumes tokens to generate test cases. As shown in Table 3, LLMCODECHOICE has consumed 4x fewer tokens per task on average than CODET, while still achieving a higher selection accuracy. This is because LLMCODECHOICE produces only a few distinguishing inputs, on which specification selection is performed. On the other hand, to perform dual-agreement selection [10], CODET has to sample a large number of tests to get a good estimate of the majority.

Time usage. Across the tasks in both datasets, the median time usage of LLMCodeChoice is 221.5 seconds, and that of AlphaCode and CodeT is 142.1 seconds.

Detection of Wrong Programs. When all programs generated for a task are wrong, the pass@1 would always be zero, regardless of the selection technique. Therefore, such tasks have been counted out of the pass@1 evaluation. However, LLMCODECHOICE may identify such a case when the LLM chooses none of the provided output options. Overall, there are 28 such tasks in HumanEval+ and 90 in MBPP+, and LLMCODECHOICE could detect 30 of these cases.

Table 4: Four Reasons for Incorrect Specification Selection in LLMCODECHOICE.

	Wrong Choice	Incomplete Problem	Ambiguous Problem	Wrong Ground Truth
HumanEval+	6	5	2	2
MBPP+	15	17	2	11
Total	(35.0%) 21	(36.7%) 22	(6.7%) 4	(21.6%) 13

Table 5: Causes of the 21 wrong choices by LLMCODECHOICE.

Type of mistakes	Occurences
Incorrect calculation	11
Misunderstanding of problem	5
Hallucinated precondition	3
Incorrect reasoning	1
Use of incorrect fact	1

Summary. Overall, our experiment shows that LLMCODECHOICE improves pass@1 over random selections by 6.7% to 16.6% and outperforms all baselines. LLMCODECHOICE also consumes significantly fewer computation resources (number of tokens) compared to other baselines, indicating its potential for deployment in real-world development processes.

6.4 RQ4: Reasons For Failure

In this section, we investigate the causes of incorrect specification selection made by LLMCODECHOICE. While LLMCODECHOICE achieves high accuracy in specification selection, it failed to make the correct choice in 60 of the 381 selection attempts throughout our experiment. We divide the failure reasons of these attempts into four categories in Table 4. Two authors made the categorization independently and discussed their differences with a non-author PhD student from the department to reach an agreement.

Wrong Choice. This means LLMCODECHOICE selects an incorrect output from all possible output options for a distinguishing input, when the expected behavior is clearly defined by the problem description. This category accounts for 35.0% of the incorrect selections in our experiment. Since this category of failures is particularly important for understanding the application scope of LLMCODECHOICE, we inspected the 21 mistakes in this category and further divided them into 5 types, as shown in Table 5.

In Table 5, *incorrect calculation* is the prevailing type of error, representing 11 wrong choices. This error type means the LLM takes the right steps to reason about the expected output, but the concrete result of a certain step is wrong. The 11 errors can be further divided into 8 arithmetic errors, 2 string manipulation errors, and 1 logical error. As an example of arithmetic error, task 782 of MBPP calculates the sum of the odd-length subarrays of an array. When given a distinguishing input [0,-1], the LLM made a mistake by stating that "the possible odd-length subarrays are [0], [-1], and [0, -1]". It is

a known issue that LLMs can often make mistakes in calculation, even when specially trained on massive mathematical content [22]. This suggests that an LLM might be less effective as an oracle when the program requirements involve complex arithmetic calculations.

The second most common type of error is *misunderstanding of problem*, roughly accounting for 1/4 of the wrong choices. For example, task 454 of MBPP involves telling whether a string contains 'z'. The quotes around the letter z mean an exact match is required (i.e., case-sensitive). However, the LLM overlooked this detail and wrongly stated that the string 'Z' also contains 'z'. This type of error suggests that the accuracy of LLMCodeChoice can be negatively impacted when the requirement is not clear or explicit enough.

The third largest error type is *hallucinated precondition*, accounting for 3 wrong choices. In this error type, the LLM "rejects" an input because the input does not satisfy a certain precondition hallucinated by the LLM. For example, task 803 of MBPP tells whether an integer is a perfect square. When given the input -1, the LLM does not respond with the correct answer "False". Instead, the LLM hallucinated that the input must be non-negative and stated that an exception should be thrown for -1, since "it is mathematically impossible for negative numbers to be perfect squares." While this type of error can also reduce the accuracy of selection, we note that a user of LLMCodeChoice can easily identify such an error by reading the LLM's response, if the selected output is an exception.

Finally, there are another two less common error types. *Incorrect reasoning* is related to a complex math problem that the LLM failed to reason about, while *use of incorrect fact* occurred when the LLM used a mistaken formula to solve a problem.

Incomplete Problem. Incomplete problem represents a situation where the task description does not explicitly specify correct execution behaviors triggered by distinguishing inputs or lack of definition for specific terms. These distinguishing inputs are often edge cases for the programming tasks, which accounts for 36.7% incorrect specification selection. For example, (1) how to handle empty strings, (2) what exceptions to throw when the program crashes, etc. Incomplete problems are especially common in MBPP, because of their vague descriptions. For example, task 264 in MBPP asks to implement a function to calculate a dog's age in dog's years, without specifying the definition of dog's year. However, we find that the original online description for task 264 clearly explained the relation between dog's year and human's year [2].

Ambiguous Problem. We categorize an incorrect specification selection as an ambiguous problem if two authors independently propose more than one interpretation of the natural language description and LLMCodeChoice selects the output that is different from the interpretation of ground truth implementation, it accounts for 6.7% of incorrect specification selection. Figure 5 shows one example from HumanEval benchmark. The "triples_sum_to_zero" task checks if there are three distinct elements in the list sum to zero. There are two interpretations of the task because *distinct elements* can either refer to elements of different values, or elements of different indices. Both of the two interpretations are correct, but the ambiguity confused LLMCodeChoice and led to incorrect output when given triples_sum_to_zero[0,0,0] as the distinguishing input.

```
1  # Task 40
2  def triples_sum_to_zero(l: list):
3    """ takes a list of integers as an input. it returns True if there are three
           distinct elements in the list that sum to zero, and False otherwise.
4    """
5  - if len(l) < 3 or len(set(l)) < 3:
6  -   return False
7    for i in range(n-2):
8      for j in range(i+1, n-1):
9        for k in range(j+1, n):
10         if l[i]+l[j]+l[k]==0:
11           return True
12   return False
```

Figure 5: Example of ambiguous task description from HumanEval benchmark. The ambiguous point is the interpretation of "distinct elements". Lines 6-7 show the specific implementation of the misinterpreted concept.

Wrong Ground Truth. We have also identified 13 cases where we believe the specification selection is correct and the ground truth program from the benchmarks is wrong, which accounts for 21.6% of the incorrect specification selections. For example, task 799 in MBPP asks to "Write a function def left_rotate(n, d) to that rotate left bits by d bits a given number. We assume that the number is 32 bit". For input (n=16, d=28), the correct output should be '1', because, in 32 bits representation, there are 27 leading zeros before the number 16 (10000), therefore the 28 times left shifts should move the 1 to the rightmost bit, which represents 1 in decimal. However, the ground truth solution produces '4294967297', which represents a 33-bit binary number. This shows the practical value of LLMCodeChoice in witnessing developers' mistakes. By contemplating the input and the output proposed by LLMCodeChoice, the developer has a greater chance to be alerted and avoid such errors. We share our explanations for each wrong ground truth in our artifact.

6.5 Discussion of Selected Specification

We have shown several limitations of the specification selection of LLMCodeChoice in RQ4. While most existing works focus on improving the pass@k result of LLM-generated programs, we propose the perspective that *providing explainable evidence to developers is more important than unexplained pass@k.* In this section, we discuss how we can utilize the byproduct of input-output specification from LLMCodeChoice to improve developers' confidence through a case study of tasks categorized as Ambiguity and Incomplete Specification in RQ4.

Contradictory Explanations. Similar to oracle-guided program synthesis [17], one advantage of LLMCodeChoice is the small-sized distinguishing input-output specification. We propose a preliminary idea called contradictory explanation generation to generate meaningful feedback for developers before they accept any suggestion from AI programming assistants. We obtain the contradictory explanation in two phases, explanation derivation and contradictory feedback. In the explanation derivation phase, for a distinguishing input i and the output o of each program cluster, we prompt the LLM with "the task returns o on input i because..", so that the LLM completes the sentence with an explanation of the output. Explanations of the different outputs may contradict each other. In the contradictory feedback phase, we put the task

[2] https://www.w3resource.com/python-exercises/python-conditional-exercise-31.php

Q: The triples_sum_to_zero([0,0,0]) returns False because ...
A: The elements are not distinct. They are all equal to 0.

Q: The triples_sum_to_zero([0,0,0]) returns True because ...
A: The three zeros in the list sum up to zero.

Q: I have two explanations for triples_to_sum_to_zero, are there any ambiguous or other problems?
A: The explanations are contradictory, hence there is an ambiguity in the understanding of what constitutes "distinct elements" in the context of this problem.

Figure 6: Example of contradictory explanation process for 'triples_sum_to_zero'. We first use the LLM to derive an explanation for each input-output pair. Then we take those explanations and query LLM again to identify the reasons for producing different explanations.

description and the explanations in one prompt, prompting the LLM to identify the reason that the task description allows contradictory explanations. Figure 6 shows task 40 of HumanEval as an example, where the LLM correctly explains the ambiguity of "distinct element" from the two contradicted explanations.

Evaluation. We implemented the contradictory explanation idea and conducted a preliminary evaluation on the tasks where LLMCodeChoice selected incorrect specifications. We again asked the volunteer PhD student who helped categorize reasons for incorrect specification selections as an external developer to evaluate the contradictory feedback generated by LLM. If contradictory feedback points out ambiguity or incompleteness in a problem description, and the volunteer thinks the feedback correctly explained the problem, we mark the task as resolved. In total, we resolved 14 of the 26 tasks with incomplete specification or ambiguity, which shows the potential of applying the input-output specification as a trustworthy guard for future automatic programming. In future work, we will investigate how to create a more comprehensive technique to produce feedback for code from LLM.

7 Threats to Validity

Internal Threats. A threat to internal validity is that our differential testing engine cannot guarantee to find all distinguishing inputs. There might exist distinguishing input that our differential testing engine fails to identify. Thus, the program clusters computed are not guaranteed to be fully accurate. Another threat is that our experiment compares LLMCodeChoice with AlphaCode and codeT. However, the OpenAI Codex (code-davinci-002) model used by codeT has been deprecated. To address this, we replaced the Codex model with GPT-4 model in the experiment for LLMCodeChoice and codeT and we replicated codeT's result using its artifact. While GPT-4 is one of the most powerful LLMs as of now, we believe this replacement will also bring benefit to codeT.

External Threats. The external threat may arise regarding the LLM we selected. Although there are many other open-source large language models [24, 33], our experiment only evaluated LLMCodeChoice using GPT-4, because it is the most representative LLM we can access. To mitigate this threat, we release LLMCodeChoice's implementation to facilitate more detailed evaluation and future development in the community. Nevertheless, our experiments show the effectiveness and potential of oracle inference with LLM.

Another external threat is the type of programs LLMCodeChoice can be applied to. In our experiments, LLMCodeChoice was only applied to standalone (self-contained) programs. If the task if to generate a non-standalone program, e.g., when the program is a complex project consisting of multiple components, or when the program is a library consisting of different utility functions, the application of LLMCodeChoice may require adjustments to reflect the particularities of the application domain. We leave investigating the selection of non-standalone programs for future work.

Finally, in terms of the modality of program output, LLMCodeChoice currently deals only with output that can be represented in text. LLMCodeChoice may not be applied directly when the output has a different modality, e.g., when the program plots an image. To mitigate this threat, a multimodal LLM can potentially be used, and LLMCodeChoice can be adapted to compare program outputs of other modalities.

8 Conclusion

In this paper, we propose LLMCodeChoice to improve the trustworthiness of LLM-generated programs with a new perspective of retrieving program specifications from natural language descriptions. LLMCodeChoice curates distinguishing inputs to reveal differences between LLM-generated programs and further utilizes LLM to select the correct output for those distinguishing inputs from all possibilities. Our evaluation on the generation of standalone programs within HumanEval and MBPP demonstrates its effectiveness. We conclude with the following perspectives:

- Semantic analysis based test generation for LLM generated programs is important. The byproducts from semantic analysis, in our case the distinguishing tests, can be potentially used as evidence of "correctness" of LLM-generated programs to enhance developers' trust when using AI programming assistants.
- Automated program specification generation from LLMs is promising. Despite not being perfect, they can be valuable hints to alert the users of AI-programming assistants about potential untrustworthiness in the automatically generated programs and they can be used for low-effort human-assisted gradual oracle inference.

DATA AVAILABILITY

We release LLMCodeChoice and all data mentioned in the paper at https://doi.org/10.5281/zenodo.10390291 to facilitate future research.

Acknowledgments

This research is supported by a Singapore Ministry of Education (MoE) Tier 3 grant "Automated Program Repair", MOE-MOET32021-0001, and the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

References

[1] 2023. Amazon CodeWhisperer. https://aws.amazon.com/codewhisperer/

[2] 2023. CrossHair: An analysis tool for Python that blurs the line between testing and type systems. https://github.com/pschanely/CrossHair

[3] 2024. Introducing ChatGPT. https://openai.com/blog/chatgpt. Accessed: 2023-11-21.

[4] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. *Syntax-guided synthesis.* IEEE. 1–8 pages. https://doi.org/10.1109/fmcad.2013.6679385

[5] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *arXiv.org* abs/2108.07732 (8 2021). https://arxiv.org/abs/2108.07732

[6] John Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. 2013. Regression verification using impact summaries. In *Model Checking Software: 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings 20.* Springer, 99–116.

[7] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering.* 13–24.

[8] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *Neural Information Processing Systems* abs/2005.14165 (5 2020), 1877–1901. https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[10] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests.. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023.* OpenReview.net. https://openreview.net/pdf?id=ktrw68Cmu9c

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv.org* abs/2107.03374 (7 2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[12] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. *arXiv.org* abs/2304.05128 (4 2023). https://doi.org/10.48550/arxiv.2304.05128

[13] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 85–99.

[14] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models.. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, IEEE, 1469–1481. https://doi.org/10.1109/icse48619.2023.00128

[15] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM SIGPLAN Notices* 46, 1 (1 2011), 317–330.

[16] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (8 2012), 97–105.

[17] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis., In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). *2010 ACM/IEEE 32nd International Conference on Software Engineering* 1, 215–224. http://nma.berkeley.edu/ark:/28722/bk000662095

[18] Shuyang Jiang, Yuhao Wang, and Yu Wang. 2023. SelfEvolve: A Code Evolution Framework via Large Language Models. *arXiv.org* abs/2306.02907 (6 2023). https://doi.org/10.48550/arxiv.2306.02907

[19] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction.. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, IEEE, 2312–2323. https://doi.org/10.1109/icse48619.2023.00194

[20] Darren Key, Wen-Ding Li, and Kevin Ellis. 2022. I Speak, You Verify: Toward Trustworthy Neural Program Synthesis. *arXiv preprint arXiv:2210.00848* abs/2210.00848 (2022). https://doi.org/10.48550/arxiv.2210.00848

[21] 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 (Melbourne, Victoria, Australia). *International Conference on Software Engineering*, 919–931. https://doi.org/10.1109/icse48619.2023.00085

[22] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. 2022. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems* 35 (2022), 3843–3857.

[23] Jia Li, Ge Li, Yongming Li, and Zhi Jin. 2023. Structured Chain-of-Thought Prompting for Code Generation. arXiv:2305.06599 [cs.SE]

[24] 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. *Science* abs/2203.07814, 6624 (12 2022), 1092–1097. https://doi.org/10.48550/arxiv.2203.07814

[25] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. *arXiv.org* abs/2305.01210 (5 2023). https://doi.org/10.48550/arxiv.2305.01210

[26] David Maciver and Zac Hatfield-Dodds. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4 (11 2019), 1891. https://doi.org/10.21105/joss.01891

[27] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. *arXiv.org* abs/2303.17651 (3 2023). https://doi.org/10.48550/arxiv.2303.17651

[28] OpenAI. 2023. GPT-4 Technical Report. https://doi.org/10.48550/arxiv.2303.08774 arXiv:2303.08774 [cs.CL]

[29] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (8 2015), 619–630.

[30] 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).

[31] Brayan Stiven Torrres Ovalle. 2023. GitHub Copilot. https://doi.org/10.26507/paper.2300

[32] Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Păsăreanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering.* 226–237.

[33] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *arXiv.org* abs/2308.12950 (8 2023). https://doi.org/10.48550/arxiv.2308.12950

[34] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A Grammar-Based Structural CNN Decoder for Code Generation., In The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01, 7055–7062. https://doi.org/10.1609/aaai.v33i01.33017055

[35] 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* (9 2020).

[36] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. 2022. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *Neural Information Processing Systems* abs/2201.11903 (1 2022). https://arxiv.org/abs/2201.11903

[37] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal Fuzzing via Large Language Models. *arXiv.org* abs/2308.04748 (8 2023). https://doi.org/10.48550/arxiv.2308.04748

[38] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. https://doi.org/10.48550/arxiv.2305.04764 arXiv:2305.04764 [cs.SE]

[39] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. *arXiv preprint arXiv:1704.01696* (2017), 440–450. https://doi.org/10.18653/v1/p17-1041

[40] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[41] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *arXiv.org* abs/2305.04207 (5 2023). https://doi.org/10.48550/arxiv.2305.04207

[42] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. *arXiv preprint arXiv:2404.05427* (2024).

Received 2024-04-12; accepted 2024-07-03