

The Path Not Taken: Duality in Reasoning about Program Execution

Eshgin Hasanov, Md Mahadi Hassan Sibat, Santu Karmaker, Aashish Yadavally

Department of Computer Science, University of Central Florida

Correspondence: eshgin.hasanov@ucf.edu, aashish.yadavally@ucf.edu

Abstract

Large language models (LLMs) have shown remarkable capabilities across diverse coding tasks. However, their adoption requires a true understanding of program execution rather than relying on surface-level patterns. Existing benchmarks primarily focus on predicting program properties tied to specific inputs (*e.g.*, code coverage, program outputs). As a result, they provide a narrow view of dynamic code reasoning and are prone to data contamination. We argue that understanding program execution requires evaluating its inherent duality through two complementary reasoning tasks: (*i*) predicting a program’s observed behavior for a given input, and (*ii*) inferring how the input must be mutated toward a specific behavioral objective. Both tasks jointly probe a model’s causal understanding of execution flow. We instantiate this duality in DEXBENCH, a benchmark comprising 445 paired instances, and evaluate 13 LLMs. Our results demonstrate that dual-path reasoning provides a robust and discriminative proxy for dynamic code understanding.

1 Introduction

Large language models (LLMs) are increasingly being adopted in the field of software engineering (SE), from writing code and unit tests to finding and fixing bugs. However, their performance remains inconsistent: these models can solve complex coding problems (Jimenez et al., 2024), yet still fail to reason about basic loops (Dhulipala et al., 2025). This can possibly be due to mimicking surface-level patterns without a *true* understanding of the program, which inherently limits their applicability in real-world settings. As a result, recent efforts have focused on a fine-grained evaluation of the reasoning capabilities of coding LLMs (Liu et al., 2024; Chen et al., 2025; Liu et al., 2025), moving beyond end-to-end evaluations on downstream tasks such as program synthesis (Chen et al., 2021; Gu et al., 2024) or repair (Jimenez et al., 2024).

In practice, such reasoning evaluations primarily rely on *executability*, a distinct feature of source code requiring accurate modeling of execution flow. These assess how well LLMs understand program behaviors during execution, including predicting code coverage (Dhulipala et al., 2025; Chen et al., 2025), input-output mappings (Ding et al., 2024; Chen et al., 2025), as well as tracking intermediate updates to program state and subsequent execution paths (Liu et al., 2023; Chen et al., 2025).

Nevertheless, current program execution-based benchmarks consider runtime behaviors under only a single test case, *i.e.*, along *one* execution path (*e.g.*, predicting the statements \mathcal{S} in a program P that are executed for an input I). However, a given program may traverse many different paths depending on its inputs. Therefore, single-test evaluations offer a narrow view of the models’ program understanding. Moreover, since benchmarks defined by fixed sets of input-output pairs can be memorized during training (Riddell et al., 2024), they are susceptible to data contamination, a growing concern in modern LLM evaluation.

To address this gap, we introduce the concept of **duality in reasoning about program execution**. The key idea is based on the argument that two program paths share a common execution space and diverge only at branching points where the current program state determines diverging execution flows. We refer to the actual path taken for a given input as *execution path* and the alternative as *counterfactual path*. We operationalize our framework by designing a pair of complementary reasoning tasks: one predicting a program property observed along the execution path (*forward reasoning*); other determining the input required for counterfactual path to be taken (*backward reasoning*).

Note that a program naturally exhibits *n*-ality (or “multiality”), characterized by many possible pairs of execution and counterfactual paths across its input domain. This property can be viewed as

a composition of discrete dual pairs. By selecting one such pair for evaluation, we can isolate and evaluate the causal logic a model employs to reason about program execution. Accordingly, we posit that probing a model’s ability to understand both *how* the observed execution occurs and under *what* conditions (e.g., intermediate program state) the execution would instead flow through the counterfactual path provides a more robust evaluation of reasoning capability in coding LLMs.

As an illustration, consider the target program in Figure 1. The program accepts multiple paths, including $\pi^{(1)} : 3 \rightarrow \dots \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow \dots \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 5 \rightarrow 12 \rightarrow [\text{END}]$, $\pi^{(2)} : 3 \rightarrow \dots \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow [\text{END}]$, $\pi^{(3)} : 3 \rightarrow \dots \rightarrow 6 \rightarrow 8 \rightarrow \dots \rightarrow 11 \rightarrow 5 \rightarrow \dots \rightarrow [\text{END}]$, among others. Any pair $\langle \pi^{(i)}, \pi^{(j)} \rangle$ reflects this multiality. For the test input “ua6hajq”, execution follows the control-flow path $\pi^{(1)}$. Here, we designate $\pi^{(1)}$ as the *execution path* π_{exec} and select $\pi^{(2)}$ as the *counterfactual path* π_{cf} to illustrate the proposed duality (highlighted in Figure 1 in green and red, respectively)¹. In particular, both π_{exec} and π_{cf} share execution space $3 \rightarrow \dots \rightarrow 6$, diverging only at line 6 depending on the branch condition. When it evaluates to “No”, the execution proceeds along π_{exec} ; otherwise, π_{cf} is taken.

By design, forward reasoning evaluates an LLM’s ability to predict program properties along π_{exec} for a given input I_{exec} , while backward reasoning evaluates whether it can infer the new input I_{cf} that I_{exec} should be mutated to such that execution now flows along π_{cf} instead. Notably, the novelty in the notion of duality lies in its focus on the causal relationships in program execution that induce divergence from π_{exec} to π_{cf} , rather than in reasoning about program properties along distinct paths in isolation (Liu et al., 2025).

Accordingly, we propose DEXBENCH, a **dual-path, program execution-based benchmark** that jointly investigates both forward execution reasoning and backward counterfactual reasoning. DEXBENCH is built from real-world programs that cover a range of complexity, extracted from widely used HumanEval (Chen et al., 2021), CruxEval (Gu et al., 2024), and PythonSaga (Yadav et al., 2024). We evaluate 9 open-source and 4 closed-source/proprietary LLMs, spanning both reasoning and general-purpose models across small- to large-scale model sizes. Our experiments reveal that:

¹In Section 3.2, we describe the algorithm used to select the counterfactual path, and in Section 4.3, we benchmark LLMs with different counterfactual path selection strategies.

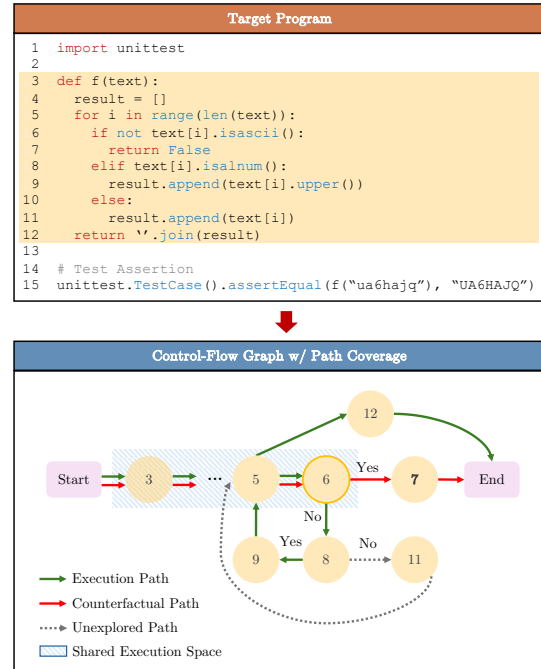


Figure 1: DEXBENCH exploits the shared execution space between execution and counterfactual paths to enable a robust evaluation of LLMs in program execution.

(i) strong performance on execution or counterfactual reasoning in isolation does not translate to success under joint evaluation, highlighting the limitations of single-path evaluation benchmarks.

(ii) dual-path reasoning provides a reliable proxy for causal, state-aware reasoning of execution flow.

(iii) parameter scaling and reasoning-focused post-training do not guarantee improvements in dual-path reasoning, as seen with some mid-sized and general-purpose models that outperformed larger or explicitly reasoning-focused counterparts.

Together, our findings indicate that DEXBENCH offers a systematic and extensible framework to evaluate dynamic code understanding in LLMs. All code and data are publicly available.²

2 Duality of Reasoning

We formalize the concept of duality in reasoning about program execution as follows.

2.1 (Forward) Execution Reasoning

Consider a program P with test input $I_{exec} \in \mathcal{I}$. Executing P on I_{exec} induces an execution trace

$$\tau_{exec} = (\langle s^{(1)}, \sigma^{(1)} \rangle \dots \langle s^{(|\tau_{exec}|)}, \sigma^{(|\tau_{exec}|)} \rangle) \in \mathcal{T},$$

where each $s^{(t)}$ is a statement in P executed at step t , and $\sigma^{(t)}$ is the corresponding program state,

²<https://github.com/sail-ucf/dexbench>

defined as a mapping from program variables to their values. Let $\phi : \mathcal{T} \rightarrow \mathcal{O}$ denotes the observable program properties or runtime behaviors derived from execution traces (e.g., program output, code coverage, or a post-condition over variables).

For program P and input I_{exec} , we refer to the task of predicting an observable behavior $\phi(\tau_{exec})$ as *forward or execution reasoning*. Formally,

$$\mathcal{R}_{exec} : (P, I_{exec}) \mapsto \phi(\tau_{exec}). \quad (1)$$

By design, this formulation captures a model’s ability to maintain a representation of the program state $\sigma^{(i)}$ and to update it according to the semantics of each executed statement $s^{(i)}$. The difficulty of execution reasoning typically scales with the length of the trace $|\tau_{exec}|$ as well as with the complexity of state transitions, such as those induced by non-trivial control flow (Liu et al., 2023).

2.2 (Backward) Counterfactual Reasoning

Let *counterfactual objective* $\phi^* \in \mathcal{O}$ be a *desired* observable behavior that is obtained by applying a transformation $f : \mathcal{O} \rightarrow \mathcal{O}$ to the original behavior induced by I_{exec} such that $\phi^* = f(\phi(\tau_{exec}))$.

For program P , original input I_{exec} and a given counterfactual objective ϕ^* , we define *backward or counterfactual reasoning* as the task of mutating I_{exec} into an alternative input $I_{cf} \in \mathcal{I}$ such that executing P on I_{cf} produces an execution trace τ_{cf} satisfying $\phi(\tau_{cf}) = \phi^*$. Formally,

$$\mathcal{R}_{cf} : (P, I_{exec}, \phi^*) \mapsto I_{cf}. \quad (2)$$

This formulation captures the inverse capability of a model, i.e., to reason backwards from a desired effect to the conditions that would produce it. In contrast to execution reasoning, which follows the program’s semantics forward from a given input, counterfactual reasoning requires analyzing how changes to the original input affect control flow and program state in order to realize a behavioral goal (e.g., execution of an otherwise uncovered branch, specific program output). The difficulty of counterfactual reasoning typically increases with the complexity of the program’s branching structure and the degree to which the counterfactual execution diverges from the original one.

2.3 Dual-Path Reasoning

We define *dual-path reasoning* \mathcal{R}_{dual} as the task of jointly reasoning over the original execution path π_{exec} and a counterfactual path π_{cf} . Formally,

$$\mathcal{R}_{dual} : \mathcal{R}_{exec} \oplus \mathcal{R}_{cf} \quad (3)$$

where “ \oplus ” denotes the composition of execution and counterfactual reasoning. It probes a model’s ability to: (i) maintain an observable behavior produced along π_{exec} , and (ii) manipulate runtime behavior to identify an alternative input I_{cf} that satisfies a counterfactual objective ϕ^* , defined relative to the original execution. Such a unified perspective is particularly important for SE tasks that require a systematic exploration of program behavior (e.g., coverage-guided fuzzing, debugging, etc.), where understanding how and why executions diverge is as important as predicting their outcomes. Furthermore, dual-path reasoning enables a robust evaluation of coding LLMs as it demands a deeper, more holistic understanding of program execution than reasoning on single, isolated paths.

3 DEXBENCH

3.1 Problem Definition

To evaluate the dual-path reasoning capabilities of LLMs, we instantiate the formulations for execution and counterfactual reasoning (Eqs. 1 and 2) as concrete, verifiable tasks: *code coverage prediction* and *branch-targeted input mutation*, respectively. For execution reasoning, the observable behavior that a model must predict is *statement coverage*, i.e., the set of line numbers $\phi(\tau_{exec})$ executed by a program under input I_{exec} . In the case of the latter, we define the counterfactual objective ϕ^* as the *reachability of a specific branch b that remains uncovered under I_{exec}* . Here, the model must generate a mutated input I_{cf} that redirects the program’s execution to flow through b such that $b \in \phi(\tau_{cf})$.

Unlike coverage-guided fuzzing (Böhme et al., 2017), which relies on random mutations and execution feedback, DEXBENCH requires models to identify necessary input changes through reasoning alone. By grounding both execution and counterfactual prediction in a shared execution context, we evaluate whether models maintain a consistent causal representation of program behavior. This prevents the evaluation from treating execution and generation as decoupled tasks, instead probing the model’s ability to logically bridge observed runtime behaviors with desired behavioral outcomes.

Note that our proposed framework can also be extended to other evaluation settings. For instance, execution reasoning may target predicting program outputs or intermediate states, while counterfactual reasoning may require predicting inputs that trigger specific program outputs or state transitions.

3.2 Benchmark Construction

Source Programs. DEXBENCH is constructed from a diverse collection of Python programs from three popular datasets: *CruxEval* (Gu et al., 2024), *HumanEval* (Chen et al., 2021), and *PythonSaga* (Yadav et al., 2024). These are designed to evaluate program understanding and execution reasoning, making them well-suited for input-dependent reasoning over program executions.

CruxEval contains short programs with relatively simple control flow. In contrast, HumanEval has moderate control-flow complexity, while PythonSaga includes programs with deeper nesting, recursion, and more advanced programming constructs. In Figure 2, we illustrate the program complexity for each dataset with different structural metrics. Based on this, we only retained programs that contain: (i) at least one conditional or loop, and (ii) a test case that yields partial code coverage (*i.e.*, less than 100%). The first criterion ensures reasoning over non-trivial control flow, while the second enables coverage-guided input mutation as required in the counterfactual or backward reasoning task.

Collecting Execution Information. For each retained program, we randomly selected a test case meeting the above criteria to instantiate the execution or forward reasoning task, collecting statement coverage information by monitoring runtime behavior with Slipcover (Pizzorno and Berger, 2023). Here, coverage information serves as *ground-truth observable behavior* (denoted by ϕ in Section 2.1).

Selecting Counterfactual Paths. As illustrated in Figure 1, the execution of a program under input I_{exec} flows along a single path, leaving multiple alternative paths unexplored that could serve as potential counterfactual targets. Among these, we select an uncovered branch b under input I_{exec} that, if reached, would enhance the execution diversity between the original execution path π_{exec} and the counterfactual path π_{cf} while resulting in an increased code coverage (see Appendix B for detailed algorithm). Here, reaching branch b thus serves as the counterfactual objective (denoted by ϕ^* in Section 2.2). In this setting, the *ground-truth counterfactual observable behavior* is defined by the program’s execution semantics: a model-generated input I_{cf} is considered correct if and only if its induced execution trace τ_{cf} satisfies $b \in \phi(\tau_{cf})$.

Benchmark Statistics. In total, DEXBENCH comprises 445 unique evaluation instances, each

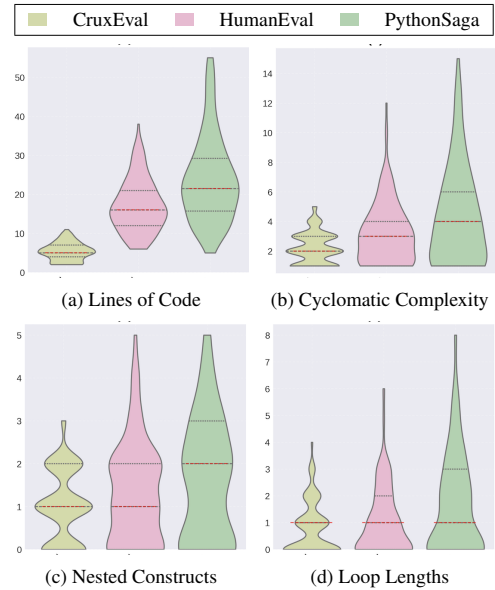


Figure 2: Distribution of different program complexity metrics in CruxEval, HumanEval, and PythonSaga.

defined by a pair of forward and backward reasoning tasks (298 from CruxEval, 100 from HumanEval, and 47 from PythonSaga). It exhibits significant structural diversity: subject programs have 10 to 78 lines of code and a mean cyclomatic complexity of 3.63 ± 2.18 (ranging between 2 and 19), thus yielding a challenging testbed for evaluating models’ ability to reason about program execution.

3.3 Baseline Models

We select a diverse set of open-source LLMs, spanning: reasoning and non-reasoning variants; small- (<10B), mid- (20B–30B), and large-size (30B–70B) models. In particular, we include Jamba Reasoning-3B (Lenz et al., 2025), Llama-3.1 Nemotron Nano (Team, 2025a), Llama-3.2-3B-Instruct and Llama-3.3-70B-Instruct (Team, 2024), Mistral Small-24B (Jiang et al., 2023), Magistral Small (AI, 2025), as well as QwQ-32B, Qwen2.5-32B and Qwen2.5-72B (Team, 2025b). We also benchmark against closed-source or frontier models, including Gemini 2.5 Flash (Gemini Team, Google, 2025), GPT-5 Mini (OpenAI, 2025), Claude Sonnet 4 (Anthropic, 2025), and Grok-4 Reasoning (xAI, 2025).

3.4 Prompts

We use two dedicated prompt templates to evaluate execution (forward) and counterfactual (backward) reasoning, respectively. To ensure fair comparison, we apply the same prompt structure and formatting constraints across all models. We adopt one-shot

Dataset (→) Model (↓)	CruxEval						HumanEval						PythonSaga					
	\mathcal{R}_{exec}		\mathcal{R}_{cf}		\mathcal{R}_{dual}		\mathcal{R}_{exec}		\mathcal{R}_{cf}		\mathcal{R}_{dual}		\mathcal{R}_{exec}		\mathcal{R}_{cf}		\mathcal{R}_{dual}	
	$k=1$	$k=5$	$k=1$	$k=5$	$k=1$	$k=5$	$k=1$	$k=5$	$k=1$	$k=5$	$k=1$	$k=5$	$k=1$	$k=5$	$k=1$	$k=5$	$k=1$	$k=5$
Jamba Reasoning 3B	0.0	0.0	9.4	14.8	0.0	0.0	0.0	0.0	15.0	31.0	0.0	0.0	0.0	0.0	10.6	12.8	0.0	0.0
Nemotron Nano	3.0	8.1	15.8	30.9	0.3	2.7	0.0	0.0	15.0	30.0	0.0	0.0	0.0	0.0	10.6	34.0	0.0	0.0
Llama-3.2-3B-Inst.	0.0	0.0	12.8	32.6	0.0	0.0	0.0	0.0	21.0	43.0	0.0	0.0	0.0	0.0	4.3	27.7	0.0	0.0
Mistral Small 24B	29.2	69.8	43.6	63.4	13.4	43.6	10.0	35.0	50.0	61.0	4.0	20.0	6.4	6.4	44.7	53.2	2.1	2.1
Magistral Small	27.2	62.4	18.5	34.2	3.0	21.5	13.0	40.0	21.0	30.0	1.0	16.0	4.3	14.9	23.4	36.2	2.1	4.3
QwQ-32B	20.1	53.4	50.7	63.1	11.4	33.2	21.0	38.0	61.0	76.0	13.0	25.0	0.0	2.1	53.2	80.9	0.0	2.1
Qwen2.5-32B	34.2	71.1	57.7	79.5	18.1	57	11.0	37.0	64.0	77.0	8.0	31.0	0.0	8.5	46.8	55.3	0.0	6.4
Llama-3.3-70B-Inst.	15.8	38.6	49.3	68.1	8.7	27.5	5.0	16.0	50	64	3.0	9.0	0.0	0.0	51.1	61.7	0.0	0.0
Qwen2.5-72B	34.9	74.8	41.3	52.3	14.1	39.3	10.0	31.0	41.0	50.0	1.0	12.0	0.0	0.0	38.3	63.8	0.0	0.0
Gemini 2.5 Flash	67.4	85.9	72.1	85.2	51.0	73.8	50.0	77.0	69.0	80.0	41.0	62.0	6.4	12.8	57.4	66.0	4.3	8.5
GPT-5 Mini	89.9	95.6	81.2	97.0	73.8	91.6	79.0	88.0	77.0	88.0	59.0	76.0	36.2	46.8	83.0	95.7	29.8	44.7
Grok-4 Reasoning	85.6	86.9	93.0	97.0	79.9	84.9	78.0	84.0	85.0	88.0	65.0	72.0	55.3	68.1	93.6	95.7	51.1	66.0
Claude Sonnet 4	91.9	98.7	94.3	96.3	87.6	95.3	83.0	92.0	84.0	87.0	69.0	79.0	59.6	72.3	91.5	95.7	55.3	70.2

Table 1: Model performance comparison on execution reasoning (\mathcal{R}_{exec}) and counterfactual reasoning (\mathcal{R}_{cf}) tasks, as well as our dual-path reasoning (\mathcal{R}_{dual}) framework across CruxEval, HumanEval, and PythonSaga datasets (ordered left-to-right by increasing program complexity). Results are reported using pass@ k metrics (in %).

prompting (*i.e.*, with a single illustrative example) for all experiments on the three benchmarks. We also analyze robustness of DEXBENCH by probing the model sensitivity to more complex prompts (Section 4). See Appendix A for all templates.

3.5 Evaluation Metrics

For both execution (forward) and counterfactual (backward) reasoning, we evaluate performance using the pass@ k metric (Chen et al., 2021). In execution reasoning, an instance is considered successful if at least one of the k generated candidates predicts code coverage exactly correctly. Formally, the success indicator S_{exec} for the instance is

$$S_{exec} = \mathbb{1} \left(\exists \tilde{\phi}_i \in \{\tilde{\phi}_1, \dots, \tilde{\phi}_k\} \text{ s.t. } \tilde{\phi}_i = \phi(\tau_{exec}) \right),$$

where $\tilde{\phi}_i$ denotes the code coverage predicted by the i -th candidate.

In counterfactual reasoning, an instance is considered successful if at least one of the k generated test inputs covers the originally uncovered branch b (as identified in Section 3.2). Formally, the success indicator S_{cf} for the instance is defined as

$$S_{cf} = \mathbb{1} \left(\exists I_{cf,i} \in \{I_{cf,1}, \dots, I_{cf,k}\} \text{ s.t. } b \in \phi(\tau_{cf,i}) \right),$$

where $\tau_{cf,i}$ is the execution trace produced by running the corresponding program on input $I_{cf,i}$.

Putting them together, we define a joint success indicator S_{dual} for dual-path reasoning as

$$S_{dual} = S_{exec} \wedge S_{cf}.$$

In other words, an instance is successful under the joint metric only if the model both correctly predicts the original coverage and generates at least one input whose execution covers the target branch.

4 Empirical Results

4.1 Performance Comparison in Reasoning about Program Execution

In Table 1, we compare model performance in execution reasoning (\mathcal{R}_{exec}), counterfactual reasoning (\mathcal{R}_{cf}), as well as their dual-path formulation (\mathcal{R}_{dual}). Our analysis considers three dimensions:

(1) **Program Complexity.** While closed-source or frontier models generally outperform the open-source ones, performance on dual-path reasoning consistently drops as program complexity increases (CruxEval < HumanEval < PythonSaga). On the PythonSaga dataset featuring intricate control-flow logic, Gemini 2.5 Flash records particularly low pass@1 and pass@5 scores of 4.3% and 8.5%, respectively. These results are comparable to those of the mid- and large-size open-source models and are significantly lower compared to other closed-source models, whose scores range from 29.8%–55.3% (pass@1) and 44.7%–70.2% (pass@5). Overall, increasing program complexity impacts success in joint reasoning about execution and counterfactual paths, as greater path divergence makes it difficult for models to maintain a consistent causal representation of the program.

(2) **Model Scaling Laws.** All small-scale models (<10B parameters) consistently fail in dual-path reasoning due to achieving near-zero performance on code coverage prediction (*i.e.*, execution reasoning, \mathcal{R}_{exec}) despite non-trivial results in branch-guided input mutation (*i.e.*, counterfactual reasoning, \mathcal{R}_{cf}). Scaling to mid-sized models (20B–30B parameters) yields substantial improvements in both reasoning tasks. However, these

trends do not extend to large-size models ($> 30B$ parameters). In particular, within the Qwen family, the 32B parameter variant outperforms the 72B model in dual-path reasoning across all datasets.

(3) Reasoning-Focused Post Training. Among the open-source models, we considered two pairs of medium-sized non-reasoning and reasoning variants: (Mistral Small 24B, Magistral Small), (Qwen2.5-32B, QwQ-32B). In both pairs, non-reasoning models surprisingly outperform the reasoning ones in dual-path reasoning by an average of 63.9% and 47.9% (pass@5) on CruxEval and HumanEval. This trend persists even on the complex PythonSaga dataset, where Qwen2.5-32B (6.4%) outperforms QwQ-32B (2.1%), although Magistral Small (4.3%) exhibits a marginal improvement over its non-reasoning counterpart (2.1%). *These results indicate that such post-training does not generalize to reasoning about program execution.*

In contrast, closed-source models demonstrate the best performance in dual-path reasoning across all datasets (Claude Sonnet 4 $>$ Grok-4 Reasoning $>$ GPT-5 Mini $>$ Gemini 2.5 Flash). However, due to a lack of transparency in their design, these gains cannot be directly attributed to reasoning-focused post-training alone and instead serve as reference upper bounds on achievable performance.

4.2 Execution vs. Counterfactual Reasoning

In this section, we investigate the asymmetry in model performance (as in Section 4.1) in both forward and backward reasoning tasks, isolating *where*, and *to what extent* models break down.

Diagnosing Asymmetry in Execution and Counterfactual Reasoning. In Table 1, we reported a consistent asymmetry in model performance on execution and counterfactual reasoning. Furthermore, as illustrated in Figure 3, many models succeed for only one of the two tasks. Here, the execution-only setting (*i.e.*, $\mathcal{R}_{exec} : \checkmark$, $\mathcal{R}_{cf} : \times$) reflects a model’s ability to predict observed program behavior for a fixed input, while failing to reason about how changes to intermediate program state induce alternative execution flows. In contrast, the counterfactual-only setting (*i.e.*, $\mathcal{R}_{exec} : \times$, $\mathcal{R}_{cf} : \checkmark$) suggests that a model can heuristically mutate inputs to alter execution flow without reliably predicting the execution behavior itself. When considered in isolation, both evaluation settings probe complementary model capabilities, as evidenced among the open-source models. However, a

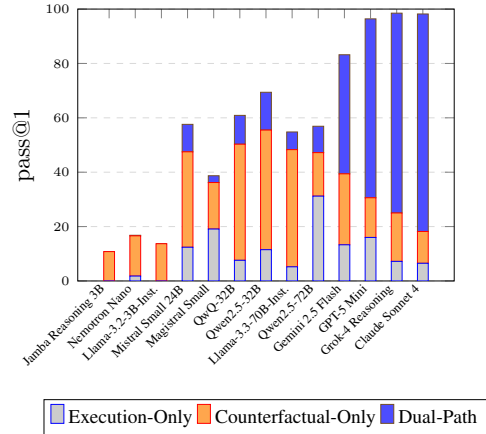


Figure 3: Strict performance comparison (pass@1, in %) in execution-only, counterfactual-only, and dual-path reasoning evaluation settings across three datasets.

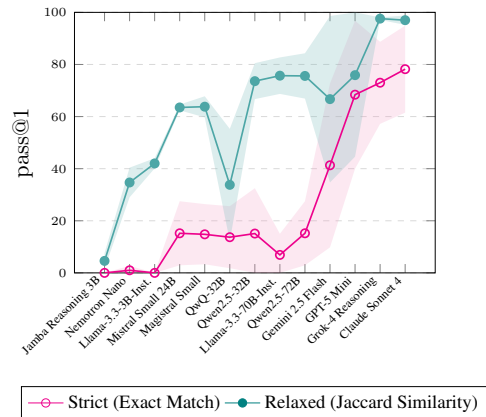


Figure 4: Strict vs. relaxed performance comparison (pass@1, in %) on execution reasoning \mathcal{R}_{exec} . Here, shaded regions indicate \pm std across three datasets.

joint evaluation under dual-path reasoning offers a systematic framework for quantifying reasoning capabilities of LLMs in program execution, enabling rigorous and comparative analysis across models.

Execution Reasoning under Relaxed Evaluation.

Figure 4 reports the execution reasoning performance of all models across three datasets in two evaluation settings: *strict* and *relaxed*. The former corresponds to the binary success indicator defined in Section 3.5, whereas the latter measures partial correctness using Jaccard similarity between the predicted ($\tilde{\phi}$) and ground-truth ($\phi(\tau_{exec})$) coverage sets. The relaxed success score for an instance is:

$$S_{exec}^r = \frac{|\tilde{\phi} \cap \phi(\tau_{exec})|}{|\tilde{\phi} \cup \phi(\tau_{exec})|}.$$

Here, a larger gap between strict and relaxed performance indicates that models often identify

Dataset (→)	C/F	CruxEval		HumanEval		PythonSaga	
		Model (↓)	Reasoning	k=1	k=5	k=1	k=5
GPT-5 Mini	Generation	89.3	97.7	93.0	99.0	87.2	95.7
	Mutation	81.2	97.0	77.0	88.0	83.0	95.7
Grok-4 Reasoning	Generation	97.3	99.0	98.0	99.0	93.6	95.7
	Mutation	93.0	97.0	85.0	88.0	93.6	95.7

Table 2: **Ablation Study.** Model performance under input generation vs. mutation (*ours*) for counterfactual reasoning (pass@k, in %) across three datasets.

a substantial subset of the correct code coverage while failing the exact-match criteria, and the width of the shaded region further captures variability in performance across datasets of increasing program complexity. Notably, the mid-size open-source reasoning model QwQ-32B exhibits less precise per-statement execution prediction than the small-size models as well. In contrast, larger open-source models achieve relaxed performance comparable to that of closed-source models like Gemini-2.5 Flash and GPT-5 Mini. Among all models, Grok-4 Reasoning and Claude Sonnet 4 make the least per-statement errors while exhibiting consistent performance across all three datasets, indicating their superior execution reasoning capabilities.

Counterfactual Reasoning w/o Test Input. In the counterfactual reasoning task, models mutate a given test input to satisfy specific branch coverage objectives. In this experiment, we remove the test input and instead require models to directly generate an input that reaches the target branch, *i.e.*, without conditioning on the test input, thus transforming branch-targeted input mutation into branch-targeted input generation (see Appendix A).

Table 2 reports the performance of the well-performing GPT-5 Mini and Grok-4 Reasoning (from Section 4.1). Contrary to initial expectations, we observed better performance under input generation compared to the mutation setting, improving pass@1 and pass@5 scores across three datasets by 11.9% and 4.4% for GPT-5 Mini, and by 6.6% and 4.9% for Grok-4 Reasoning. However, we soon realized that this is due to the relaxation of causal constraints in the input generation setting, where models no longer require to reason about how specific input perturbations propagate through intermediate program states. As such, these results support our original design choice by highlighting the greater difficulty of branch-targeted input mutation and thereby, motivating its use in DEXBENCH as the counterfactual reasoning task.

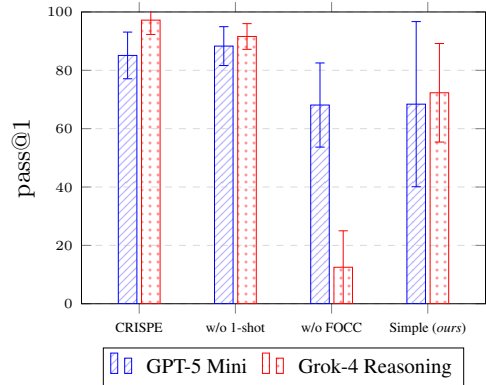


Figure 5: **Robustness to prompt complexity.** Average model performance for execution reasoning (pass@1, in %). Here, whiskers denote $\pm\sigma$ across three datasets.

4.3 Sensitivity Analysis

In this section, we examine the robustness of model performance in both execution and counterfactual reasoning under different perturbations.

Robustness to Prompt Complexity. To assess the robustness of execution reasoning within DEXBENCH under increased prompt complexity, we replace our prompt for predicting code coverage with that from the state-of-the-art CRISPE (Dhulipala et al., 2025). The latter: (i) adopts one-shot prompting to enable step-by-step reasoning about program execution, and (ii) provides a finite set of feasible options for code coverage (FOCC) for the model to select from, thus transforming prediction task into a constrained selection problem.

Figure 5 compares the performance of CRISPE, its ablation baselines (w/o one-shot prompting and w/o FOCC), and our prompt for execution reasoning. CRISPE yields consistently high performance across datasets for both GPT-5 Mini and Grok-4 Reasoning, as indicated by narrower performance variability. The ablation results further reveal that Grok-4 Reasoning relies more on FOCC than GPT-5 Mini: removing FOCC results in a substantially larger drop in pass@1 for Grok-4 Reasoning (677.6%) compared to GPT-5 Mini (25%). This suggests that Grok-4 Reasoning is more sensitive to increased prompt complexity and subsequently longer step-by-step reasoning traces.

Robustness to Counterfactual Path Selection Strategies. In Section 3.2, we selected the counterfactual path that yields the *most* increase in code coverage relative to the original execution path. To evaluate the sensitivity of counterfactual reasoning in DEXBENCH to this selection heuristic, we

Dataset (→) Model (↓)	C/F Path Selection	CruxEval		HumanEval		PythonSaga	
		k=1	k=5	k=1	k=5	k=1	k=5
GPT-5 Mini	Random	71.4	83.3	67.9	67.9	58.8	70.6
	Min	76.2	92.9	71.4	89.3	76.5	94.1
	Max (<i>ours</i>)	85.7	97.6	78.6	89.3	60.7	94.1
Grok-4 Reasoning	Random	85.7	88.1	89.3	89.3	94.1	94.1
	Min	85.7	90.5	89.3	89.3	94.1	94.1
	Max (<i>ours</i>)	97.6	97.6	89.3	89.3	88.2	94.1

Table 3: **Robustness to counterfactual paths.** Model performance on counterfactual reasoning (pass@ k , %).

compare against two additional strategies with the counterfactual path: (i) picked at *random*, and (ii) results in the *least* increase in code coverage.

Table 3 reports the performance of GPT-5 Mini and Grok-4 Reasoning in counterfactual reasoning. We observe that the models are generally sensitive to the counterfactual objective, which determines the degree of causal reasoning required to identify input perturbations that induce a specific execution flow. In particular, paths that yield maximal coverage increase typically execute the core logic of the program, while those with minimal coverage increase often correspond to early exit conditions. The intermediate program state triggering the latter depends on satisfying highly specific constraints, thus requiring deeper causal understanding. Between both models, we see that Grok-4 Reasoning is more robust to counterfactual path selection and records high performance across target branches, whereas GPT-5 Mini shows larger variability. As such, while a maximal coverage heuristic is reasonable, these findings suggest that augmenting DEXBENCH with additional counterfactual targets could enable an even comprehensive assessment of counterfactual reasoning performance.

4.4 Qualitative Analysis of Model Outputs

To identify the root causes of model failures, we manually inspected reasoning traces produced by closed-source models under both execution (forward) and counterfactual (backward) reasoning. Following prior work (Abdollahi et al., 2025), we categorize failures into following high-level error types: *Computation Errors*, *Control Flow Errors*, *Lack of Fact Verification*, *Native API Miscalculation*, *Prompt Misread*, *Null Response*, *Output Format Errors*, *Misreporting Final Output*, *Skipping Statements*, and *Hallucinations* (Appendix C.1).

As shown in Table 4, the error categories do not apply uniformly across both reasoning tasks. For execution reasoning, the model errors primarily

occur due to: misunderstanding task instructions (Prompt Misread), incorrectly parsing and processing program representations (Skipping Statements, Hallucinations), and failing to produce outputs (Null Response). For counterfactual reasoning, Gemini-2.5 Flash and GPT-5 Mini make the most errors due to producing a null response or an error in the output format. In contrast, Grok-4 Reasoning and Claude Sonnet 4 produce few or no such errors. However, Claude Sonnet 4 exhibits more semantic errors (*e.g.*, Native API Miscalculation, Control Flow Errors) than even Gemini-2.5 Flash, suggesting a gap in modeling execution flow which requires the model to maintain causal consistency.

Illustration. In multiple API-centric tasks that involve string manipulation, Claude Sonnet 4 correctly predicts the code coverage (*i.e.*, execution reasoning) but fails to produce a valid counterfactual input (*i.e.*, counterfactual reasoning). In some cases, the failure arises from a misinterpretation of API semantics (*e.g.*, `splitlines()` vs. `split()`). Consistent with recent findings on Claude’s handling of `split()` API (Abdollahi et al., 2025), these are not due to ambiguity in task specification but reflect an incorrect understanding of API behavior.

The above analyses show that models may succeed in execution reasoning while failing in counterfactual reasoning due to semantic errors, or vice versa. As a result, isolated evaluations based on execution-only or counterfactual-only settings do not provide a robust evaluation of model capabilities, underscoring the need for jointly evaluating both directions, as in our dual-path framework.

5 Related Work

Emergent reasoning capabilities in LLMs have been studied across multiple domains, including mathematical reasoning (Chen et al., 2024; Jain et al., 2025; Lewkowycz et al., 2022), formal logic (Morishita et al., 2023), and program understanding (Xie et al., 2025; Chen et al., 2025; Abdollahi et al., 2025). For source code, prior work has focused on evaluating the ability of LLMs to reason about program properties, either derived from program structure and semantics or observed runtime behaviors, *i.e.*, static (Yadavally et al., 2024b; Xie et al., 2025) and dynamic code reasoning (Liu et al., 2023; Yadavally et al., 2024a; Patel et al., 2025), respectively. In this work, we focus on fine-grained benchmarking of dynamic code understanding.

CruxEval (Gu et al., 2024) evaluates LLMs by

pairing output prediction (given a program and input) with input prediction (given a program and output). CodeMind (Liu et al., 2024) extends beyond input-output prediction by introducing tasks on understanding semantics, specification, and execution simulation. R-Eval (Chen et al., 2025) introduced IC-Score, a measure derived from the composition of performance on program execution-specific tasks such as predicting code coverage, program states, execution paths, and program outputs. However, these benchmarks remain limited to *single-path evaluations* induced by specific inputs. While CES (Liu et al., 2025) focuses on reasoning consistency across multiple inputs, it treats resulting execution paths independently, thereby leaving the causal logic that is characteristic of program execution unexplored. In contrast, DEXBENCH introduces a dual-path reasoning framework. By requiring the model to predict the observed behaviors along an execution path, and the behavioral changes required to induce an alternative flow, DEXBENCH provides a more robust and systematic evaluation of LLMs over program execution.

6 Conclusion

In this work, we argue that reasoning about program execution is inherently dual in nature, requiring both the ability to predict dynamic program behavior and to reason counterfactually to induce alternative behaviors. Our results show that strong performance on either task does not consistently translate to success under joint evaluation, an asymmetry not reflected in single-path evaluations. These findings suggest that the proposed dual-path reasoning framework provides a more robust evaluation of dynamic code understanding in LLMs and can be extended to alternative pairs of observed runtime behaviors and counterfactual objectives.

7 Limitations

Data Contamination. This work sources Python programs from public benchmarks, *i.e.*, CruxEval, HumanEval, and PythonSaga, and therefore may be susceptible to data contamination. However, the formulation in DEXBENCH mitigates this risk by requiring fine-grained reasoning over intermediate program state rather than direct input-output mapping. In particular, the counterfactual reasoning task requires conditional mutation of inputs, where a model must identify and analyze specific branches and perturb inputs to redirect the flow of

execution. Thus, even if a model has potentially “seen” the underlying program, successfully synthesizing a constraint-satisfying input requires an explicit understanding of how input perturbations propagate under execution. As a result, model performance in DEXBENCH is less likely to stem from surface-level memorization and instead reflects its ability to reason over intrinsic execution flow.

Programming Languages. Constructing our benchmark relies on the availability of tools for execution tracing and instrumentation to reliably capture fine-grained runtime information. While these are readily available for Python (*e.g.*, SlipCover), extending this to other programming languages requires substantial engineering effort to build secure and reproducible sandboxed runtime environments. As a result, DEXBENCH focuses on Python programs drawn from widely adopted CruxEval, HumanEval, and PythonSaga datasets. However, the proposed dual-path reasoning framework itself is programming language-agnostic.

Counterfactual Path Selection. By default, this work selects the branch that maximizes increase in code coverage relative to the original execution. However, this can bias selection towards counterfactual paths that diverge more from the original execution, while overlooking paths corresponding to more localized behavioral variations. Although we explore alternative strategies in Section 4.3, identifying counterfactual paths that are both informative and representative remains challenging. Therefore, model performance reported in DEXBENCH may vary under different counterfactual selection strategies, and our findings should be interpreted in the context of this design choice.

Dynamic Program Behavior and Counterfactual Objective. This work focuses on evaluating dynamic program behavior through code coverage prediction, with the corresponding counterfactual objective being branch-targeted input mutation. While this provides a concrete and tractable formulation of dual-path reasoning, other alternatives are possible. For example, execution reasoning could instead target predicting program outputs or intermediate execution states, while counterfactual reasoning might involve identifying inputs that induce specific outputs or state transitions. Exploring such alternatives are useful to coding tasks like testing, debugging, and may yield complementary evaluation perspectives. This is left for future work.

References

- Mohammad Abdollahi, Khandaker Rifah Tasnia, Soumit Kanti Saha, Jinqiu Yang, Song Wang, and Hadi Hemmati. 2025. [Demystifying errors in LLM reasoning traces: An empirical study of code execution simulation](#). *CoRR*, abs/2512.00215.
- Mistral AI. 2025. [Magistral](#). *CoRR*, abs/2506.10910.
- Anthropic. 2025. [System card: Claude opus 4 & claude sonnet 4](#).
- Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. [Directed greybox fuzzing](#). In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2329–2344. ACM.
- Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. [Reasoning runtime behavior of a program with LLM: how far are we?](#) In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pages 1869–1881. IEEE.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. [Evaluating large language models trained on code](#). *CoRR*, abs/2107.03374.
- Nuo Chen, Hongguang Li, Baoyuan Wang, and Jia Li. 2024. [From good to great: Improving math reasoning with tool-augmented interleaf prompting](#). *CoRR*, abs/2401.05384.
- Hridya Dhulipala, Aashish Yadavally, Smit Soneshbhai Patel, and Tien N. Nguyen. 2025. [CRISPE: semantic-guided execution planning and dynamic reasoning for enhancing code coverage prediction](#). *Proc. ACM Softw. Eng.*, 2(FSE):2965–2986.
- Yangruibo Ding, Jinjun Peng, Marcus J. Min, Gail E. Kaiser, Junfeng Yang, and Baishakhi Ray. 2024. [Semcoder: Training code language models with comprehensive semantics reasoning](#). In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.
- Gemini Team, Google. 2025. [Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities](#).
- Alex Gu, Baptiste Rozière, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. [Cruxeval: A benchmark for code reasoning, understanding and execution](#). In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, *Proceedings of Machine Learning Research*, pages 16568–16621. PMLR / OpenReview.net.
- Kushal Jain, Moritz Miller, Niket Tandon, and Kumar Shridhar. 2025. [First-step advantage: Importance of starting right in multi-step math reasoning](#). In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, *Findings of ACL*, pages 766–778. Association for Computational Linguistics.
- Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Léo Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. [Mistral 7b](#). *CoRR*, abs/2310.06825.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. [Swe-bench: Can language models resolve real-world github issues?](#) In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.
- Barak Lenz, Opher Lieber, Alan Arazi, Amir Bergman, Avshalom Manevich, Barak Peleg, Ben Aviram, Chen Almagor, Clara Fridman, Dan Padnos, Daniel Gissin, Daniel Jannai, Dor Muhlgay, Dor Zimberg, Edden M. Gerber, Elad Dolev, Eran Krakovsky, Erez Safahi, Erez Schwartz, and 2 others. 2025. [Jamba: Hybrid transformer-mamba language models](#). In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.
- Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay V. Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. 2022. [Solving quantitative reasoning problems with language models](#). In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.
- Changshu Liu, Yang Chen, and Reyhaneh Jabbarvand. 2025. [Assessing coherency and consistency of code execution reasoning by large language models](#). *CoRR*, abs/2510.15079.
- Changshu Liu, Shizhuo Dylan Zhang, and Reyhaneh Jabbarvand. 2024. [Codemind: A framework to challenge large language models for code reasoning](#). *CoRR*, abs/2402.09664.

- Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. 2023. [Code execution with pre-trained language models](#). In [Findings of the Association for Computational Linguistics: ACL 2023](#), Toronto, Canada, July 9-14, 2023, Findings of ACL, pages 4984–4999. Association for Computational Linguistics.
- Terufumi Morishita, Gaku Morio, Atsuki Yamaguchi, and Yasuhiro Sogawa. 2023. [Learning deductive reasoning from synthetic corpus based on formal logic](#). In [International Conference on Machine Learning, ICML 2023](#), 23-29 July 2023, Honolulu, Hawaii, USA, Proceedings of Machine Learning Research, pages 25254–25274. PMLR.
- OpenAI. 2025. [Gpt-5 system card](#).
- Smit Patel, Aashish Yadavally, Hridya Dhulipala, and Tien N. Nguyen. 2025. [Planning a large language model for static detection of runtime errors in code snippets](#). In [47th IEEE/ACM International Conference on Software Engineering, ICSE 2025](#), Ottawa, ON, Canada, April 26 - May 6, 2025, pages 872–884. IEEE.
- Juan Altmayer Pizzorno and Emery D. Berger. 2023. [Slipcover: Near zero-overhead code coverage for python](#). In [Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023](#), Seattle, WA, USA, July 17-21, 2023, pages 1195–1206. ACM.
- Martin Riddell, Ansong Ni, and Arman Cohan. 2024. [Quantifying contamination in evaluating code generation capabilities of language models](#). In [Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics \(Volume 1: Long Papers\)](#), ACL 2024, Bangkok, Thailand, August 11-16, 2024, pages 14116–14137. Association for Computational Linguistics.
- Llama Team. 2024. [The llama 3 herd of models](#). [CoRR](#), abs/2407.21783.
- Llama-Nemotron Team. 2025a. [Llama-nemotron: Efficient reasoning models](#). [CoRR](#), abs/2505.00949.
- Qwen Team. 2025b. [Qwen3 technical report](#). [CoRR](#), abs/2505.09388.
- xAI. 2025. [Grok 4 model card](#).
- Danning Xie, Mingwei Zheng, Xuwei Liu, Jiannan Wang, Chengpeng Wang, Lin Tan, and Xiangyu Zhang. 2025. [CORE: benchmarking llms code reasoning capabilities through static analysis tasks](#). [CoRR](#), abs/2507.05269.
- Ankit Yadav, Himanshu Beniwal, and Mayank Singh. 2024. [Pythonsaga: Redefining the benchmark to evaluate code generating llms](#). In [Findings of the Association for Computational Linguistics: EMNLP 2024](#), Miami, Florida, USA, November 12-16, 2024, Findings of ACL, pages 17113–17126. Association for Computational Linguistics.
- Aashish Yadavally, Yi Li, and Tien N. Nguyen. 2024a. [Predictive program slicing via execution knowledge-guided dynamic dependence learning](#). [Proc. ACM Softw. Eng.](#), 1(FSE):271–292.
- Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2024b. [A learning-based approach to static program slicing](#). [Proc. ACM Program. Lang.](#), 8(OOPSLA1):83–109.

A Prompt Design

We design dedicated prompts to evaluate distinct reasoning abilities of LLMs:

- Figure 6 (top): Forward reasoning, where the model must simulate execution of a program to predict which lines would be executed for a given input (*i.e.*, statement coverage).
- Figure 6 (bottom): Backward reasoning, where the model must predict an input that would reach a branch uncovered by the original input for the same program.

Both templates enforce strict formatting to enable automated parsing and evaluation. Here, [PYTHON]/[/PYTHON] tags enclose the line number-indexed Python program, [TASK]/[/TASK] tags enclose the corresponding task instructions, <REASONING>/</REASONING> tags enclose model-generated reasoning traces, and [ANSWER]/[/ANSWER] tags enclose the model-generated task outputs.

B Counterfactual Path Selection

In this section, we describe the algorithm used to define behavioral goals for branch-targeted input mutation under counterfactual reasoning (\mathcal{R}_{cf}). As noted in Section 3.2, this procedure grounds counterfactual input mutation in branches that yield maximal increase in code coverage.

Given a coverage map produced by executing the program on a seed input, we first identify the set of uncovered source lines. We then analyze the program’s abstract syntax tree (AST) to enumerate all conditional branches and loop constructs, associating each such construct with the set of source lines contained in its body. The primary objective is to identify branches whose bodies are *entirely uncovered*, all of which represent possible targets for counterfactual exploration. Among these, we select the branch with the largest body, measured as the number of statements it contains. This favors branches that are likely to yield the *highest* increase in code coverage if successfully executed.

If no fully uncovered branch exists, we select uncovered lines that do not belong to the body of any conditional or loop. These standalone lines typically correspond to return statements, computations, or other control-flow-independent code regions. In this case, we deterministically select the

uncovered line with the smallest line number to serve as the target. Finally, if no such branches or statements can be identified, we ignore such programs as no meaningful counterfactual target is available under the current state. We formalize this alternative path selection process as follows:

Algorithm 1: Alternative Path Selection

```
Input: Coverage map  $\mathcal{C}$ , AST  $\mathcal{A}$ 
Output: Target line number  $\ell^*$  or null
1  $\mathcal{U} \leftarrow \{\ell \mid \text{coverage\_count}(\ell) = 0\}$ 
2  $\mathcal{T} \leftarrow \emptyset$ 
3 foreach branch  $b \in \text{FINDALLBRANCHES}(\mathcal{A})$  do
4    $\mathcal{B} \leftarrow \text{LINESINBRANCHBODY}(b)$ 
5   if  $\mathcal{B} \neq \emptyset$  and  $\mathcal{B} \subseteq \mathcal{U}$  then
6      $\mathcal{T} \leftarrow \mathcal{T} \cup \{b\}$ 
7   end
8 end
9 if  $\mathcal{T} \neq \emptyset$  then
10   $\mathcal{S} \leftarrow \emptyset$ 
11  foreach branch  $b \in \mathcal{T}$  do
12     $s_b \leftarrow |\text{LINESINBRANCHBODY}(b)|$ 
13     $\mathcal{S} \leftarrow \mathcal{S} \cup \{(b, s_b)\}$ 
14  end
15   $b^* \leftarrow \arg \max_{(b, s_b) \in \mathcal{S}} s_b$ 
16  return  $\text{LINENUMBER}(b^*)$ 
17 end
18  $\mathcal{U}_{\text{standalone}} \leftarrow \mathcal{U} \setminus \text{ALLBRANCHBODYLINES}(\mathcal{A})$ 
19 if  $\mathcal{U}_{\text{standalone}} \neq \emptyset$  then
20   return  $\min(\mathcal{U}_{\text{standalone}})$ 
21 end
22 return null
```

C Qualitative Analysis of Model Outputs

C.1 Error Taxonomy

We manually inspected reasoning traces for failure cases and categorized them into ten distinct error types, as summarized in Table 4.

C.2 Illustrative Examples

- In Figure 7 (CRUXEval-492), the model produces a largely correct trace but fails due to a basic counting error (hallucinating character counts), demonstrating that even strong models are prone to verifiable logic lapses when constraints are inverted.
- In Figure 8, the model misinterprets the behavior of Python’s `splitlines()`, propagating this error into an invalid input suggestion. This aligns with discrepancies noted in recent literature regarding Claude’s handling of the `split()` function (Abdollahi et al., 2025).

Prompt Template for Execution (Forward) Reasoning

You are given a Python program inside the [PYTHON] and [/PYTHON] tags and a single task between [TASK][/]TASK] tags. Your only function is to comply with Output requirements below.

```
[PYTHON]
{program_code}
[/PYTHON]
```

```
[TASK]
Analyze the provided Python script to determine its code coverage when executed. Identify all executable line numbers that are run at least once.
[/TASK]
```

Output requirements (strict):

1. First, analyze the execution paths and predict which lines would be executed. Put your reasoning inside <REASONING></REASONING> tags.
2. Then, provide the final answer inside [ANSWER] and [/ANSWER] tags as a single JSON object with one key:
 - `executed_lines`: JSON list of integers (sorted ascending) for all executable lines that would run.

Prompt Template for Counterfactual (Backward) Reasoning

You are an expert in directed fuzzing and test-case generation. You are given a Python program inside the [PYTHON] and [/PYTHON] tags, and a single task between [TASK][/]TASK] tags. Your only function is to comply with Output requirements below.

```
[PYTHON]
{program_code}
[/PYTHON]
```

```
[TASK]
The target line {priority_line} is either a conditional (if/elif), a loop header (for/while), or a return statement. Mutate the existing input seed so that, when passed to the function under test, the body of the conditional or loop at line {priority_line} will be executed. If the line is a return statement, that line should be triggered.
[/TASK]
```

Output requirements (strict):

1. First, analyze the program and reason about what inputs would reach line {priority_line}. Put your reasoning inside <REASONING></REASONING> tags.
2. Then, provide the final answer inside [ANSWER] and [/ANSWER] tags with ****only**** the comma-separated argument values (for example: ``["a", "bb"], 0.5``).

Figure 6: Prompt templates for: (top) execution, and (bottom) counterfactual reasoning in DEXBENCH.

```

1 import unittest
2
3 def f(text, value):
4     ls = list(text)
5     if (ls.count(value)) % 2 == 0:
6         while value in ls:
7             ls.remove(value)
8     else:
9         ls.clear()
10    return ''.join(ls)
11
12 # Test Assertion
13 unittest.TestCase().assertEqual(
14     f('abbkebanuuwrzvr', 'm'), 'abbkebanuuwrzvr'
15 )

```

Program: CRUXEval-492

Uncovered Branch: 6→7

Target: Branch at line 6

LLM Answer:

'abbkebanuuwrzvr', 'b'

Error: Computation Error and Prompt Misunderstanding

Chain-of-Thought Reasoning Trace of Claude Sonnet 4

Let me analyze the program to understand when line 6 will be executed. The function `f(text, value)` takes two parameters: `text` is converted to a list `ls`, and `value` is the character to count and potentially remove. The flow is:

1. Convert text to list.
2. Check if count of value is even.
3. If even: enter while loop removing all instances.
4. If odd: clear the list.

The loop runs only if the count of value is even, and value exists in the list. In the given test case, 'm' appears 0 times (even), but is not in the list, so the loop does not execute. To trigger execution, choose a character with even count > 0. From the string:

- 'a' appears 2 times
- 'b' appears 4 times
- 'e' appears 2 times
- 'u' appears 2 times
- 'r' appears 2 times

We can use any of these; choosing 'b' since it appears 4 times.

Figure 7: Incorrect letter counting logic: Here, the model fails to correctly analyze string operations.

```

1 import unittest
2
3 def f(text):
4     created = []
5     for line in text.splitlines():
6         if line == '':
7             break
8         created.append(list(list(line.rstrip())[:-1][
9             flush]))
10
11     return created [::-1]
12
13 flush = 0
14
15 # Test Assertion
16 unittest.TestCase().assertEqual(f('A(hiccup)A'), [['A'
17 ]])

```

Program: CRUXEval-712

Uncovered Branch: 6→7

Target: Branch at line 6

LLM Answer: 'A\n'

Error: Native API Misevaluation

Chain-of-Thought Reasoning Trace of Claude Sonnet 4

Let me analyze the program step by step:

1. The function `f(text)` takes a text parameter.
2. It splits the text into lines using `text.splitlines()`.
3. It iterates through each line in a for loop (line 5).
4. Line 6 is the conditional: `if line == ''`.
5. If this condition is true (line is empty), it breaks out of the loop
6. Otherwise, it processes the line and appends to `created`. To reach line 6, I need the function to be called with text that, when split into lines, produces at least one line to iterate over the for loop.

Looking at the current test case: `f('A(hiccup)A')`

- `'A(hiccup)A'.splitlines()` returns `'A(hiccup)A'` (a list with one element)
- The for loop will iterate once with line `'A(hiccup)A'`
- Line 6 will be reached and will check if `'A(hiccup)A' == ''`:
- This evaluates to `False`, so the `break` does not execute. To make the body of the conditional at line 6 execute (the `break` statement), I need a line that equals an empty string. This can happen if:
 - The text contains an empty line (two consecutive newlines)
 - The text starts with a newline
 - The text ends with a newline followed by nothing.

The simplest way is to provide text that contains an empty line. For example, `'A\n'` would create `['A', '']` when split; `'A\n\n'` would create `['A', '', '']` when split; `'\n'` would create `['', '']` when split. Let me use `'A\n'` which will create two lines `['A', '']`. The loop will first process `'A'`, then on the second iteration, line will be `''`, making the condition `line == ''` true, thus executing the `break` statement.

Figure 8: Incorrect `splitlines()` analysis: Here, the model misunderstands the edge cases in text processing.

Error Category	Reasoning	Gemini-	GPT-5	Grok-4	Claude
		2.5 Flash	Mini	Reasoning	Sonnet-4
<i>Computation Errors</i> : Arithmetic and type conversion faults	\mathcal{R}_{exec}	3	2	0	2
	\mathcal{R}_{cf}	2	0	2	2
<i>Control Flow Errors</i> : Incorrect predicate evaluation in loops or conditionals	\mathcal{R}_{exec}	3	0	0	1
	\mathcal{R}_{cf}	1	1	2	4
<i>Lack of Fact Verification</i> : Model fails to logically link input mutation to target path, leading to a false result despite seemingly coherent reasoning steps	\mathcal{R}_{exec}	0	1	1	5
	\mathcal{R}_{cf}	1	3	3	3
<i>Native API Misevaluation</i> : Hallucinations or errors related to standard library behavior (e.g., string manipulation)	\mathcal{R}_{exec}	1	0	0	1
	\mathcal{R}_{cf}	2	1	1	4
<i>Prompt Misread</i> : Model misinterprets the task objective, such as targeting the execution line number rather than the branch body	\mathcal{R}_{exec}	57	4	7	4
	\mathcal{R}_{cf}	18	0	5	3
<i>Null Response</i> : Model generates empty or incomplete outputs	\mathcal{R}_{exec}	90	40	0	2
	\mathcal{R}_{cf}	81	33	0	6
<i>Output Format Errors</i> : Test inputs under counterfactual reasoning violate the function signature (e.g., providing a list instead of comma-separated values)	\mathcal{R}_{exec}	0	0	0	0
	\mathcal{R}_{cf}	22	39	11	1
<i>Misreporting Final Output</i> : Model correctly infers which lines are executed but inconsistently reports them in its final output (or vice versa)	\mathcal{R}_{exec}	1	1	0	3
	\mathcal{R}_{cf}	0	0	0	0
<i>Skipping Statements</i> : Model omits one or more consecutively executed, executable statements within the same basic block	\mathcal{R}_{exec}	17	0	1	19
	\mathcal{R}_{cf}	0	0	0	0
<i>Hallucination</i> : Model marks non-executable lines as executed	\mathcal{R}_{exec}	51	41	79	46
	\mathcal{R}_{cf}	0	0	0	0

Table 4: Error taxonomy for Gemini-2.5 Flash, GPT-5 Mini, Grok-4 Reasoning and Claude Sonnet-4 models across three datasets, under both execution (\mathcal{R}_{exec}) and counterfactual (\mathcal{R}_{cf}) reasoning settings.