CodeARC: Benchmarking Reasoning Capabilities of LLM Agents for Inductive Program Synthesis

Anonymous Author(s)
Affiliation
Address
email

Abstract

Inductive program synthesis, or programming by example, requires synthesizing functions from input-output examples that generalize to unseen inputs. While large language model agents have shown promise in programming tasks guided by natural language, their ability to perform inductive program synthesis is underexplored. Existing evaluation protocols rely on static sets of examples and held-out tests, offering no feedback when synthesized functions are incorrect and failing to reflect real-world scenarios such as reverse engineering. We propose CodeARC, the Code Abstraction and Reasoning Challenge, a new evaluation framework where agents interact with a hidden target function by querying it with new inputs, synthesizing candidate functions, and iteratively refining their solutions using a differential testing oracle. This interactive setting encourages agents to perform function calls and self-correction based on feedback. We construct the first large-scale benchmark for general-purpose inductive program synthesis, featuring 1114 functions. Among 18 models evaluated, o3-mini performs best with a success rate of 52.7%, highlighting the difficulty of this task. Fine-tuning LLaMA-3.1-8B-Instruct on curated synthesis traces yields up to a 31% relative performance gain. CodeARC provides a more realistic and challenging testbed for evaluating LLM-based program synthesis and inductive reasoning.

1 Introduction

Inductive reasoning, i.e., the ability to identify patterns and form abstractions from limited examples, is widely recognized as a fundamental aspect of human intelligence [22, 75]. In the context of programming, inductive reasoning underpins the task of synthesizing functions that satisfy given input-output examples and generalize to unseen inputs. This task, commonly referred to as *inductive program synthesis* or programming by example [48, 52, 15, 34], has broad application domains [66, 10].

Recent advances in Large Language Models (LLMs) have led to the emergence of autonomous agents capable of decision-making, multi-step planning, tool use, and iterative self-improvement through interaction and feedback [7, 44, 42, 19, 70]. While much of the existing work focuses on programming tasks guided by natural language [6, 2, 28, 27], we study a fundamentally different problem: inductive program synthesis, where the objective is to infer the target program solely from input-output examples. This setting provides a more rigorous test of inductive reasoning capabilities, as it eliminates natural language descriptions that can trigger retrieval-based completions memorized during model training.

Designing an effective evaluation protocol for inductive program synthesis with LLMs is inherently challenging, as multiple valid functions may satisfy a given set of input-output examples (as demonstrated in Section 5.2). The state-of-the-art protocol [34], which evaluates synthesized functions on

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

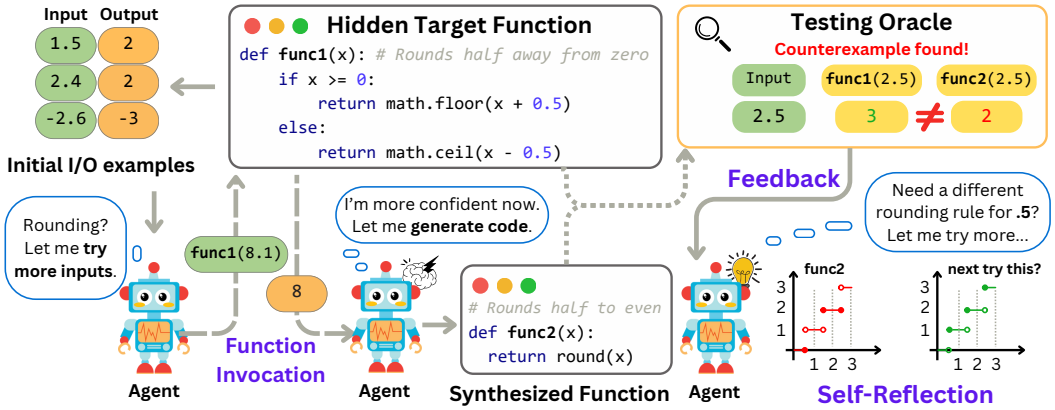

Figure 1: **Overview of CodeARC.** Our framework evaluates LLMs' reasoning capabilities in inductive program synthesis. The agent begins with input-output examples, interacts with a hidden target function via function calls, and uses a differential testing oracle to check the correctness of the synthesized function for self-reflection and refinement.

held-out test cases after presenting 10 fixed input-output examples, has several notable limitations. First, a small, static set of input-output examples may underspecify the target functions, especially for those with complex logic. Moreover, held-out tests may fail to reveal subtle semantic discrepancies between the generated and intended implementations. In addition, when the model produces an incorrect solution, it receives no feedback and has no opportunity to revise or explore alternatives. Finally, existing benchmarks for inductive program synthesis [18, 66, 69, 10, 34] are focused on domain-specific tasks and do not assess the ability of LLMs to synthesize functions written in general-purpose programming languages.

To address the limitations of existing evaluation protocols for inductive program synthesis, we introduce CodeARC, the *Code Abstraction and Reasoning Challenge*, inspired by real-world scenarios such as decompilation and reverse engineering [36, 49, 1]. In such settings, an agent is given a binary executable (without source code) and must synthesize an equivalent source code by observing input-output behavior. Rather than relying on a fixed dataset, the agent can query the binary with new inputs, invoke a differential testing oracle to compare the synthesized program against the reference binary, and use counterexamples to iteratively refine its solution.

As shown in Figure 1, our framework CodeARC emulates this process: LLM-based agents interact with an environment by starting with an initial set of input-output examples, querying the ground-truth function for additional examples, and debugging synthesized code using a differential testing oracle. We impose fixed budgets on both the number of observable input-output examples and the number of testing oracle invocations for self-debugging. The task requires agents to proactively generate inputs (function calls) and revise solutions based on feedback (self-reflection). This interactive setup offers a more realistic alternative to prior static evaluation protocols.

We construct the first comprehensive dataset for general-purpose inductive program synthesis, featuring 1114 functions with initial input-output examples. Our benchmark targets general programming tasks and employs two state-of-the-art differential testing tools [45, 13] for correctness evaluation.

Our experiments demonstrate that CodeARC poses a significant challenge for LLM-based inductive program synthesis. Among the 18 models evaluated, OpenAI o3-mini performs the best overall, yet only achieves 52.7% success rate. We further conduct ablation studies to analyze how budgets on the number of input-output examples and oracle calls affect model performance. To enhance model capabilities, we generate synthetic fine-tuning data with curated synthesis traces that capture the reasoning steps. We show that supervised fine-tuning on LLaMA-3.1-8B-Instruct yields up to a 31% relative performance improvement.

In summary, our contributions are as follows:

- **Interactive evaluation protocol for inductive program synthesis.** We introduce a setup where agents start with fixed input-output examples but can generate new inputs to query ground-truth

functions and invoke a differential testing oracle to self-correct their solutions. This setup brings the task closer to a real-world setting, e.g. reverse-engineering.

- **General-purpose benchmark with extensive evaluation.** We construct the first large-scale, general-purpose benchmark for this task, including 1114 diverse functions. Among the 18 models evaluated, o3-mini achieves the best overall performance but still only reaches a success rate of 52.7%.
- **Synthetic data and fine-tuning.** To boost model performance, we generate synthetic fine-tuning data containing curated synthesis traces that capture both function invocations and reasoning steps. We show that fine-tunining on LLaMA-3.1-8B-Instruct yields up to a 31% relative performance improvement.

2 Related Work

Inductive Program Synthesis Traditional inductive program synthesis methods rely solely on input-output examples, without natural language input. They focus on domain-specific tasks like string and data transformations [18, 60, 73], SQL [66], visual programming [67], and quantum computing [10]. These approaches typically define a domain-specific language and use tailored search algorithms to prune the space efficiently [16, 58, 14, 20, 50]. In contrast, we introduce the first general-purpose program synthesis benchmark for LLM-powered agents.

LLM Benchmarks for Code Most LLM benchmarks—such as HumanEval+ [6, 40], MBPP+ [2, 40], APPS [23], and others [35, 41, 33, 26, 3, 54, 43, 12, 62, 80, 76, 31, 56, 64, 65]—evaluate code generation from natural language. Beyond generation, tasks like I/O prediction [17, 37], execution prediction [38, 30, 53, 11], bug localization [61], and program equivalence [68] have also been studied. In contrast, we focus on predicting function bodies purely from input-output examples, without natural language. Prior work [34, 4] targets domain-specific tasks, while we introduce a general-purpose benchmark with an interactive evaluation protocol.

LLM Benchmarks for Reasoning LLMs are widely benchmarked on reasoning tasks across domains, including commonsense [63], mathematical [9], and logical [21, 51, 39, 55]. Inductive reasoning, a core cognitive skill that generalizes from limited examples [22], is increasingly studied in LLMs [32, 46, 71, 5]. ARC [8] is a prominent benchmark for abstract pattern induction. Our work shares this goal but is for inductive program synthesis.

LLM-powered Agents LLM-based agents have shown strong performance in domains like web navigation [78, 79], code generation [77, 29], and ML experimentation [25]. They interact with environments, invoke functions, make decisions, and self-reflect [47, 57, 72, 24, 59]. We introduce the first benchmark to systematically evaluate agents' capabilities in inductive program synthesis, providing a rigorous testbed for inductive reasoning and program synthesis.

3 Method

3.1 Problem Definition of Inductive Program Synthesis

We formalize the inductive program synthesis task as follows. Let f^* be a *hidden* ground-truth function that maps inputs $x \in \mathcal{X}$ to outputs $y \in \mathcal{Y}$. The synthesizer is given an initial set of input-output examples $\mathcal{E}_0 = \{(x_i, y_i)\}_{i=1}^n$, where $y_i = f^*(x_i)$, and the goal is to synthesize a program $\hat{f}$ such that $\hat{f} \equiv f^*$, i.e.,

$$\forall x \in \mathcal{X}, \quad \hat{f}(x) = f^*(x).$$

To evaluate whether a synthesized function $\hat{f}$ is correct, we introduce a *differential testing oracle* $\mathcal{O}$. The oracle takes as input both the synthesized function $\hat{f}$ and the hidden ground-truth function f^* and attempts to identify inputs on which their behaviors differ. Formally, the oracle operates as follows:

$$\mathcal{O}(f^*, \hat{f}) = \begin{cases} \text{Pass}, & \text{if } \forall x \in \mathcal{X}_{\text{test}},\ \hat{f}(x) = f^*(x); \\ \text{Fail}(x), & \text{if } \exists x \in \mathcal{X}_{\text{test}} \text{ such that } \hat{f}(x) \neq f^*(x), \end{cases}$$

where $\mathcal{X}_{\text{test}} \subseteq \mathcal{X}$ is a set of test inputs dynamically selected by the oracle.

Unlike fixed held-out test sets used in prior work, the differential testing oracle conditions on both f^* and the candidate $\hat{f}$, generating targeted inputs to expose discrepancies. On failure, it returns a counterexample $x \in \mathcal{X}_{\text{test}}$ such that $\hat{f}(x) \neq f^*(x)$. Note that program equivalence checking is fundamentally undecidable, and thus no perfect oracle exists. To approximate oracle functionality, we adopt two state-of-the-art differential testing tools, providing a more robust and practical evaluation.

3.2 Interactive Evaluation Protocol for LLM Agents

To evaluate the capabilities of LLM-based agents in inductive program synthesis, we introduce an interactive protocol. This protocol extends beyond static evaluation settings by enabling dynamic interaction with the hidden ground-truth function and the differential testing oracle.

Initial Information. At the beginning of the task, the agent is provided with an initial set of input-output examples $\mathcal{E}_0 = \{(x_i, y_i)\}_{i=1}^{n}$, as explained previously. This set serves as partial information about the target function.

Action Space. During evaluation, the agent may take two types of actions. First, it may query the ground-truth function f^* at a chosen input $x \in \mathcal{X}$, and obtain the corresponding output $f^*(x)$, thereby augmenting its observed set of input-output pairs. Second, it may synthesize a candidate program $\hat{f}$ and invoke the differential testing oracle $\mathcal{O}(f^*, \hat{f})$, which returns PASS if no discrepancies are found on a dynamically generated test set, or FAIL with a counterexample $x \in \mathcal{X}_{\text{test}}$ such that $\hat{f}(x) \neq f^*(x)$.

Self-Reflection. If the oracle returns FAIL, a counterexample δ, which is a tuple of $(x, \hat{f}(x), f^*(x))$ will be provided to the agent. This counterexample helps the agent to self-reflect and revise its current hypothesis, either by issuing additional queries f^* or synthesizing new programs. The ability to take such feedback is crucial for iterative refinement.

Budget Constraints. The agent operates under two budget parameters: B_{io} and B_{oracle}. B_{io} limits the total number of input-output examples that the agent can observe from the ground-truth function f^*, while B_{oracle} limits the number of invocations to the differential testing oracle $\mathcal{O}$.

Evaluation Metrics. The task is considered successful if the final synthesized program $\hat{f}$, produced within the given budgets, receives a PASS from the differential testing oracle $\mathcal{O}(f^*, \hat{f})$. We assess the performance of an LLM agent along two dimensions: correctness and efficiency, with correctness being the primary criterion. The objective is to synthesize correct programs, even at the cost of increased resource consumption. Correctness is quantified by the success rate, defined as the proportion of problems in the dataset for which the agent successfully synthesizes a correct program within the specified budget. Efficiency is measured by the average number of input-output queries and oracle invocations utilized per problem.

3.3 Benchmark Preparation

Our benchmark is designed to evaluate LLM agents on the inductive synthesis of general-purpose Python programs. This contrasts with prior benchmarks that focus on domain-specific tasks or programs written in domain-specific languages, such as string manipulation and SQL query generation [74, 10, 34].

Programs for Synthesis. We curate a diverse collection of Python functions sampled from three established code generation benchmarks: HumanEval [6], MBPP [2], and APPS [23]. HumanEval and MBPP primarily consist of simple, entry-level programming tasks, whereas APPS contains more challenging problems that resemble competition-level code exercises. Importantly, we extract only the function bodies from these benchmarks and do not use any accompanying natural language descriptions.

Annotated vs. Anonymized. To assess the extent to which function names help the LLM agent synthesize the correct program, we construct two versions of the benchmark. In the *annotated* version, function names that reflect the intended functionality of the task (e.g., `is_palindrome`) are made available to the agent. In the *anonymized* version, all function names are replaced with a generic identifier (i.e., `solution`). This design isolates the influence of identifier cues on synthesis performance. We report results on both versions.

Initial Input-Output Examples. Each synthesis task includes 10 fixed input-output examples that specify the target function's expected behavior. We use GPT-4o to generate diverse inputs and execute the original function to obtain the corresponding ground-truth outputs.

Synthetic Data Generation. We generate a synthetic dataset for fine-tuning (described in more detail in Section 3.4) by first collecting 50 seed Python functions that are *disjoint* from the evaluation set. Using these seeds, we prompt GPT-4o to synthesize a diverse set of new functions, yielding 10,000 candidates. For each generated function, we additionally instruct GPT-4o to produce 10 representative inputs that expose the function's behavior and highlight patterns in its input-output relationships. These inputs are executed to verify the executability of the functions, and we discard any that fail at this stage. To reduce redundancy, we deduplicate the dataset based on function names, finally resulting in 5,405 unique Python functions used for fine-tuning.

3.4 Fine-Tuning on Synthetic Data

We investigate whether simply fine-tuning on curated synthesis traces, which capture both function invocations and reasoning steps, can improve LLM performance on CodeARC. We adopt a knowledge distillation fine-tuning approach, where the student model learns to imitate the reasoning and behavior of a teacher model that has access to the ground-truth function body of f^*.

During training, we first run the interactive evaluation protocol described in Section 3.2 with a frozen teacher model. Unlike a standard evaluation, we prepend a set of task-specific instructions P_{f^*} to each teacher prompt. This prefix includes the function body of f^* and explicitly instructs the teacher to (1) query f^* on informative inputs, (2) explain the rationale behind those queries, and (3) synthesize the full implementation of f^* only when confident that the correct logic can be inferred from the accumulated input-output pairs. The student model, which is the only model we fine-tune, learns to mimic the teacher's reasoning and synthesis behavior without ever seeing the function body.

We provide the teacher with access to f^* because we find that, in many cases, the model struggles to solve the task independently. Without knowledge of the ground truth, the teacher is often too weak to generate meaningful queries or explanations, limiting the effectiveness of the resulting supervision.

While executing the evaluation protocol, we record the multi-turn conversation history C_T, which comprises the teacher model's prompts and responses. Let n be the total number of turns in C_T and denote by x^i the sequence of tokens in the ith turn. Furthermore, let p represent the number of tokens in the teacher-specific instruction prefix P_{f^*}.

We fine-tune the student model using a language modeling objective by minimizing the negative log-likelihood of predicting the next token in C_T:

$$\mathcal{L} = -\sum_{i=1}^{n} \sum_{j=p+1}^{|x^i|} \log P\left(x_j^i \mid C_{T,<i}, x_{<j}^i\right).$$

Here, $P\left(x_j^i \mid C_{T,<i}, x_{<j}^i\right)$ denotes the probability of generating token x_j^i given all tokens from previous turns, $C_{T,<i}$, and the tokens preceding x_j^i in the current turn, $x_{<j}^i$. By starting the inner sum at $j = p + 1$, the teacher-specific instructions P_{f^*} are excluded from the loss computation, ensuring that the training signal comes only from the parts of C_T available during inference.

4 Experiment Setup

We construct two versions (annotated and anonymized) of the 1114 Python functions, drawn from HumanEval+, MBPP+ [40], and APPS [23]. Table 1 summarizes key statistics. Unlike prior work

Model	Annotated Dataset			Anonymized Dataset		
	#I/O	# Oracle	Success (%)	#I/O	# Oracle	Success (%)
Llama-3.2-3B	28.3	1.9	11.0	29.3	2.0	4.8
Mixtral-8x7B	27.4	1.9	20.3	28.5	1.9	12.0
Llama-3.1-8B	28.0	1.8	19.3	28.6	1.9	13.7
Mixtral-8x22B	26.7	1.8	25.1	28.1	1.9	15.0
QwQ-32B	24.6	1.8	20.0	25.7	1.9	15.4
Qwen2.5-7B	26.9	1.8	29.2	28.3	1.9	15.8
Llama-3.2-11B	27.3	1.8	24.9	28.3	1.9	16.1
gpt-4o-mini	27.0	1.8	26.1	27.9	1.8	18.5
Llama-3.2-90B	26.2	1.8	28.4	27.7	1.9	19.7
Llama-3.1-70B	26.9	1.8	30.1	27.9	1.9	20.0
Qwen2.5-72B	25.5	1.7	30.1	27.1	1.8	21.6
Llama-3.1-405B	24.2	1.7	38.6	26.0	1.8	26.7
gpt-4o	23.4	1.7	37.8	25.2	1.8	28.7
DeepSeek-V3	23.7	1.7	37.7	25.1	1.8	29.5
claude3.7-sonnet	23.6	1.7	39.0	24.6	1.7	33.8
DeepSeek-R1	18.6	1.6	49.8	20.3	1.7	41.3
o1-mini	21.0	1.6	53.2	21.5	1.6	47.7
o3-mini	15.6	1.5	59.5	16.0	1.6	52.7

Table 2: Success rates of LLMs on CodeARC using both annotated and anonymized datasets. We also report the average number of observed input-output examples and oracle invocations. All open-source models are instruction-tuned.

in program synthesis that often focuses on domain specific languages and constrained settings, our benchmark consists of programs written in Python, a general purpose language that captures a broader range of real-world algorithms and tasks.

For the main evaluation (Section 5.1), we provide 10 initial input-output examples and set the query budget to 30 input-output pairs and 2 oracle calls ($B_{\text{io}} = 30$, $B_{\text{oracle}} = 2$), chosen based on practical constraints such as API cost and runtime. Section 5.3 reports ablation studies on both budgets. We use two state-of-the-art differential testing tools, PYNGUIN [45] and MOKAV [13]. For supervised fine-tuning on synthetic reasoning trajectories, we use gpt-4o as the teacher model and LLaMA-3.1-8B-Instruct as the student. See Appendix A for further experiment details, including prompts and the fine-tuning parameters.

Source	Functions	Lines of Code		
		Min	Max	Avg
HumanEval+	78	7	56	18.5
MBPP+	131	2	21	3.9
APPS	905	2	74	9.5
Annotated	1114	2	74	9.5
Anonymized	1114	2	74	9.5

Table 1: Number of functions and lines of code statistics for each benchmark source across both dataset versions.

5 Results

5.1 Main Results

Table 2 shows the results for 18 large language models on CodeARC. The order is sorted based on the success rate on the anonymized dataset. We also report the average number of observed input-output examples and oracle invocations. Our findings are as follows:

Reasoning models perform best. Reasoning models (o3-mini, o1-mini, DeepSeek-R1) achieve the highest success rates, all exceeding 40% on the anonymized dataset. They also require fewer I/O examples and oracle calls, indicating greater accuracy and efficiency.

CodeARC is a challenging benchmark. Among the 18 evaluated models, only OpenAI's o3-mini achieves over 50% success on both datasets (i.e., 59.5% on the annotated and 52.7% on the

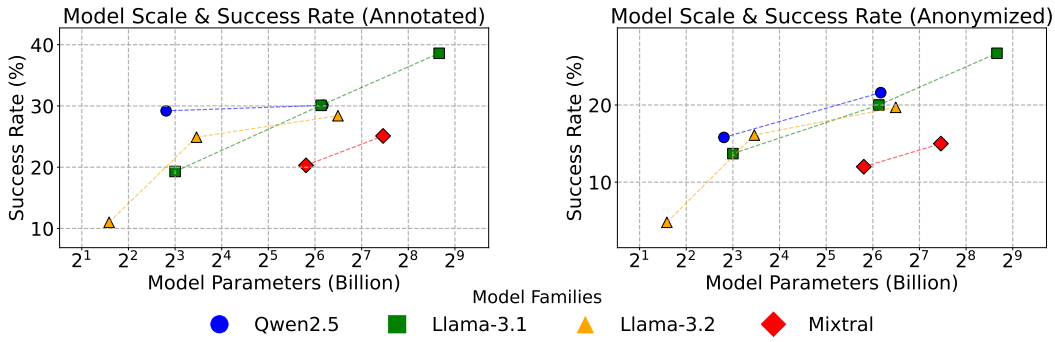

Figure 2: Scaling trend on CodeARC.

Metric	# Problems (%)
Pass1: Initial I/O Examples	1506 (67.6%)
Pass2: Testing Oracle	866 (38.9%)
Δ = **Pass1** − **Pass2**	**640 (28.7%)**

Table 3: Number of problems (in both datasets) where the synthesized function passes the initial examples compared to the oracle.

Model	Success Rate (%)		
	10 I/O	20 I/O	30 I/O
o1-mini	43.7	49.6	50.5
o3-mini	51.3	53.8	56.1

Table 4: Success rates (%) with varying budgets on the observable input-output examples (on both datasets).

anonymized) while all other models fall short of this threshold. This underscores the difficulty of the task and reveals the limitations of current models in inductive reasoning.

Anonymization of function names reduces performance, but trends persist. All models show a modest drop in success rate on the anonymized dataset. However, the overall ranking remains largely consistent. This suggests that while the presence of meaningful function names provides some benefit, strong inductive reasoning remains the main factor behind high performance on this synthesis benchmark.

Scaling up model size improves performance. Larger models generally achieve better performance, as shown in Figure 2 (log-scale x-axis). All model families exhibit scaling trends, though with varying consistency. Llama-3.1 scales steadily, while Llama-3.2 plateaus at larger sizes, likely due to its multimodal focus. Qwen2.5 shows clearer scaling on anonymized data, where reasoning is required over memorization, highlighting model size's impact on generalization.

5.2 Do Initial Input-Output Examples Underspecify the Target Function?

To assess whether 10 input-output examples [34] suffice to specify the target function, we evaluate the first synthesized function of o3-mini—the strongest model in our study. Functions that pass the initial examples but fail under oracle testing indicate under-specification, motivating the need for additional examples or oracle-guided feedback. As shown in Table 3, 67.6% pass the initial 10 input-output example tests, but only 38.9% pass the oracle, revealing 640 cases (28.7%) where the initial examples fail to uniquely specify the target function. These findings show that initial input-output examples often under-specify program behavior, motivating additional queries and oracle-guided feedback for reliable evaluation. This motivates the design of our interactive evaluation protocol.

5.3 Ablation Study: Input-Output Queries and Oracle Feedback

We perform two ablation studies to evaluate how varying the budgets for querying ground-truth functions and invoking the oracle impacts performance.

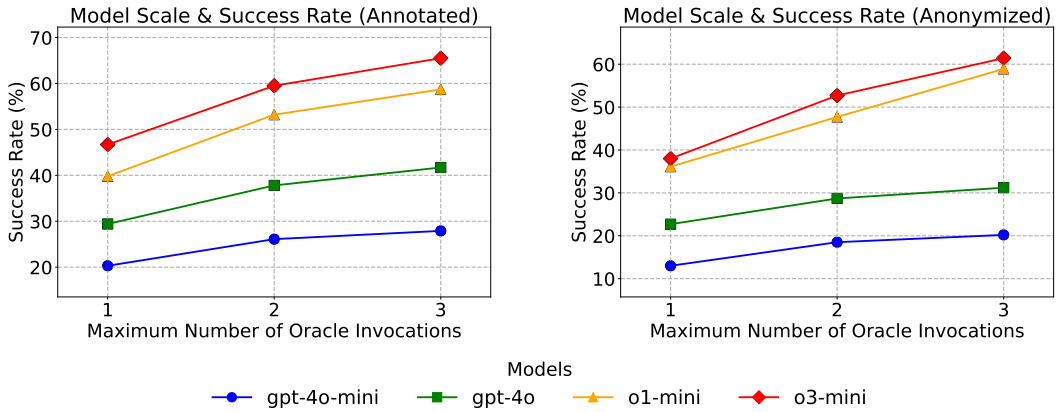

Figure 3: Success rates (%) of LLM models across varying numbers of oracle invocations.

Effect of Input-Output Query Budget. We evaluate o3-mini and o1-mini with input-output budgets of 10, 20, and 30, using the same setup as Section 5.1. Table 4 shows that success rates improve consistently with more examples.

Effect of Oracle Invocation Budget. We vary the number of allowed oracle invocations and report success rates in Figure 3 for four models on both datasets. More oracle calls consistently improve performance, showing that counterexamples from differential testing are valuable for guiding iterative refinement.

These results demonstrate that incorporating both querying mechanisms and oracle feedback consistently enhances overall performance. This improvement underscores the importance of adopting an interactive evaluation protocol rather than relying solely on static, one-shot evaluation approaches.

5.4 Performance of Fine-Tuned Models

Table 5 shows that fine-tuning the LLaMA-3.1-8B-Instruct model on curated synthesis traces yields consistent improvements across both datasets. The larger gain on the annotated variant suggests that fine-tuning is particularly effective when semantically informative identifiers are present. Notably, this performance gap emerges despite both datasets being evaluated under the same model architecture and training methodology discussed in Section 3.4. These results indicate that while fine-tuning helps, there remains substantial room for further improvement. This suggests that future research may focus on enhancing the quality and diversity of the fine-tuning dataset, particularly for the anonymized variant where gains are more limited. Another promising direction is to explore reinforcement learning approaches that directly optimize for higher synthesis success rates, potentially overcoming limitations of supervised fine-tuning alone.

Dataset	Success Rate (%)		
	Base Model	Fine-Tuned	Rel. Δ
Annotated	19.3	25.3	+31%
Anonymized	13.7	15.0	+9.5%

Table 5: Success rates of LLaMA-3.1-8B-Instruct and its fine-tuned variant on annotated and anonymized datasets. Fine-tuning improves performance, especially on the annotated dataset.

5.5 Case Study

Figure 4 shows an interaction trace from our benchmark. The model starts by querying the ground-truth function with edge-case inputs, aiming to probe its behavior beyond the initial examples. It then synthesizes a candidate solution using pairwise comparisons, which passes the given examples but fails on a counterexample with unhashable elements. From the error message, the model correctly infers that the ground-truth function raises a `TypeError`, while its own does not. On the second attempt, it reasons that using a set simplifies the uniqueness check and synthesizes the correct set-

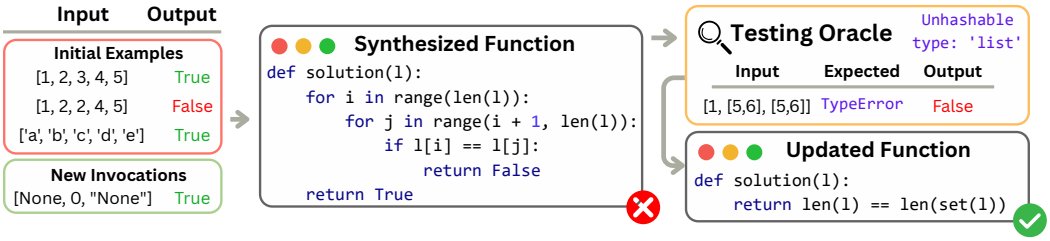

Figure 4: **Case Study.** The model queries edge cases, synthesizes a comparison function, receives a counterexample from the oracle, and corrects it with a set-based solution.

based function. This case illustrates how the model combines function invocation and oracle feedback to perform inductive program synthesis. The full trace is in Appendix A.1.

6 Conclusion

We introduce CodeARC, a new framework for evaluating LLMs on inductive program synthesis through interactive input generation and self-correction. Unlike static protocols, CodeARC allows agents to query a ground truth function and use a differential testing oracle to get feedback for iterative refinement. Designed to assess inductive reasoning from input-output examples, our benchmark covers 1114 diverse and general-purpose functions and evaluates 18 language models. The best-performing model, OpenAI o3-mini, achieves a success rate of 52.7%. Fine-tuning LLaMA-3.1-8B-Instruct on curated synthesis traces results in a 31% relative performance gain. CodeARC provides a more realistic and challenging testbed for evaluating LLM-based inductive program synthesis.

References

[1] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. Extending source code pre-trained language models to summarise decompiled binaries. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 260–271. IEEE, 2023.

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

[3] Debangshu Banerjee, Tarun Suresh, Shubham Ugare, Sasa Misailovic, and Gagandeep Singh. Crane: Reasoning with constrained llm generation, 2025. URL https://arxiv.org/abs/2502.09061.

[4] Shraddha Barke, Emmanuel Anaya Gonzalez, Saketh Ram Kasibatla, Taylor Berg-Kirkpatrick, and Nadia Polikarpova. Hysynth: Context-free llm approximation for guiding program synthesis. *Advances in Neural Information Processing Systems*, 37:15612–15645, 2024.

[5] Chengkun Cai, Xu Zhao, Haoliang Liu, Zhongyu Jiang, Tianfang Zhang, Zongkai Wu, Jenq-Neng Hwang, and Lei Li. The role of deductive and inductive reasoning in large language models. *arXiv preprint arXiv:2410.02892*, 2024.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[7] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

[8] François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

[9] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[10] Haowei Deng, Runzhou Tao, Yuxiang Peng, and Xiaodi Wu. A case for synthesis of recursive quantum unitary programs. *Proceedings of the ACM on Programming Languages*, 8(POPL): 1759–1788, 2024.

[11] Yangruibo Ding, Jinjun Peng, Marcus J Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. Semcoder: Training code language models with comprehensive semantics reasoning. *arXiv preprint arXiv:2406.01006*, 2024.

[12] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.

[13] Khashayar Etemadi, Bardia Mohammadi, Zhendong Su, and Martin Monperrus. Mokav: Execution-driven differential testing with llms. *arXiv preprint arXiv:2406.10375*, 2024.

[14] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices*, 53(4):420–435, 2018.

[15] Jack Feser, Işıl Dillig, and Armando Solar-Lezama. Inductive program synthesis guided by observational program similarity. *Proceedings of the ACM on Programming Languages*, 7 (OOPSLA2):912–940, 2023.

[16] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, 50(6):229–239, 2015.

[17] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

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

[19] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*, 2024.

[20] Sankha Narayan Guria, Jeffrey S Foster, and David Van Horn. Absynthe: Abstract interpretation-guided synthesis. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1584–1607, 2023.

[21] Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Wenfei Zhou, James Coady, David Peng, Yujie Qiao, Luke Benson, et al. Folio: Natural language reasoning with first-order logic. *arXiv preprint arXiv:2209.00840*, 2022.

[22] Brett K Hayes, Evan Heit, and Haruka Swendsen. Inductive reasoning. *Wiley interdisciplinary reviews: Cognitive science*, 1(2):278–292, 2010.

[23] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

[24] Jiaxin Huang, Shixiang Shane Gu, Le Hou, Yuexin Wu, Xuezhi Wang, Hongkun Yu, and Jiawei Han. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*, 2022.

[25] Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation, 2024. URL https://arxiv.org/abs/2310.03302.

[26] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[27] Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2e: Turning any github repository into a programming agent environment. In *Forty-first International Conference on Machine Learning*, 2024.

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

[29] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

[30] Emanuele La Malfa, Christoph Weinhuber, Orazio Torre, Fangru Lin, Samuele Marro, Anthony Cohn, Nigel Shadbolt, and Michael Wooldridge. Code simulation challenges for large language models. *arXiv preprint arXiv:2401.09074*, 2024.

[31] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR, 2023.

[32] Jiachun Li, Pengfei Cao, Zhuoran Jin, Yubo Chen, Kang Liu, and Jun Zhao. Mirage: Evaluating and explaining inductive reasoning process in language models. *arXiv preprint arXiv:2410.09542*, 2024.

[33] Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*, 2023.

[34] Wen-Ding Li and Kevin Ellis. Is programming by example solved by llms? *Advances in Neural Information Processing Systems*, 37:44761–44790, 2025.

[35] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[36] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, pages 1–1, 2010.

[37] Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. Codemind: A framework to challenge large language models for code reasoning. *arXiv preprint arXiv:2402.09664*, 2024.

[38] Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. Code execution with pre-trained language models. *arXiv preprint arXiv:2305.05383*, 2023.

[39] Jian Liu, Leyang Cui, Hanmeng Liu, Dandan Huang, Yile Wang, and Yue Zhang. Logiqa: A challenge dataset for machine reading comprehension with logical reasoning. *arXiv preprint arXiv:2007.08124*, 2020.

[40] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023.

[41] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.

[42] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.

[43] Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023.

[44] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

[45] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, 2022.

[46] Kaijing Ma, Xinrun Du, Yunran Wang, Haoran Zhang, Zhoufutu Wen, Xingwei Qu, Jian Yang, Jiaheng Liu, Minghao Liu, Xiang Yue, et al. Kor-bench: Benchmarking language models on knowledge-orthogonal reasoning tasks. *arXiv preprint arXiv:2410.06526*, 2024.

[47] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594, 2023.

[48] Zohar Manna and Richard J Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[49] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. {RE-Mind}: a first look inside the mind of a reverse engineer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2727–2745, 2022.

[50] Stephen Mell, Steve Zdancewic, and Osbert Bastani. Optimal program synthesis via abstract interpretation. *Proceedings of the ACM on Programming Languages*, 8(POPL):457–481, 2024.

[51] Shen-Yun Miao, Chao-Chun Liang, and Keh-Yih Su. A diverse corpus for evaluating and developing english math word problem solvers. *arXiv preprint arXiv:2106.15772*, 2021.

[52] Brad A Myers. Visual programming, programming by example, and program visualization: a taxonomy. *ACM sigchi bulletin*, 17(4):59–66, 1986.

[53] Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. Next: Teaching large language models to reason about code execution. *arXiv preprint arXiv:2404.14662*, 2024.

[54] Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, et al. L2ceval: Evaluating language-to-code generation capabilities of large language models. *Transactions of the Association for Computational Linguistics*, 12:1311–1329, 2024.

[55] Mihir Parmar, Nisarg Patel, Neeraj Varshney, Mutsumi Nakamura, Man Luo, Santosh Mashetty, Arindam Mitra, and Chitta Baral. Logicbench: Towards systematic evaluation of logical reasoning ability of large language models. *arXiv preprint arXiv:2404.15522*, 2024.

[56] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37: 126544–126565, 2025.

[57] Debjit Paul, Mete Ismayilzada, Maxime Peyrard, Beatriz Borges, Antoine Bosselut, Robert West, and Boi Faltings. Refiner: Reasoning feedback on intermediate representations. *arXiv preprint arXiv:2304.01904*, 2023.

[58] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. *ACM SIGPLAN Notices*, 51(6):522–538, 2016.

[59] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

[60] Rishabh Singh and Sumit Gulwani. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 343–356, 2016.

[61] Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. Cornstack: High-quality contrastive data for better code retrieval and reranking, 2025. URL https://arxiv.org/abs/2412.01007.

[62] Tarun Suresh, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. Is the watermarking of llm-generated code robust?, 2025. URL https://arxiv.org/abs/2403.17983.

[63] Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. Commonsenseqa: A question answering challenge targeting commonsense knowledge. *arXiv preprint arXiv:1811.00937*, 2018.

[64] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. Syncode: Llm generation with grammar augmentation, 2024. URL https://arxiv.org/abs/2403.01632.

[65] Shubham Ugare, Rohan Gumaste, Tarun Suresh, Gagandeep Singh, and Sasa Misailovic. Itergen: Iterative semantic-aware structured llm generation with backtracking, 2025. URL https://arxiv.org/abs/2410.07295.

[66] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466, 2017.

[67] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. Visualization by example, 2019. URL https://arxiv.org/abs/1911.09668.

[68] Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yaofeng Sun, Yuan Liu, Thiago SFX Teixeira, Diyi Yang, et al. Equibench: Benchmarking code reasoning capabilities of large language models via equivalence checking. *arXiv preprint arXiv:2502.12466*, 2025.

[69] Catherine Wong, Kevin M Ellis, Joshua Tenenbaum, and Jacob Andreas. Leveraging language to learn program abstractions and search heuristics. In *International conference on machine learning*, pages 11193–11204. PMLR, 2021.

[70] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.

[71] Yijia Xiao, Edward Sun, Tianyu Liu, and Wei Wang. Logicvista: Multimodal llm logical reasoning benchmark in visual contexts. *arXiv preprint arXiv:2407.04973*, 2024.

[72] Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, James Xu Zhao, Min-Yen Kan, Junxian He, and Michael Xie. Self-evaluation guided beam search for reasoning. *Advances in Neural Information Processing Systems*, 36:41618–41650, 2023.

[73] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. *ACM SIGPLAN Notices*, 51(6):508–521, 2016.

[74] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA): 1–26, 2017.

[75] Kai Yan, Zhan Ling, Kang Liu, Yifan Yang, Ting-Han Fan, Lingfeng Shen, Zhengyin Du, and Jiecao Chen. Mir-bench: Benchmarking llm's long-context intelligence via many-shot in-context inductive reasoning. *arXiv preprint arXiv:2502.09933*, 2025.

[76] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, et al. Natural language to code generation in interactive data science notebooks. *arXiv preprint arXiv:2212.09248*, 2022.

[77] Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *Advances in Neural Information Processing Systems*, 36:54769–54784, 2023.

[78] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models. In *Forty-first International Conference on Machine Learning*, 2024.

[79] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024. URL https://arxiv.org/abs/2307.13854.

[80] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

A Appendix

A.1 Case Study

In the below example, the model first invokes the ground-truth function with 10 more inputs, then tries to synthesize the function, and then gets back the counter-example from the differential testing oracle. Its second trial then passes the oracle's test.

Prompt:

You are a software engineer tasked with implementing a blackbox Python function `solution`. Your goal is to implement this function correctly, ensuring it works for all possible valid inputs, not just the example inputs provided.

Your primary goal is to get the function implementation exactly right. Your secondary goal is to minimize the number of function invocations and implementation attempts.

GIVEN EXAMPLES:

```
print('Result 1: ' + str(solution([1, 2, 3, 4, 5])))
print('Result 2: ' + str(solution([1, 2, 2, 4, 5])))
print('Result 3: ' + str(solution(['a', 'b', 'c', 'd', 'e'])))
print('Result 4: ' + str(solution(['apple', 'banana', 'apple'])))
print('Result 5: ' + str(solution([])))
print('Result 6: ' + str(solution([10, 20, 30, 40, 50, 60])))
print('Result 7: ' + str(solution([10, 20, 30, 30, 50, 60])))
print('Result 8: ' + str(solution(['x', 'y', 'z'])))
print('Result 9: ' + str(solution([1, 1, 1, 1])))
print('Result 10: ' + str(solution(list(range(100)))))
```

Corresponding outputs:

```
Result 1: True
Result 2: False
Result 3: True
Result 4: False
Result 5: True
Result 6: True
Result 7: False
Result 8: True
Result 9: False
Result 10: True
```

Strategy Overview:
You should work in two phases:
1. Exploration Phase: use function invocations strategically to fully understand the function's behavior.
2. Implementation Phase: only after a thorough understanding, attempt an implementation.

This is the first iteration. You must first reason and then decide between one of the following two options:
1. Generate more function invocations in the same format as the examples. You can generate up to 20 additional function invocations across all iterations.
2. Generate an implementation of `solution`. You have only 1 debugging check total. If your implementation is incorrect, you lose your check and are shown a counterexample. If you attempt an implementation and have no debugging checks left and your implementation is incorrect, you cannot take any more actions. This attempt will be considered your final answer.

Assistant:

I'll start with the Exploration Phase and generate additional strategic invocations to better understand the function's behavior over a wide range of inputs. I want to explore cases like unique negative and

non-negative numbers, duplicate negative numbers, duplicate empty strings, duplicate boolean values, and unique special character strings. I also want to verify that single element lists should be unique.

INVOCATIONS:

```
print('Result 11: ' + str(solution([-1, -2, 0, 1])))
print('Result 12: ' + str(solution([-1, -2, -1])))
print('Result 13: ' + str(solution([1, 1.0, 2])))
print('Result 14: ' + str(solution([(1,), (2,), (1,)])))
print('Result 15: ' + str(solution([42])))
print('Result 16: ' + str(solution(["", "a", "b"])))
print('Result 17: ' + str(solution(["", "", "b"])))
print('Result 18: ' + str(solution([True, False, False])))
print('Result 19: ' + str(solution(['!', '@', '#', '$'])))
print('Result 20: ' + str(solution([None, 0, "None"])))
```

Prompt:

Here are the outputs of the function invocations from the previous iteration:

```
Result 11: True
Result 12: False
Result 13: False
Result 14: False
Result 15: True
Result 16: True
Result 17: False
Result 18: False
Result 19: True
Result 20: True
```

STRATEGY UPDATE: Based on the new data above, you should update your understanding of the function and decide on your next action. You have 10 additional function invocations and 1 debugging check left.

Assistant: Based on the input-output pairs I have analyzed, I infer that this function is intended to evaluate whether all elements in a given list are distinct—that is, to determine if there are any duplicate entries within the list. To implement this functionality, I propose using a pairwise comparison strategy. By systematically comparing each element in the list with every subsequent element, it becomes possible to identify any duplicates that may exist. This approach ensures thorough coverage of all potential comparisons and provides a reliable method for detecting non-unique elements in the list.

IMPLEMENTATION:

```
def solution(lst):
    # Use a simple pairwise comparison approach
    n = len(lst)
    for i in range(n):
        for j in range(i + 1, n):
            if lst[i] == lst[j]:
                return False
    return True
```

Prompt:

The implementation you generated in the previous iteration failed for the following input (and potentially other inputs):

```
Failed input: [1, 2, 3, 4, [5, 6], [5, 6]]
Ground Truth Function != Output From Generated Code:
'Error' != 'No Error'
```

This failure provides valuable information about a case you didn't account for. You should decide between generating more invocations to explore and debug the function behavior or generating one final implementation. You have 10 additional function invocations and NO debugging checks left.

Assistant: Based on the error message, it appears that raising an error for this input would be the appropriate response. To handle this effectively, a set-based approach might be more suitable than a pairwise comparison approach. By converting the list to a set, I can efficiently detect unhashable elements, as attempting this operation would naturally raise a TypeError for any such elements. Let me proceed with implementing this approach to validate the reasoning.

IMPLEMENTATION:

```python
def solution(lst):
    # Attempt to convert the list to a set.
    # If any element is unhashable, this will raise a TypeError as expected.
    return len(lst) == len(set(lst))
```

This final implementation passes the differential testing oracle.

A.2 Supervised Fine-Tuning Experiment Setup

Table 6 presents the parameters used for our supervised fine-tuning experiments.

Name	**Value**
Teacher Model	gpt-4o
Student Model	LLaMA-3.1-8B-Instruct
Fine-tuning Context Length	32768
Learning Rate	1.00E-5
LR Schedule	linear
Warmup Ratio	0.03
Epoch	3
Batch Size	8
Gradient Accumulation Steps	8
LoRA Rank	8
LoRA α	8

Table 6: LoRA Fine-Tuning Parameters.