EquiBench: Benchmarking Large Language Models' Understanding of Program Semantics via Equivalence Checking

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

As large language models (LLMs) become integral to code-related tasks, a central question emerges: do LLMs truly understand program execution semantics? We introduce EquiBench, a new benchmark for evaluating LLMs through equivalence checking, i.e., determining whether two programs produce identical outputs for all possible inputs. Unlike prior code generation benchmarks, this task directly tests a model's understanding of code execution semantics. EquiBench consists of 2400 program pairs across four languages and six categories. These pairs are generated through program analysis, compiler scheduling, and superoptimization, ensuring high-confidence labels, nontrivial difficulty, and full automation. The transformations span syntactic edits, structural modifications, and algorithmic changes, covering a broad spectrum of semantic variation. We evaluate 19 state-of-the-art LLMs and find that in the most challenging categories, the best accuracies are 63.8% and 76.2%, only modestly above the 50% random baseline. Further analysis reveals that models often rely on syntactic similarity rather than exhibiting robust reasoning over execution semantics, highlighting fundamental limitations.

1 Introduction

Large language models (LLMs) have rapidly become central to software engineering workflows, powering tools for code generation, program repair, test case generation, debugging, and beyond, significantly boosting developers' productivity [1–3]. This surge of capability has prompted a natural yet fundamental question: Do LLMs merely mimic code syntax they have seen during training, or do they genuinely understand what programs do?

Unlike natural language, code is executable, i.e., its semantic meaning is defined by execution behavior, not just its form [4]. Two programs may differ syntactically yet be semantically equivalent, producing identical outputs for all inputs. Conversely, programs with only minor syntactic differences can behave quite differently at runtime. This gap between surface-level program features and actual execution behavior raises an important question: Does training on static code corpora equip LLMs with a grounded understanding of *program semantics*?

To rigorously assess whether LLMs truly understand code, we need benchmarks that demand reasoning about program execution semantics. However, widely used coding benchmarks such as HumanEval [5] and MBPP [6] primarily test a model's ability to generate short code snippets from natural language descriptions, offering limited insight into whether the model grasps the underlying execution semantics of the code it generates.

In this work, we introduce **equivalence checking** as a new task to evaluate LLMs' understanding of program semantics. Unlike tasks that hinge on superficial syntactic similarity, equivalence checking

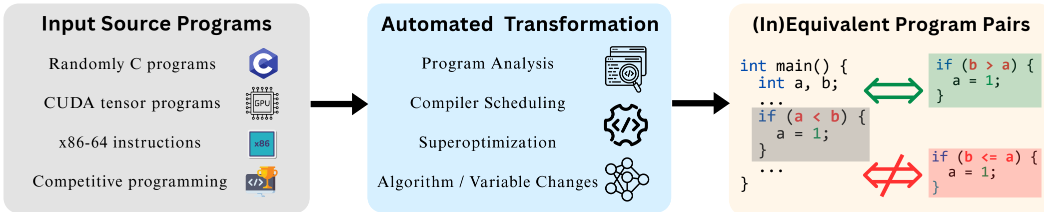

Figure 1: **Overview of EquiBench.** We construct (in)equivalent program pairs from diverse sources, including C and CUDA programs, x86-64 assembly, and competitive programming, using automated transformations based on program analysis, compiler scheduling, superoptimization, and changes in algorithms or variable names.

asks whether *two programs* are semantically equivalent, i.e., whether they *produce identical outputs for all possible inputs*, regardless of how differently they may be written. This poses a fundamentally harder challenge: it requires models to reason about program behavior, not just form. For LLMs, success on this task cannot be achieved through shallow memorization or token-level heuristics alone; it demands a deeper grasp of the control flow, data dependencies of programs, and requires generalization beyond surface patterns.

Designing a benchmark for equivalence checking requires both equivalent and inequivalent program pairs spanning diverse categories, which poses several challenges in terms of *label soundness*, *problem difficulty*, and *automation*. First, it is difficult to guarantee high-confidence labels, as verifying equivalence by exhaustively executing all possible inputs is almost always computationally infeasible. Second, existing generation techniques rely on superficial syntactic edits [7, 8], which are too simplistic to meaningfully challenge state-of-the-art LLMs and fail to probe their semantic reasoning limits. Third, to enable comprehensive evaluation, the benchmark must be large-scale and modular, necessitating a fully automated construction pipeline.

In this work, we introduce **EquiBench**, a dataset of 2400 program pairs for evaluating large language models on equivalence checking. Covering Python, C, CUDA, and x86-64 programs, it enables a systematic assessment of LLMs' ability to reason about program execution semantics.

As illustrated in Figure 1, EquiBench addresses these challenges by automatically constructing both equivalent and inequivalent program pairs from diverse input sources, including randomly generated C and CUDA code, assembly instructions, and competitive programming solutions. To ensure label soundness without exhaustive execution, we apply program transformation techniques grounded in program analysis and superoptimization. To increase problem difficulty beyond trivial edits, we incorporate structural transformations through compiler scheduling and algorithmic equivalences. Finally, the entire generation pipeline is fully automated, enabling scalable construction of a large and diverse benchmark.

Our experiments show that EquiBench is a challenging benchmark for LLMs. Among the 19 models evaluated, OpenAI o4-mini performs best overall, yet achieves only 60.8% in the CUDA category despite reaching the highest overall accuracy of 82.3%. In the two most difficult categories, the best accuracies are 63.8% and 76.2%, respectively, only *modestly better than the random baseline* of 50% for binary classification. In contrast, purely syntactic changes such as variable renaming are much easier, with accuracies as high as 96.5%. We further find, through difficulty analysis, that models often rely on superficial form features such as syntactic similarity rather than demonstrating robust semantic understanding. Moreover, prompting strategies such as few-shot in-context learning and Chain-of-Thought (CoT) prompting *barely* improve LLM performance, underscoring the difficulty of understanding code execution semantics.

In summary, our contributions are as follows:

- **New Task and Dataset:** We introduce equivalence checking as a new task to assess LLMs' understanding of code execution semantics. We present *EquiBench*, a benchmark for equivalence checking spanning four languages and six equivalence categories.
- **Automated Generation:** We develop a fully automated pipeline for constructing diverse (in)equivalent program pairs using techniques that ensure high-confidence labels and nontriv-

ial difficulty. The pipeline covers transformations ranging from syntactic edits to structural modifications and algorithmic equivalence.

- **Evaluation and Analysis:** We evaluate 19 state-of-the-art models on EquiBench. In the two most challenging categories, the best accuracies are only 63.8% and 76.2%, highlighting fundamental limitations. Our analysis shows that models often rely on superficial form features rather than demonstrating a robust understanding of execution semantics.

2 Related Work

LLM Reasoning Benchmarks Extensive research has evaluated LLMs' reasoning capabilities across diverse tasks [9–18]. In the context of code reasoning, i.e., predicting a program's execution behavior without running it, CRUXEval [19] focuses on input-output prediction, while CodeMind [20] extends evaluation to natural language specifications. Another line of work seeks to improve LLMs' code simulation abilities through prompting [21] or targeted training [22–25]. Unlike prior work that tests LLMs on specific inputs, our benchmark evaluates their ability to reason over all inputs.

Equivalence Checking Equivalence checking underpins applications such as performance optimization [26–28], code transpilation [29–32], refactoring [33], and testing [34, 35]. Due to its undecidable nature, no algorithm can decide program equivalence for all program pairs while always terminating. Existing techniques [36–41] focus on specific domains, such as SQL query equivalence [42–44]. EQBENCH [7] and SeqCoBench [8] are the main datasets for equivalence checking but have limitations. EQBENCH is too small (272 pairs) for LLM evaluation, while SeqCoBench relies only on statement-level syntactic changes (e.g., renaming variables). In contrast, our work introduces a broader set of equivalence categories, creating a more systematic and challenging benchmark.

3 Benchmark Construction

While we have so far discussed only the standard notion of equivalence (that two programs produce the same output on any input), there are other, more precise definitions of equivalence used for each category in the benchmark. For each category, we provide the definition of equivalence, which is included in the prompt when testing LLM reasoning capabilities. We describe the process of generating (in)equivalent pairs for the following six categories:

- **DCE**: C program pairs generated via the compiler's dead code elimination (DCE) pass (Section 3.1).
- **CUDA**: CUDA program pairs created by applying different scheduling strategies using a tensor compiler (Section 3.2).
- **x86-64**: x86-64 assembly program pairs generated by a superoptimizer (Section 3.3).
- **OJ_A**, **OJ_V**, **OJ_VA**: Python program pairs from online judge submissions, featuring algorithmic differences (OJ_A), variable-renaming transformations (OJ_V), and combinations of both (OJ_VA) (Section 3.4).

3.1 Pairs from Program Analysis (DCE)

Dead code elimination (DCE), a compiler pass, removes useless program statements. After DCE, remaining statements in the modified program naturally *correspond* to those in the original program.

Definition of Equivalence. Two programs are considered equivalent if, when executed on the same input, they *always* have identical *program states* at all corresponding points reachable by program execution. We expect language models to identify differences between the two programs, align their states, and determine whether these states are consistently identical.

Example. Figure 2 illustrates an inequivalent pair of C programs. In the left program, the condition `(p1 == p2)` compares the memory address of the first element of the array `b` with that of the static variable `c`. Since `b` and `c` reside in different memory locations, this condition can never be satisfied.

```
__global__ void GEMV(const float* A,
                     const float* x,
                     float* y,
                     int R,
                     int C) {

    // Calculate the row index
    // assigned to the thread
    int r = blockIdx.x * blockDim.x
            + threadIdx.x;

    // Return if out of bounds
    if (r >= R) return;
    float s = 0.0f;

    for (int c = 0; c < C; c++) {
        s += A[r * C + c] * x[c];
    }

    y[r] = s;
}
```

```
__global__ void GEMV(const float* A, const float* x,
                     float* y, int R, int C) {
    __shared__ float tile[32]; // tiling with shared memory
    int r = blockIdx.x * blockDim.x + threadIdx.x;
    bool valid = (r < R);
    float s = 0.0f;
    for (int start = 0; start < C; start += 32) {
        for (int i = threadIdx.x; i < 32; i += blockDim.x) {
            int c = start + i;
            if (c < C) tile[i] = x[c]; // load x into tile
        }
        __syncthreads();
        if (valid) {
            for (int j = 0; j < min(32, C - start); j++) {
                s += A[r * C + (start + j)] * tile[j];
            }
        }
        __syncthreads();
    }
    if (valid) y[r] = s;
}
```

Figure 3: **An equivalent pair from the CUDA category in EquiBench.** Both programs perform matrix-vector multiplication ($y = Ax$). The right-hand program uses *shared memory tiling* to improve performance. Tensor compilers are utilized to explore different *scheduling strategies*, automating the generation.

As a result, the assignment `c = 1` is never executed in the left program but is executed in the right program. This difference in program state during execution renders the pair inequivalent.

Automation. This reasoning process is automated by compilers through *alias analysis*, which statically determines whether two pointers can reference the same memory location. Based on this analysis, the compiler's *Dead Code Elimination (DCE)* pass removes code that does not affect program semantics to improve performance.

Dataset Generation. We utilize CSmith [45] to create an initial pool of random C programs. Building on techniques from prior compiler testing research [46], we implement an LLVM-based tool [47] to classify code snippets as either dead or live. Live code is further confirmed by executing random inputs with observable side effects. Equivalent program pairs are generated by eliminating dead code, while inequivalent pairs are generated by removing live code.

```
char b[2];
static int c = 0;

int main() {
  char* p1 = &b[0];
  int*  p2 = &c;
  ...
  if (p1 == p2) {
    c = 1; //dead code
  }
  ...
  return 0;
}
```

```
char b[2];
static int c = 0;

int main() {
  char* p1 = &b[0];
  int*  p2 = &c;
  ...
  if (true) {
    c = 1; //live code
  }
  ...
  return 0;
}
```

Figure 2: **An inequivalent pair from the DCE category in EquiBench.** In the left program, `c = 1` is dead code and has no effect on the program state, whereas in the right program, it is executed and alters the program state. Such cases are generated using the Dead Code Elimination (DCE) pass in compilers.

3.2 Pairs from Compiler Scheduling (CUDA)

Definition of Equivalence. Two CUDA programs are considered equivalent if they produce the same mathematical output for any valid input, *disregarding floating-point rounding errors*. This definition *differs* from that in Section 3.1, as it does not require the internal program states to be identical during execution.

Example. Figure 3 shows an equivalent CUDA program pair. Both compute matrix-vector multiplication $y = Ax$, where A has dimensions (R, C) and x has size C. The right-hand program applies the *shared memory tiling* technique, loading `x` into shared memory `tile` (declared with `__shared__`). Synchronization primitives `__syncthreads()` are properly inserted to prevent synchronization issues.

Automation. The program transformation can be automated with tensor compilers, which provide a set of *schedules* to optimize loop-based programs. These schedules include loop tiling, loop fusion, loop reordering, loop unrolling, vectorization, and cache optimization. For any given schedule, the compiler can generate the transformed code. While different schedules can significantly impact program performance on the GPU, they do not affect the program's correctness (assuming no compiler bugs), providing the foundation for automation.

Dataset Generation. We utilize TVM as the tensor compiler [48] and sample tensor program schedules from TenSet [49] to generate equivalent CUDA program pairs. Inequivalent pairs are created by sampling code from different tensor programs.

3.3 Pairs from a Superoptimizer (x86-64)

Definition of Equivalence. Two x86-64 assembly programs are considered equivalent if, for any input provided in the specified input registers, both programs produce identical outputs in the specified output registers. Differences in other registers or memory are ignored for equivalence checking.

Example. Figure 4 shows an example of an equivalent program pair in x86-64 assembly. Both programs implement the same C function, which counts the number of bits set to 1 in the variable `x` (mapped to the `%rdi` register) and stores the result in `%rax`. The left-hand program, generated by GCC with O3 optimization, uses a loop to count each bit individually, while the right-hand program, produced by a superoptimizer, leverages the `popcnt` instruction, a hardware-supported operation for efficient bit counting. The superoptimizer verifies that both programs are semantically equivalent. Determining this equivalence requires a solid understanding of x86-64 assembly semantics and the ability to reason about all possible bit patterns.

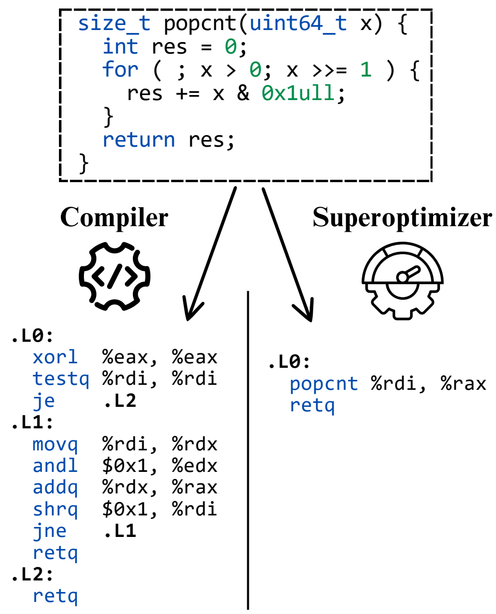

Figure 4: **An equivalent pair from the x86-64 category in EquiBench.** Both programs are compiled from the same C function shown above—the left using a compiler and the right using a *superoptimizer*. The function counts the number of set bits in the input `%rdi` register and stores the result in `%rax`. Their equivalence has been formally verified by the superoptimizer.

Automation. A superoptimizer searches a space of programs to find one equivalent to the target. Test cases efficiently prune incorrect candidates, while formal verification guarantees the correctness of the optimized program. Superoptimizers apply aggressive and non-local transformations, making semantic equivalence reasoning more challenging. For example, in Figure 4, while a traditional compiler translates the loop in the source C program into a loop in assembly, a superoptimizer can find a more optimal instruction sequence by leveraging specialized hardware instructions. Such semantic equivalence is beyond the scope of traditional compilers.

Dataset Generation. We use Stoke [50] to generate program pairs. Assembly programs are sampled from prior work [51], and Stoke applies transformations to produce candidate programs. If verification succeeds, the pair is labeled as equivalent; if the generated test cases fail, it is labeled as inequivalent.

3.4 Pairs from Programming Contests

Definition of Equivalence. Two programs are considered equivalent if they solve the same problem by producing the same output for any valid input, as defined by the problem description. Both programs, along with the problem description, are provided to determine equivalence.

Example. Given the problem description in Figure 5, all four programs are equivalent as they correctly compute the Fibonacci number. The **OJ_A** pairs demonstrate **algorithmic** equivalence—the left-hand program uses recursion, while the right-hand program employs a for-loop. The **OJ_V** pairs are generated through **variable renaming**, a pure syntactic transformation that can obscure the program's semantics by removing meaningful variable names. The **OJ_VA** pairs combine **both** algorithmic differences and variable renaming.

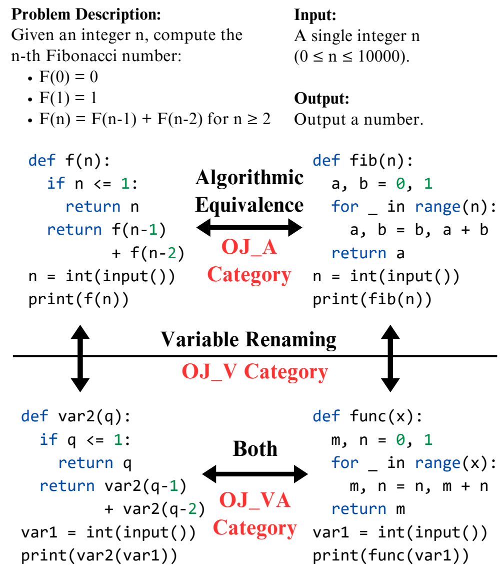

Figure 5: **Equivalent pairs from the OJ_A, OJ_V, OJ_VA categories in EquiBench.** OJ_A pairs demonstrate *algorithmic equivalence*, OJ_V pairs involve *variable renaming* transformations, and OJ_VA pairs combine *both* types of variations.

Dataset Generation. We sample Python submissions using a publicly available dataset from Online Judge (OJ) [52]. For OJ_A pairs, accepted submissions are treated as equivalent, while pairs consisting of an accepted submission and a wrong-answer submission are considered inequivalent. Variable renaming transformations are automated with an open-source tool [53].

4 Experimental Setup

Dataset. EquiBench consists of 2,400 program pairs across six equivalence categories, each with 200 equivalent and 200 inequivalent pairs. Table 1 summarizes the statistics of program lengths. Constructing the program pairs required substantial systems effort. For example, for the DCE category, we developed a 2,917-line LLVM-based tool, including 1,472 lines in C and C++, with alias analysis and path feasibility analysis to accurately classify live and dead code. The complexity of EquiBench should not be underestimated.

Category	Language	# Pairs	Lines of Code		
			Min	Max	Avg.
DCE	C	400	98	880	541
CUDA	CUDA	400	46	1733	437
x86-64	x86-64	400	8	29	14
OJ_A	Python	400	3	3403	82
OJ_V	Python	400	2	4087	70
OJ_VA	Python	400	3	744	35

Table 1: **Statistics of the EquiBench dataset.**

Prompts. The 0-shot evaluation is conducted using the prompt "You are here to judge if two programs are semantically equivalent. Here equivalence means {*definition*}. [Program 1]: {code1} [Program 2]: {code2} Please only output the answer of whether the two programs are equivalent or not. You should only output Yes or No." The definition of equivalence and the corresponding program pairs are provided for each category. Additionally, for the categories of OJ_A, OJ_V, and OJ_VA, the prompt also includes the problem description. The full prompts used in our experiments for each equivalence category are in Appendix A.4.

Model	DCE	CUDA	x86-64	OJ_A	OJ_V	OJ_VA	Overall Accuracy
Random Baseline	*50.0*	*50.0*	*50.0*	*50.0*	*50.0*	*50.0*	*50.0*
Llama-3.2-3B-Instruct-Turbo	50.0	49.8	50.0	51.5	51.5	51.5	50.7
Llama-3.1-8B-Instruct-Turbo	41.8	49.8	50.5	57.5	75.5	56.8	55.3
Mistral-7B-Instruct-v0.3	51.0	57.2	73.8	50.7	50.5	50.2	55.6
Mixtral-8x7B-Instruct-v0.1	50.2	47.0	64.2	59.0	61.5	55.0	56.1
Mixtral-8x22B-Instruct-v0.1	46.8	49.0	62.7	63.5	76.0	62.7	60.1
Llama-3.1-70B-Instruct-Turbo	47.5	50.0	58.5	66.2	72.0	67.5	60.3
QwQ-32B-Preview	48.2	50.5	62.7	65.2	71.2	64.2	60.3
Qwen2.5-7B-Instruct-Turbo	50.5	49.2	58.0	62.0	80.8	63.0	60.6
gpt-4o-mini-2024-07-18	46.8	50.2	56.8	64.5	91.2	64.0	62.2
Qwen2.5-72B-Instruct-Turbo	42.8	56.0	64.8	72.0	76.5	70.8	63.8
Llama-3.1-405B-Instruct-Turbo	40.0	49.0	75.0	72.2	74.5	72.8	63.9
DeepSeek-V3	41.0	50.7	69.2	73.0	83.5	72.5	65.0
gpt-4o-2024-11-20	43.2	49.5	65.2	71.0	87.0	73.8	65.0
claude3.5-sonnet-2024-10-22	38.5	62.3	70.0	71.2	78.0	73.5	65.6
claude3.7-sonnet-2025-04-16	40.5	**63.8**	64.8	70.5	89.2	73.5	67.0
o1-mini-2024-09-12	55.8	50.7	74.2	80.0	89.8	78.8	71.5
DeepSeek-R1	52.2	61.0	78.2	79.8	91.5	78.0	73.5
o3-mini-2025-01-31	68.8	59.0	**84.5**	84.2	88.2	83.2	78.0
o4-mini-2025-04-16	**76.2**	60.8	83.0	**89.0**	**96.5**	**88.5**	**82.3**
Mean	49.0	53.4	66.7	68.6	78.1	68.5	64.0

Table 2: **Accuracy of 19 models on EquiBench under 0-shot prompting.** We report accuracy for each of the six equivalence categories along with the overall accuracy.

5 Results

5.1 Model Accuracy

Table 2 shows the accuracy results for 19 state-of-the-art large language models on EquiBench under zero-shot prompting. Our findings are as follows:

Reasoning models achieve the highest performance. As shown in Table 2, reasoning models such as OpenAI o3-mini, DeepSeek R1, and o1-mini significantly outperform all others in our evaluation. This further underscores the complexity of equivalence checking, where reasoning models exhibit a distinct advantage.

EquiBench is a challenging benchmark. Among the 19 models evaluated, OpenAI o4-mini achieves only 60.8% in the CUDA category despite being the top-performing model overall, with an accuracy of 82.3%. For the two most difficult categories, the highest accuracy across all models is 63.8% and 76.2%, respectively, only modestly above the random baseline of 50% accuracy for binary classification, highlighting the substantial room for improvement.

Scaling up models improves performance. Larger models generally achieve better performance. Figure 6 shows scaling trends for the Qwen2.5, Llama-3.1, and Mixtral families, where accuracy improves with model size. The x-axis is on a logarithmic scale, highlighting how models exhibit consistent gains as parameters increase.

5.2 Difficulty Analysis

We conduct a detailed difficulty analysis across equivalence categories and study how syntactic similarity influences model predictions.

Difficulty by Transformation Type. Across categories, we find that purely syntactic transformations are substantially easier for models, while structural and compiler-involved transformations are much harder. Specifically, **OJ_V** (variable renaming) achieves the highest mean accuracy of 78.1%, as it only requires surface-level reasoning. **OJ_A** (algorithmic equivalence) and **OJ_VA** (variable renaming combined with algorithmic differences) achieve similar accuracies of 68.6% and 68.5%, respectively. In contrast, **x86-64** (66.7%) and **CUDA** (53.4%) involve complex instruction-level or memory-level transformations, requiring deeper semantic reasoning. **DCE** (dead code elimination) is the most difficult category, with a mean accuracy of 49.0%, suggesting that models struggle with nuanced program analysis concepts. We also observe biases in how models behave across certain categories, which we evaluate further in Appendix A.1.

Difficulty by Syntactic Similarity. To assess whether LLM predictions reflect understanding of execution semantics rather than mere reliance on surface-level syntax, we analyze how syntactic similarity affects model behavior. Using Moss [54], a plagiarism detection tool, we observe the following:

- For program pairs with low syntactic similarity, models tend to predict "inequivalent," even when the programs are semantically equivalent. This suggests an overreliance on the superficial form of the code.
- For syntactically similar pairs, models are more likely to predict "equivalent," indicating a tendency to associate similarity in form with equivalence in execution semantics.

We validate this trend through statistical testing: at significance level $(\alpha = 0.05)$, model accuracy on equivalent pairs increases with syntactic similarity, while accuracy on inequivalent pairs decreases. This disconnect between syntactic form and execution behavior, as discussed in Section 1, suggests that models are not grounded in program semantics.

Implications for Benchmark Design. These findings suggest that future benchmarks should emphasize *syntactically dissimilar yet equivalent program pairs* and *syntactically similar yet inequivalent program pairs* to create more challenging and diagnostic benchmarks for evaluating the deep semantic reasoning capabilities of LLMs.

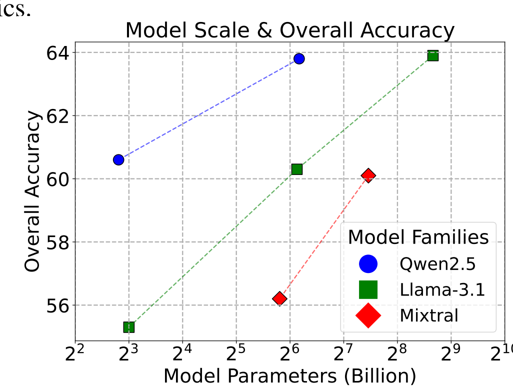

Figure 6: **Scaling Trend on EquiBench.**

5.3 Prompting Strategies Analysis

We study few-shot in-context learning and Chain-of-Thought (CoT) prompting, evaluating four strategies: 0-shot, 4-shot, 0-shot with CoT, and 4-shot with CoT. For 4-shot, prompts include 2 equivalent and 2 inequivalent pairs. Table 3 shows the results.

Our key finding is that **prompting strategies *barely* improve performance on EquiBench**, highlighting the difficulty in understanding code execution semantics.

6 Discussion and Future Directions

Scope and Positioning Machine learning has been applied to many code-related tasks, such as clone detection [55], code search [56], and bug finding [57]. EquiBench focuses on equivalence checking, which differs fundamentally by evaluating a model's understanding of execution semantics. Unlike natural language, code is executable, and its correctness depends on execution results rather than form. For example, clone detection captures syntactic or structural similarity without considering behavior. In contrast, EquiBench tests whether two programs produce the same outputs for all inputs, offering a stricter and more informative benchmark for reasoning about code execution.

Model	0S	4S	0S-CoT	4S-CoT
o1-mini	71.5	71.5	**71.9**	**71.9**
gpt-4o	65.0	**66.5**	62.5	62.7
DeepSeek-V3	65.0	**66.9**	63.3	62.5
gpt-4o-mini	62.2	**63.5**	60.2	61.2

Table 3: **Accuracies of different prompting techniques.** We evaluate 0-shot and 4-shot in-context learning, both without and with Chain-of-Thought (CoT). Prompting strategies barely improve performance.

Labeling Soundness To ensure high-assurance equivalence labels, EquiBench relies on transformations grounded in program analysis, compiler scheduling, and superoptimization, all of which offer strong soundness guarantees. In contrast, approaches such as random testing [58], similarity-based tools [59], and refactoring datasets lack formal guarantees and risk introducing incorrect labels.

Modularity and Extensibility EquiBench is designed with modularity in mind, with each equivalence category representing a distinct class of transformations. The dataset is extensible and can be expanded to include additional categories in future work. As the first benchmark in this space, EquiBench provides a foundation for systematic evaluation and further development.

Evaluation of Reasoning Trace While our evaluation centers on binary classification, understanding the rationale behind model predictions is an important direction. Explanations may take the form of natural language or formal proofs, but verifying their correctness remains difficult. Natural language lacks reliable automated validation, since using LLMs as judges can produce unsound results. Building a proof-based evaluation framework using tools such as Lean is also highly nontrivial. We present a manual case analysis of reasoning trace correctness in Appendix A.2 and leave automated robust evaluation of reasoning as future work.

Implications for Model Training Our results suggest that current models have limited understanding of code execution semantics. To improve this, future work may incorporate *program execution traces* [24, 23] into training, enabling models to learn execution behavior more directly rather than relying on next-token prediction over surface-level syntax.

7 Conclusion

We introduced EquiBench, a benchmark for evaluating whether large language models (LLMs) truly understand code execution semantics. We propose the task of equivalence checking, which asks whether two programs produce identical outputs for all possible inputs, as a direct way to test a model's ability to reason about program behavior. The dataset consists of 2400 program pairs across four languages and six categories, constructed through a fully automated pipeline that provides high-confidence labels and nontrivial difficulty. Our evaluation of 19 state-of-the-art LLMs shows that even the best-performing models achieve only modest accuracy in the most challenging categories. Further analysis shows that LLMs often rely on syntactic similarity instead of demonstrating robust reasoning about code execution semantics, underscoring the need for further advances in semantic understanding of programs.

References

[1] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, "Livecodebench: Holistic and contamination free evaluation of large language models for code," *arXiv preprint arXiv:2403.07974*, 2024.

[2] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "Whitefox: Whitebox compiler fuzzing empowered by large language models," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 709–735, 2024.

[3] C. Yang, Z. Zhao, and L. Zhang, "Kernelgpt: Enhanced kernel fuzzing via large language models," *arXiv preprint arXiv:2401.00563*, 2023.

[4] E. M. Bender and A. Koller, "Climbing towards nlu: On meaning, form, and understanding in the age of data," in *Proceedings of the 58th annual meeting of the association for computational linguistics*, 2020, pp. 5185–5198.

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

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

[7] S. Badihi, Y. Li, and J. Rubin, "Eqbench: A dataset of equivalent and non-equivalent program pairs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 610–614.

[8] N. Maveli, A. Vergari, and S. B. Cohen, "What can large language models capture about code functional equivalence?" *arXiv preprint arXiv:2408.11081*, 2024.

[9] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano *et al.*, "Training verifiers to solve math word problems," *arXiv preprint arXiv:2110.14168*, 2021.

[10] J. Huang and K. C.-C. Chang, "Towards reasoning in large language models: A survey," *arXiv preprint arXiv:2212.10403*, 2022.

[11] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.

[12] I. Mirzadeh, K. Alizadeh, H. Shahrokhi, O. Tuzel, S. Bengio, and M. Farajtabar, "Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models," *arXiv preprint arXiv:2410.05229*, 2024.

[13] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le *et al.*, "Least-to-most prompting enables complex reasoning in large language models," *arXiv preprint arXiv:2205.10625*, 2022.

[14] N. Ho, L. Schmid, and S.-Y. Yun, "Large language models are reasoning teachers," *arXiv preprint arXiv:2212.10071*, 2022.

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

[16] L. Chen, B. Li, S. Shen, J. Yang, C. Li, K. Keutzer, T. Darrell, and Z. Liu, "Large language models are visual reasoning coordinators," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[17] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord, "Think you have solved question answering? try arc, the ai2 reasoning challenge," *arXiv preprint arXiv:1803.05457*, 2018.

[18] D. Zhang, C. Tigges, Z. Zhang, S. Biderman, M. Raginsky, and T. Ringer, "Transformer-based models are not yet perfect at learning to emulate structural recursion," *arXiv preprint arXiv:2401.12947*, 2024.

[19] A. Gu, B. Rozière, H. Leather, A. Solar-Lezama, G. Synnaeve, and S. I. Wang, "Cruxeval: A benchmark for code reasoning, understanding and execution," *arXiv preprint arXiv:2401.03065*, 2024.

[20] C. Liu, S. D. Zhang, A. R. Ibrahimzada, and R. Jabbarvand, "Codemind: A framework to challenge large language models for code reasoning," *arXiv preprint arXiv:2402.09664*, 2024.

[21] E. La Malfa, C. Weinhuber, O. Torre, F. Lin, S. Marro, A. Cohn, N. Shadbolt, and M. Wooldridge, "Code simulation challenges for large language models," *arXiv preprint arXiv:2401.09074*, 2024.

[22] C. Liu, S. Lu, W. Chen, D. Jiang, A. Svyatkovskiy, S. Fu, N. Sundaresan, and N. Duan, "Code execution with pre-trained language models," *arXiv preprint arXiv:2305.05383*, 2023.

[23] A. Ni, M. Allamanis, A. Cohan, Y. Deng, K. Shi, C. Sutton, and P. Yin, "Next: Teaching large language models to reason about code execution," *arXiv preprint arXiv:2404.14662*, 2024.

[24] Y. Ding, J. Peng, M. J. Min, G. Kaiser, J. Yang, and B. Ray, "Semcoder: Training code language models with comprehensive semantics reasoning," *arXiv preprint arXiv:2406.01006*, 2024.

[25] M. Chen, G. Li, L.-I. Wu, and R. Liu, "Dce-llm: Dead code elimination with large language models," in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2025, pp. 9942–9955.

[26] A. Shypula, A. Madaan, Y. Zeng, U. Alon, J. Gardner, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh, "Learning performance-improving code edits," *arXiv preprint arXiv:2302.07867*, 2023.

[27] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve *et al.*, "Large language models for compiler optimization," *arXiv preprint arXiv:2309.07062*, 2023.

[28] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, and H. Leather, "Meta large language model compiler: Foundation models of compiler optimization," *arXiv preprint arXiv:2407.02524*, 2024.

[29] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. GONG, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. LIU, "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. [Online]. Available: https://openreview.net/forum?id=6lE4dQXaUcb

[30] Z. Yang, F. Liu, Z. Yu, J. W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, and G. Li, "Exploring and unleashing the power of large language models in automated code translation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1585–1608, 2024.

[31] A. R. Ibrahimzada, K. Ke, M. Pawagi, M. S. Abid, R. Pan, S. Sinha, and R. Jabbarvand, "Repository-level compositional code translation and validation," *arXiv preprint arXiv:2410.24117*, 2024.

[32] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[33] S. Pailoor, Y. Wang, and I. Dillig, "Semantic code refactoring for abstract data types," *Proceedings of the ACM on Programming Languages*, vol. 8, no. POPL, pp. 816–847, 2024.

[34] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, "Automating regression verification," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 349–360.

[35] Z. Tian, H. Shu, D. Wang, X. Cao, Y. Kamei, and J. Chen, "Large language models for equivalent mutant detection: How far are we?" in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1733–1745.

[36] R. Sharma, E. Schkufza, B. Churchill, and A. Aiken, "Data-driven equivalence checking," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 391–406.

[37] M. Dahiya and S. Bansal, "Black-box equivalence checking across compiler optimizations," in *Asian Symposium on Programming Languages and Systems*. Springer, 2017, pp. 127–147.

[38] S. Gupta, A. Saxena, A. Mahajan, and S. Bansal, "Effective use of smt solvers for program equivalence checking through invariant-sketching and query-decomposition," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2018, pp. 365–382.

[39] F. Mora, Y. Li, J. Rubin, and M. Chechik, "Client-specific equivalence checking," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 441–451.

[40] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1027–1040.

[41] S. Badihi, F. Akinotcho, Y. Li, and J. Rubin, "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*, 2020, pp. 13–24.

[42] F. Zhao, L. Lim, I. Ahmad, D. Agrawal, and A. E. Abbadi, "Llm-sql-solver: Can llms determine sql equivalence?" *arXiv preprint arXiv:2312.10321*, 2023.

[43] H. Ding, Z. Wang, Y. Yang, D. Zhang, Z. Xu, H. Chen, R. Piskac, and J. Li, "Proving query equivalence using linear integer arithmetic," *Proceedings of the ACM on Management of Data*, vol. 1, no. 4, pp. 1–26, 2023.

[44] R. Singh and S. Bedathur, "Exploring the use of llms for sql equivalence checking," *arXiv preprint arXiv:2412.05561*, 2024.

[45] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[46] T. Theodoridis, M. Rigger, and Z. Su, "Finding missed optimizations through the lens of dead code elimination," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 697–709.

[47] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[48] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[49] L. Zheng, R. Liu, J. Shao, T. Chen, J. E. Gonzalez, I. Stoica, and A. H. Ali, "Tenset: A large-scale program performance dataset for learned tensor compilers," in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[50] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 305–316, 2013.

[51] J. R. Koenig, O. Padon, and A. Aiken, "Adaptive restarts for stochastic synthesis," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 696–709.

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

[53] D. Flook, "Python variable renaming tool," 2025, https://github.com/dflook/python-minifier.

[54] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 76–85.

[55] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 87–98.

[56] Z. Gao, H. Wang, Y. Wang, and C. Zhang, "Virtual compiler is all you need for assembly code search," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024, pp. 3040–3051.

[57] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.

[58] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 81–92.

[59] D. Silva and M. T. Valente, "Refdiff: Detecting refactorings in version histories," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 269–279.

A Appendix

A.1 Bias in Model Prediction

We evaluate the prediction bias of the models and observe **a pronounced tendency to misclassify equivalent programs as inequivalent in the CUDA and x86-64 categories**. Table A1 presents the results for four representative models, showing high accuracy for inequivalent pairs but significantly lower accuracy for equivalent pairs, with full results for all models in Appendix A.3.

The bias in the CUDA category arises from extensive structural transformations, such as loop restructuring and shared memory optimizations, which make paired programs appear substantially different. In the x86-64 category, superoptimization applies non-local transformations to achieve optimal instruction sequences, introducing aggressive code restructuring that complicates equivalence reasoning and leads models to misclassify equivalent pairs as inequivalent frequently.

Model	**CUDA**		**x86-64**	
	Eq	**Ineq**	**Eq**	**Ineq**
Random Baseline	*50.0*	*50.0*	*50.0*	*50.0*
o3-mini	27.5	90.5	69.5	99.5
o1-mini	2.5	99.0	50.0	98.5
DeepSeek-R1	28.0	94.0	57.5	99.0
DeepSeek-V3	8.5	93.0	44.0	94.5

Table A1: Accuracies on equivalent and inequivalent pairs in the CUDA and x86-64 categories under 0-shot prompting, showing that **models perform significantly better on inequivalent pairs**. Random guessing serves as an unbiased baseline for comparison. Full results for all models are shown in Appendix A.3.

A.2 Case Studies

Models lack capabilities for sound equivalence checking. We find that simple changes that lead to semantic differences can confuse the models, causing them to produce incorrect predictions despite their correct predictions on the original program pairs. For example, o3-mini, which is one of the top-performing models in CUDA category, can correctly classify the pair shown in Figure 3 as equivalent. Next, we introduce synchronization bugs into the right-hand program, creating two inequivalent pairs with the original left-hand program: (1) removing the first `__syncthreads();` allows reads before all writes complete, causing race conditions; (2) removing the second `__syncthreads();` lets faster threads overwrite shared data while slower threads read it. Despite these semantic differences, o3-mini misclassifies both pairs as equivalent.

Proper hints enable models to correct misjudgments. After o3-mini misclassifies the modified pairs, a hint about removed synchronization primitives allows it to correctly identify both as inequivalent, with accurate explanations highlighting data races. This suggests that training models on dedicated program analysis datasets, beyond only raw source code, may be useful for improving their code reasoning capabilities.

A.3 Model Prediction Bias

We evaluate the prediction bias of the models and observe a pronounced tendency to misclassify equivalent programs as inequivalent in the CUDA and x86-64 categories. Table A2 here shows the full results on all models under 0-shot prompting.

Model	CUDA		x86-64	
	Eq	Ineq	Eq	Ineq
Random Baseline	*50.0*	*50.0*	*50.0*	*50.0*
deepseek-ai/DeepSeek-V3	8.5	93.0	44.0	94.5
deepseek-ai/DeepSeek-R1	28.0	94.0	57.5	99.0
meta-llama/Llama-3.1-405B-Instruct-Turbo	6.0	92.0	68.5	81.5
meta-llama/Llama-3.1-8B-Instruct-Turbo	2.0	97.5	1.0	100.0
meta-llama/Llama-3.1-70B-Instruct-Turbo	7.0	93.0	27.5	89.5
meta-llama/Llama-3.2-3B-Instruct-Turbo	0.0	99.5	0.0	100.0
anthropic/claude-3-5-sonnet-20241022	62.5	62.0	49.5	90.5
Qwen/Qwen2.5-7B-Instruct-Turbo	18.5	80.0	17.5	98.5
Qwen/Qwen2.5-72B-Instruct-Turbo	14.5	97.5	36.0	93.5
Qwen/QwQ-32B-Preview	35.0	66.0	39.0	86.5
mistralai/Mixtral-8x7B-Instruct-v0.1	18.0	76.0	50.5	78.0
mistralai/Mixtral-8x22B-Instruct-v0.1	10.5	87.5	32.5	93.0
mistralai/Mistral-7B-Instruct-v0.3	52.5	62.0	87.0	60.5
openai/gpt-4o-mini-2024-07-18	0.5	100.0	16.5	97.0
openai/gpt-4o-2024-11-20	0.0	99.0	68.5	62.0
openai/o3-mini-2025-01-31	27.5	90.5	69.5	99.5
openai/o1-mini-2024-09-12	2.5	99.0	50.0	98.5

Table A2: Model prediction bias.

A.4 Prompts

A.4.1 DCE Category

We show the prompts for the 0-shot setting.

You are here to judge if two C programs are semantically equivalent.
Here equivalence means that, when run on the same input, the two programs always have the same program state at all corresponding points reachable by program execution.
[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.

A.4.2 CUDA Category

We show the prompts for the 0-shot setting.

You are here to judge if two CUDA programs are semantically equivalent.
Here equivalence means that, when run on the same valid input, the two programs always compute the same mathematical output (neglecting floating point rounding errors).
[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.

A.4.3 x86-64 Category

We show the prompts for the 0-shot setting.

You are here to judge if two x86-64 programs are semantically equivalent.
Here equivalence means that, given any input bits in the register `{def_in}`, the two programs always have the same bits in register `{live_out}`. Differences in other registers do not matter for equivalence checking.

[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.

A.4.4 OJ_A, OJ_V, OJ_VA Category

We show the prompts for the 0-shot setting.

You are here to judge if two Python programs are semantically equivalent.
You will be given [Problem Description], [Program 1] and [Program 2].
Here equivalence means that, given any valid input under the problem description, the two programs will always give the same output.

[Problem Description]:

```
{problem_html}
```

[Program 1]:

```
{program_1_code}
```

[Program 2]:

```
{program_2_code}
```

Please only output the answer of whether the two programs are equivalent or not. You should only output YES or NO.