CoRE: A Fine-Grained Code Reasoning Benchmark Beyond Output Prediction

Anonymous ACL submission

Abstract

Despite strong performance on code generation tasks, it remains unclear whether large language models (LLMs) genuinely reason about code execution. Existing code reasoning benchmarks primarily evaluate final output correctness under a single canonical implementation, leaving two critical aspects underexplored: (1) whether LLMs can maintain consistency to functionally equivalent implementations, and (2) whether LLMs can accurately reason about intermediate execution states. We introduce **CoRE**, a **Co**de **Re**asoning benchmark that evaluates code reasoning through **implementation invariance** and **process transparency**. Extensive evaluations on eight frontier LLMs reveal two fundamental limitations. First, models exhibit a substantial **robustness gap**, with performance varying significantly across equivalent implementations. Second, we observe **superficial execution**, where models arrive at correct final outputs without correctly reasoning about intermediate execution states. Together, these findings demonstrate that output-only evaluations are insufficient for assessing code reasoning and position CoRE as a necessary benchmark for evaluating robust and faithful code reasoning.

1 Introduction

The capabilities of Large Language Models (LLMs) in code-relevant tasks have evolved rapidly from simple code completion to solving complex programming problems (Hurst et al., 2024; Liu et al., 2024a; Yang et al., 2024; Bouzenia et al., 2024; Sun et al., 2025; Guo et al., 2025). However, recent studies indicate that the ability to generate syntactically correct code does not necessarily imply a genuine understanding of its execution (Zhao et al., 2025; Gao et al., 2025b). This discrepancy raises a fundamental question: *Do LLMs truly understand the execution logic or are they merely relying on superficial heuristics?*

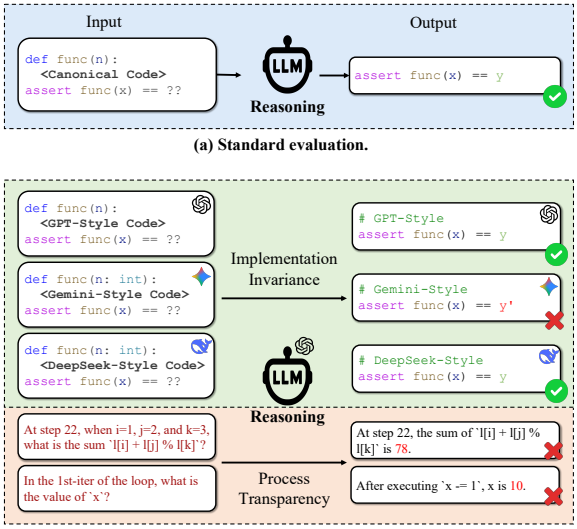
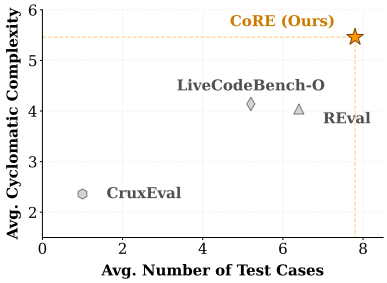


Figure 1: The code reasoning comparison of standard code reasoning evaluation and our holistic evaluation with diverse implementations and intermediate probing.

In this paper, we argue that current benchmarks are insufficient due to two critical flaws. First, they lack **implementation invariance**, defined as the ability to evaluate reasoning robustness across functionally equivalent but structurally or lexically distinct code implementations. However, benchmarks like CruxEval (Gu et al., 2024), LiveCodeBench-O (Jain et al.), and REval (Chen et al., 2025) primarily rely on a single *canonical* solutions, as illustrated in Fig.1(a). Second, existing benchmarks lack **process transparency**, as they typically predict the final outputs without verifying intermediate execution states. While REval pioneered the exploration of intermediate reasoning states, it remains limited by a rigid templating paradigm and a fundamental disregard for implementation invariance.

In light of these limitations, we introduce CoRE, a **Co**de **Re**asoning benchmark that jointly evaluates **implementation invariance** and **process transparency** in code reasoning. CoRE is constructed



	Samples	Tests	Impl.	$\mathcal{J}^{\downarrow}$	CC $^{\uparrow}$	Inter. $^{\uparrow}$
CruxEval	800	1.0	1.0	1.0	2.4	✗
LiveCodeBench-O	478	5.2	1.0	1.0	4.1	✗
REval	955	6.4	1.0	1.0	4.0	3.0
CoRE (ours)	**3,050**	**7.8**	**4.3**	**0.6**	**5.5**	**4.4**

Figure 2: Comparison of **CoRE** against existing code reasoning benchmarks. (Left) Distribution of Average Cyclomatic Complexity and Average Number of Test Cases. (Right) Comparison of code reasoning benchmarks. **Samples**: Total number of samples for inference. **Tests**: Average number of test cases per instance. **Impl.**: Average number of diverse code implementations per coding problem. $\mathcal{J}$: Average Jaccard similarity between diverse implementations for each coding problem, where lower indicates higher diversity. **CC**: Average Cyclomatic Complexity of code implementations. **Inter.**: Average number of intermediate state probes per coding problem.

from 91 coding problems from HumanEval (Chen et al., 2021) and LiveCodeBench (Jain et al., 2024), selected for their diverse test cases and algorithmic complexity. For each problem, CoRE evaluates whether LLMs produce consistent predictions across diverse but functionally equivalent implementations. As illustrated in Fig.1(b), we leverage implementations generated by the various LLMs listed in Tab.3 to quantify implementation invariance. Beyond this, CoRE further evaluates process transparency by examining the model's capacity to reason about intermediate execution states. To this end, we utilize verified codebases and employ five LLMs listed in Tab.3 to generate probes targeting intermediate states within nested loops and complex conditional branches. These intermediate state probes cover four empirical reasoning dimensions, *Arithmetic, Logic, State, and Boundary*, and are designed to test whether LLMs follow faithful step-by-step reasoning rather than relying on superficial heuristics. In total, as shown in Fig.2 and Fig.4, CoRE comprises 391 unique code implementations, with an average of 4.3 implementations per problem for implementation invariance evaluation. Each coding problem contains 7.8 test cases on average, and each implementation exhibits an average cyclomatic complexity of 5.5, as shown in Fig.2. For process transparency, each problem includes an average of 4.4 intermediate state probes, each covering 2.2 reasoning dimensions on average.

Our evaluation of eight frontier LLMs, including GPT-5, o3, Claude-4.5, and DeepSeek-V3.2, reveals critical limitations in code reasoning. First, we identify a significant **Robustness Gap**, where performance is inconsistent across syntactically diverse but functionally identical codes. Specifically, we observe that LLMs typically exhibit the highest proficiency with code generated by their own model family, and second-best on code from the OpenAI series, indicating an overfitting where LLMs are biased toward familiar styles. Second, we observe **Superficial Execution**, where LLMs produce correct function outputs but hallucinate intermediate states, revealing reliance on superficial heuristics rather than genuine execution understanding.

Overall, our contributions are threefold: (1) We propose CoRE, a challenging benchmark designed to rigorously assess code reasoning through implementation invariance and process transparency. (2) We reveal that LLMs frequently fail due to stylistic overfitting and superficial execution. (3) We introduce a holistic evaluation protocol that exposes the fragility of current LLMs, demonstrating that genuine execution understanding lags significantly behind output prediction.

2 Related Studies

Code reasoning tasks have emerged as a rigorous method to assess execution understanding in LLMs (Liu et al., 2024b; Gao et al., 2025b; Zhao et al., 2025). Although benchmarks like CruxEval (Gu et al., 2024) and LiveCodeBench-O (Jain et al., 2024) establish a foundation for this domain, they prioritize output prediction and effectively treat the reasoning mechanism as a black box. The reliance on single canonical implementations and low-complexity code makes it difficult to distinguish genuine execution simulation from heuristic input-output mapping. This distinction is vital given that LLM reasoning remains fragile and often unfaithful. Even with Chain-of-Thought

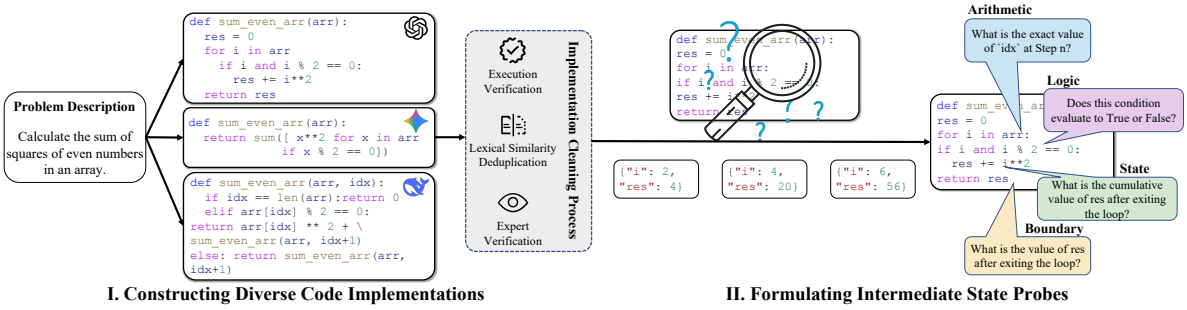


Figure 3: The construction pipeline of CoRE. It consists of two stages: (I) **Constructing Diverse Code Implementations**, which produces functionally equivalent code via massive LLMs, and validates its functionality and diversity, and (II) **Formulating Intermediate State Probes**, where LLMs are employed to synthesize probes grounded in captured execution traces across four dimensions: *Arithmetic, Logic, State, and Boundary*.

prompting (Wei et al., 2022), models frequently generate plausible yet hallucinated traces derived from shallow heuristics rather than actual logic reasoning (Liu et al., 2024b; Beger and Dutta, 2025; Gao et al., 2025c; Lanham et al., 2023; Turpin et al., 2023; Wang et al., 2024; Gao et al., 2025a). Such unreliability necessitates validating intermediate states instead of relying solely on terminal outputs (Uesato et al., 2022; Lightman et al., 2023). While REval (Chen et al., 2025) attempts to incorporate intermediate state prediction, its dependence on rigid templates and simple canonical code restricts its ability to evaluate reasoning robustness across varied coding styles. Current methods fail to verify if LLMs maintain consistency when processing functionally equivalent but syntactically diverse solutions. CoRE addresses these limitations by enforcing implementation invariance and process transparency to provide a strict evaluation of code reasoning.

3 CoRE Benchmark

In this section, we detail the construction process of CoRE, which aims to evaluate both the implementation invariance and process transparency of LLMs in code reasoning. The construction pipeline is shown in Fig.3, which consists of two stages. The first stage constructs diverse code implementations, and the second stage formulates intermediate state probes.

3.1 Constructing Diverse Code Implementations

To establish a high-quality foundation for code reasoning, we first collect an initial coding problem set from HumanEval (Chen et al., 2021) and LiveCodeBench (Jain et al., 2024). These benchmarks are selected for their rigorous coverage of test cases and algorithmic complexity, which can be used for evaluating code reasoning. We initially aggregate a pool of problems $\mathcal{P}$ comprising 164 coding problems from HumanEval and 880 from LiveCodeBench. As illustrated in first stage of our construction workflow in Fig.3, we then leverage seven diverse LLMs $\mathcal{M}$ to generate potential solutions, formulating a comprehensive pool $\mathcal{S}$ containing 7,308 distinct code implementations for 1,044 unique problems, defined as:

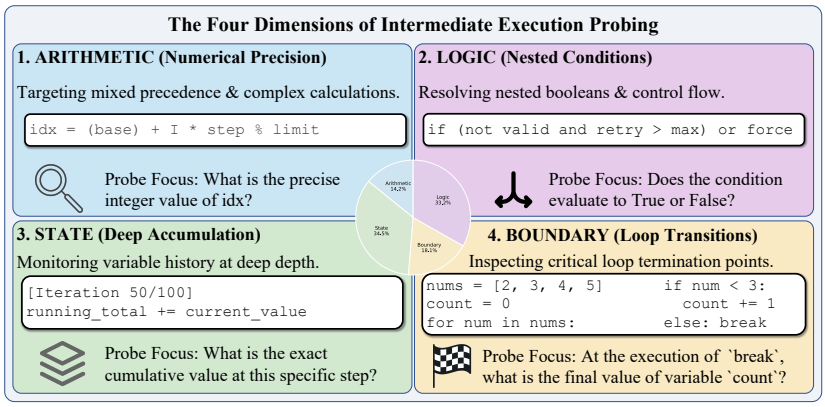
$$\mathcal{S} = \{s \mid s \sim m(p), \forall p \in \mathcal{P}, \forall m \in \mathcal{M}\}. \quad (1)$$

To ensure that the benchmark effectively evaluates code reasoning, we apply a hierarchical implementation cleaning process. We first perform execution validation to ensure functional equivalence, retaining only the solutions that pass their corresponding test sets $\mathcal{T}_{\mathcal{P}}$. We further refine this set to guarantee reasoning depth and lexical variety. Specifically, an instance s is retained if and only if its cyclomatic complexity $\mathcal{C}(s)$ exceeds a threshold τ_{cc} and its 1-gram Jaccard similarity $\mathcal{J}$ with other implementations *sharing the same problem ID* remains below τ_{sim}. The 1-gram Jaccard similarity (Wikipedia, 2026) are formulated as:

$$\mathcal{J}(s,t) = \frac{|\mathcal{G}_1(s) \cap \mathcal{G}_1(t)|}{|\mathcal{G}_1(s) \cup \mathcal{G}_1(t)|}, \quad (2)$$

where $\mathcal{G}_1(s)$ denotes the set of unique 1-grams (tokens) in code implementation s. Overall, the implementation cleaning process is defined as:

$$\mathcal{S}' = \left\{ s \;\middle|\; \begin{array}{l} \mathcal{C}(s) \geq \tau_{\text{cc}} \\ \wedge \max\limits_{\substack{t \in \mathcal{D}_{\text{io}} \setminus \{s\} \\ \text{id}(s)=\text{id}(t)}} \mathcal{J}(s,t) \leq \tau_{\text{sim}} \end{array} \right\}, \quad (3)$$



	Aspect	Number
Basic Statistic	Code Problems	91
	Code Candidates	391
	Intermediate State Probes	400
	Samples	3050
Impl. Statistic	OpenAI	119
	DeepSeek	88
	Gemini	95
	Claude	58
	Qwen	31
Probe Statistic	Arithmetic	124
	Logic	289
	State	300
	Boundary	157
	Avg. Dimensions	2.2

Figure 4: The taxonomy and distribution of Intermediate Probing dimensions. The left panel illustrates the four dimensions of execution probing. The right table summarizes the dataset composition. The middle section details the distribution of code candidates generated by various LLMs, and the bottom specifies the number of intermediate probing questions assigned to each reasoning dimension.

Recognizing that evaluating LLMs on such diverse candidates with massive test suites presents significant computational challenges, we optimize evaluation efficiency by computing $\mathcal{T}_{\mathcal{P}}{}^*$, the minimal subset of test cases achieving maximal coverage for problem $\mathcal{P}$. This reduction strategy reduces the average number of test cases per instance from 24.1 to 7.8 without compromising coverage integrity. Therefore, the larger number of test cases reported in Fig.2 is not a superficial increase in quantity, but reflects test cases that are necessary to ensure adequate coverage. Finally, human expert verification ensures implementation diversity, producing 391 curated implementations across 91 problems, as shown in Fig.2.

3.2 Formulating Intermediate State Probes

To formulate intermediate state probes, we first employ Python execution tracing to capture the runtime values of all variables, as illustrated in the middle of Fig.3, serving as ground truth for process transparency evaluation. Then, we employ an ensemble of LLMs to automatically produce intermediate state probes targeting four critical dimensions of program behavior: *Arithmetic, Logic, State, and Boundary*. Specifically, the *Arithmetic* dimension tracks numerical precision in composite operations and complex indexing, e.g., `res = (a + b) * c % d`. The *Logic* dimension evaluates the resolution of compound boolean conditions and nested control flow decisions. To ensure reasoning depth, the *State* dimension monitors variable histories across long execution paths, whereas the *Boundary* dimension focuses on critical transition points, such as loop terminations. Notably, to increase challenge and reasoning depth, we ask LLMs to prioritize probes that span multiple dimensions, for example, *tracking state accumulation within nested logical branches*. Finally, human experts validate the intermediate probes for logical correctness, diversity, and complexity to ensure the rigor of CoRE. This process yields an intermediate state probe set for each problem $\mathcal{P}$, denoted as $\mathcal{Q}_{\mathcal{P}}$, which contains an average of 4.4 probes, with each probe spanning 2.2 dimensions on average.

3.3 Benchmark Statistics Analysis

Basic statistics. CoRE expands existing benchmarks by an order of magnitude with 3,050 samples. It raises the evaluation complexity, with an average cyclomatic complexity of 5.5 and an average of 7.8 test cases per coding problem, markedly surpassing existing benchmarks, as shown in Fig.2.

Implementation Invariance. As detailed in the basic statistic in Fig.4, CoRE consists of 91 coding problems, which contain a total of 391 code candidates, namely, an average of 4.3 implementations per coding problem. The diversity of the dataset is demonstrated in the Impl. Statistics in Fig.4, comprising 119 from OpenAI, 95 from Gemini, 88 from DeepSeek, 58 from Claude, and 31 from Qwen. The inner diversity of implementations within each coding problem is evidenced by a low 1-gram Jaccard similarity score of $\mathcal{J} = 0.6$ as listed in Fig.2.

Process Transparency. To evaluate LLMs in Process Transparency, we also formulate intermediate probes for each coding problem across four different dimensions. As shown in probe statistic in Fig.4, 124 probes involving *Arithmetic*, 289 involv-

ing *Logic*, 300 involving *State*, and 157 involving *Boundary*. These dimensions are not mutually exclusive, and they are often combined. For example, the *State* dimension is frequently interacted with others as illustrated in Fig.7. On average, each individual probe covers 2.2 dimensions, ensuring that the evaluation effectively challenges LLMs' process reasoning.

4 Evaluation Protocol

Unlike traditional evaluations, which typically focus on the output accuracy of a single canonical implementation, our protocol provides a holistic assessment of code reasoning by leveraging the diverse implementations $\mathcal{S}_\mathcal{P}$ and intermediate probes $\mathcal{Q}_\mathcal{P}$ constructed in Sec.3.1 and Sec.3.2.

4.1 Preliminaries

We first recall some background in code reasoning evaluation in this section.

Standard Code Reasoning. Traditional benchmarks (Chen et al., 2021; Jain et al., 2024) primarily focus on output prediction using a canonical Implementation c. This approach treats the reasoning process as a black box, where the model is evaluated on its ability to map an input x to the execution result $\hat{y}$ for the given c:

$$\hat{y} = \text{Pred}(c, x), \tag{4}$$

where $\hat{y}$ is expect to match the ground truth y.

Intermediate State Reasoning. Benchmarks such as REval (Chen et al., 2025) frame intermediate-state reasoning as a question-answering task, covering path reachability and variable prediction. Formally, given a code implementation c, a intermediate probes p, and a specific test input x, the model predicts the intermediate state as:

$$\hat{a} = \text{Pred}(p, c, x), \tag{5}$$

where $\hat{a}$ is the predicted answer to the intermediate probes p.

4.2 Implementation Invariance

A robust model should maintain consistency across diverse yet functionally equivalent implementations. For a given problem $\mathcal{P}$, we define its diverse code implementations in CoRE as a set $\mathcal{S}_\mathcal{P} = \{c_1, c_2, ..., c_n\}$. Implementation invariance is quantified via the **Strict Output Consistency** I, a binary indicator that is satisfied only if the model $\mathcal{M}$ correctly predicts the final output for every candidate solution across all associated test cases $\mathcal{T}_\mathcal{P}$:

$$I(\mathcal{P}) = \begin{cases} 1, & \forall c_i \in \mathcal{S}_\mathcal{P}, \mathcal{M}(c_i, {\mathcal{T}_\mathcal{P}}^x) = {\mathcal{T}_\mathcal{P}}^y \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

Additionally, we also define the soft accuracy I_s across multiple code implementations within each question $\mathcal{P}$:

$$I_s(\mathcal{P}) = \frac{\sum_{c_i \in \mathcal{S}_\mathcal{P}} \mathcal{M}(c_i, {\mathcal{T}_\mathcal{P}}^x) == {\mathcal{T}_\mathcal{P}}^y}{|\mathcal{S}_\mathcal{P}|} \tag{7}$$

4.3 Process Transparency

Moreover, reliable LLMs are expected to effectively demonstrate process transparency in code reasoning rather than simple input-output mapping. We define $\mathcal{Q}_\mathcal{P} = \{(q_1, a_1), (q_2, a_2), \ldots, (q_m, a_m)\}$ as the probing question set of question $\mathcal{P}$. Process transparency of code reasoning is calculated by **Process Fidelity Weight** W_s, the ratio of probing questions correctly answered by the model $\mathcal{M}$:

$$W_s(\mathcal{P}) = \frac{\sum_{(q_i, a_i) \in \mathcal{Q}_\mathcal{P}} \mathbb{1}(\mathcal{M}(q_i, \mathcal{P}) = a_i)}{|\mathcal{Q}_\mathcal{P}|}, \tag{8}$$

where $\mathbb{1}(\cdot)$ is the indicator function, and $|\mathcal{Q}_\mathcal{P}|$ represents the total number of probes for the problem $\mathcal{P}$. Additionally, we define a strict fidelity indicator $W \in \{0, 1\}$, where $W = 1$ if and only if all intermediate probing questions in $\mathcal{Q}_\mathcal{P}$ are answered correctly, i.e., $W_s = 1$, and $W = 0$ otherwise.

4.4 Reasoning Consistency Score

To rigorously quantify the robustness of code reasoning and penalize superficial heuristics, we propose the **Reasoning Consistency Score (RCS)**. The final RCS for problem $\mathcal{P}$ is computed as the product of the strict output consistency and the process fidelity weight:

$$RCS(\mathcal{P}) = I \times W_s \tag{9}$$

This formulation ensures that *any* failure in implementation invariance nullifies the final score, while scaling that score by the depth of its internal execution probing. By coupling these two dimensions, the RCS effectively filters out shortcut reasoning, distinguishing genuine code reasoning from a reliance on superficial heuristics.

Method	Impl. Invar.		Proc. Trans.		Overall		Impl. Invar.		Proc. Trans.		Overall	
	I	I_s	W	W_s	RCS	$Cons.$	I	I_s	W	W_s	RCS	$Cons.$
OpenAI Series												
			GPT-5						*o3*			
IO	84.62	95.44	0.00	18.96	15.93	0.15	83.52	95.74	0.00	9.73	8.90	0.16
CoT	**87.91**	**96.77**	26.37	64.93	55.75	0.27	82.42	94.74	28.57	67.60	54.25	0.31
CoC	86.81	96.04	20.88	59.14	50.24	0.30	**87.91**	**97.05**	**31.87**	67.55	58.24	0.35
RHDA-1	86.81	95.84	19.78	59.49	51.70	0.29	82.42	94.45	**31.87**	**70.46**	58.32	**0.36**
RHDA-2	86.81	95.83	**30.77**	**68.92**	**58.48**	**0.33**	85.71	95.75	29.67	68.28	**59.16**	0.33
DeepSeek Series												
			DeepSeek-V3.2						*DeepSeek-R1*			
IO	56.04	88.04	29.67	**67.95**	40.49	**0.56**	32.97	84.43	24.18	62.93	20.60	**0.58**
CoT	64.84	90.38	25.27	64.76	41.94	0.41	34.07	86.19	20.88	63.26	18.52	0.52
CoC	60.44	88.13	26.37	66.47	40.40	0.48	50.55	90.01	21.98	62.60	32.36	0.54
RHDA-1	62.64	88.50	27.47	66.67	41.48	0.45	70.33	90.85	**25.27**	**65.29**	44.34	0.37
RHDA-2	**71.43**	**91.18**	**30.77**	67.07	**48.70**	0.42	**72.53**	**91.51**	24.18	64.49	**47.16**	0.38
Claude Series												
			Claude-4.5						*Claude-3.7*			
IO	65.93	89.75	25.27	62.40	39.45	0.37	12.09	67.17	20.88	61.90	6.81	**0.69**
CoT	68.13	88.50	27.47	64.23	43.90	0.37	56.04	87.38	18.68	63.00	37.53	0.49
CoC	61.54	88.05	23.08	64.78	39.78	**0.44**	48.35	86.27	**27.47**	64.76	31.48	0.53
RHDA-1	65.93	89.39	28.57	65.71	44.34	0.43	56.04	86.04	24.18	**65.15**	39.73	0.53
RHDA-2	**81.32**	**93.71**	**29.67**	**68.74**	**54.45**	0.29	**64.84**	**88.75**	21.98	64.32	**42.77**	0.44
Gemini & Qwen Series												
			Gemini-2.5						*Qwen-3*			
IO	41.76	86.83	1.10	27.73	12.11	0.57	5.49	53.02	**0.00**	9.38	0.27	**0.95**
CoT	51.65	88.04	**30.77**	**64.74**	33.13	0.51	12.09	78.07	**0.00**	18.90	2.69	0.88
CoC	38.46	87.15	18.68	59.58	23.41	**0.58**	6.59	71.75	**0.00**	17.53	1.26	0.93
RHDA-1	61.54	87.82	21.98	62.88	38.90	0.47	16.48	76.39	**0.00**	**23.99**	4.18	0.84
RHDA-2	**71.43**	**91.78**	29.67	64.54	**44.25**	0.38	**47.25**	**84.22**	**0.00**	22.20	**10.66**	0.53

Table 1: Code Reasoning performance of baselines on the CoRE benchmark. The best performance is **bolded** and the second best is underline. **Across all frontier LLMs, models exhibit preferences over different code implementations, and frequently produce correct outputs despite incorrect intermediate execution states.**

5 Experiments

5.1 Experimental Setup

LLMs. We evaluate the code reasoning capabilities of eight leading LLMs, including GPT-5 (OpenAI, 2025a), o3 (OpenAI, 2025b), Claude-4.5 (Anthropic, 2025), DeepSeek-V3.2 (Liu et al., 2025), DeepSeek-R1 (Guo et al., 2025), and Qwen-3 (Yang et al., 2025).

Prompting Methods. To provide a comprehensive assessment, we employ diverse prompting strategies including standard Input-Output (IO), Chain-of-Thought (CoT) (Wei et al., 2022), and Chain-of-Code (CoC) (Li et al.). We also evaluate the RHDA framework (Zhao et al., 2025) as a strong reasoning baseline. In this setting, RHDA-1 and RHDA-2 correspond to the framework configured with one and two iterations, respectively. Details of the prompts and postprocessing are provided in Appendix B.

Metrics. We report the metrics defined in Sec. 4, including Strict Output Consistency (I), soft accuracy across implementations (I_s), Process Fidelity Weight (W_s), and strict process consistency (W). Our primary metric is the Reasoning Consistency Score (RCS), which assesses whether correct outputs are supported by consistent intermediate-execution reasoning across functionally equivalent implementations. We additionally report Cons., defined as $1-\text{MSE}(I, W_s)$, indicating how closely final output performance matches intermediate probe performance.

5.2 Results

Tab.1 presents a comprehensive comparison of code reasoning performance across various LLM series and prompting methods. We observe a significant performance gap, characterized by a disconnect between the reasoning step and the final predic-

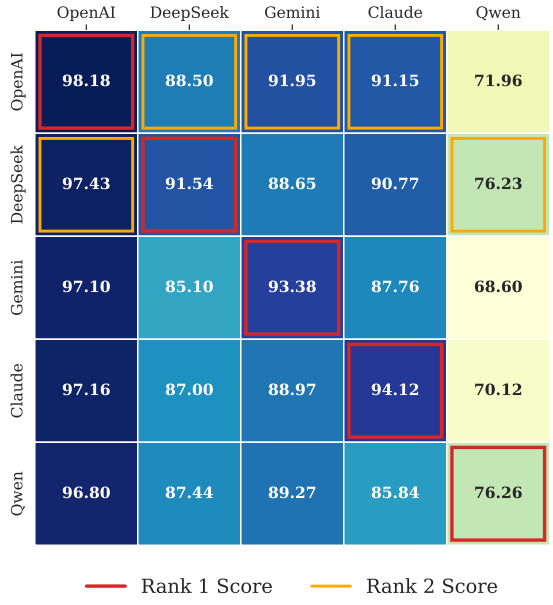


Figure 5: I_s scores performance heatmap across five LLM families. Red and gold boxes indicate the rank-1 and rank-2 I_s scores. **The pronounced diagonal pattern indicates a strong style bias.**

tion. For instance, in the IO setting, GPT-5 achieves a high strict output accuracy I of 84.62% but a low process fidelity weight W_s of 18.96%. This discrepancy indicates that models often hallucinate intermediate states despite generating correct final answers, a phenomenon we term superficial execution. Incorporating CoT substantially mitigates this issue, increasing W_s to 64.93% for GPT-5, which demonstrates that explicitly reasoning improves process fidelity. Regarding the reflection-based RHDA framework, implementation invariance is consistently enhanced, whereas gains in process transparency remain limited. RHDA-2 even underperforms RHDA-1 in some cases. This behavior reflects the inherent fragility of LLM reasoning. When feedback is derived solely from outputs, surface-level discrepancies in results can influence previous reasoning steps, leading models to erroneously revise previously correct intermediate states.

Additionally, a performance gap between I and I_s implicitly suggests that models struggle to maintain consistency across different implementations, revealing a potential robustness gap. A detailed analysis of this phenomenon is provided in Sec. 6.1.

6 Analysis

To gain deeper insights into reasoning behavior in CoRE, we analyze several factors that influence their code reasoning performance.

6.1 Robustness Gap and Style Overfitting

To examine the detailed implementation invariance of LLMs in CoRE, we analyze their performance consistency across diverse LLM families. Fig.5 shows a heatmap of evaluation scores with a clear diagonal dominance, indicating that models typically exhibit the highest proficiency with code generated by their own model family. For example, OpenAI models score 98.18 on OpenAI-generated implementations, noticeably higher than on code from other families. This pattern suggests that current LLMs are strongly influenced by familiar implementation styles, indicating a tendency toward stylistic overfitting. The lexical diversity of implementations, evidenced by a low 1-gram Jaccard similarity of $\mathcal{J} = 0.6$, further highlights that surface-level differences can trigger this inconsistent reasoning.

6.2 Investigating Superficial Execution

A central contribution of the CoRE benchmark is the identification of superficial execution, defined as the phenomenon where models arrive at correct final outputs without accurately reasoning about intermediate states. For instance, under the native IO prompting, GPT-5 achieves a high strict output accuracy $I = 84.62$ but a remarkably low process fidelity weight W_s of only 18.96. This discrepancy yields a poor RCS score of 15.93, as it penalizes such heuristic reasoning, proving that RCS effectively distinguishes genuine code reasoning from superficial execution.

6.3 Analysis of Dimensional Challenges

To analyze how four dimensions challenge LLMs in process transparency evaluation, we report model performance across *Arithmetic, Logic, State, and Boundary* in Tab.2. The Arithmetic dimension exposes a severe lack of numerical precision. Under IO prompting, GPT-5 and o3 achieve W_S scores of only 11.11 and 9.26, respectively. This suggests that native models often bypass exact arithmetic. Structural fragility is evident in the limited performance of Logic and Boundary dimensions, which yield average W_s of 44.96 and 47.17, respectively. These results indicate that models often

Model	IO				CoT				CoC			
	A	L	S	B	A	L	S	B	A	L	S	B
GPT-5	11.11	25.00	20.78	21.33	68.52	74.19	77.27	76.00	74.07	63.71	68.83	70.67
o3	9.26	10.48	11.69	17.33	**77.78**	**75.00**	**80.52**	**81.33**	**77.78**	**73.39**	**77.92**	78.67
Claude-4.5	68.52	**72.58**	72.73	**78.67**	72.22	71.77	74.68	78.67	70.37	72.58	74.68	78.67
Claude-3.7	75.93	67.74	72.08	72.00	**77.78**	70.16	73.38	72.00	75.93	**73.39**	74.68	73.33
DeepSeek-V3.2	**79.63**	71.77	**76.62**	77.33	74.07	70.97	74.03	76.00	74.07	71.77	77.27	**80.00**
DeepSeek-R1	75.93	69.35	71.43	68.00	72.22	72.58	72.73	77.33	75.93	67.74	70.13	73.33
Gemini-2.5	24.07	32.26	31.82	30.67	**77.78**	69.35	72.08	77.33	70.37	67.74	71.43	72.00
Qwen-3	11.11	10.48	11.04	12.00	18.52	24.19	18.18	16.00	27.78	22.58	20.78	18.67
Average	44.45	44.96	46.02	47.17	67.36	66.03	67.86	69.33	68.29	64.11	66.97	68.17

Table 2: Comparison of model performance across four problem complexity levels (A: Arithmetic, L: Logical, S: State, B: Boundary). The process fidelity weight W_s is reported. The best performance is **bolded** and the second best is underline. **Overall, CoT and CoC substantially improve intermediate probe performance across all complexity dimensions compared to direct IO.**

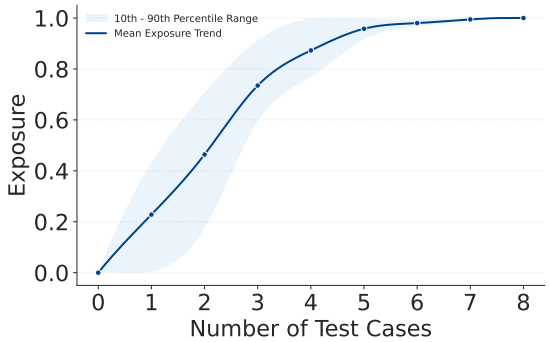


Figure 6: Exposure trend across increasing test cases. The solid blue line denotes the mean exposure ratio, while the shaded region represents the 10 th to 90th percentile distribution. **The model's performance shows initial variability before converging to saturation as the number of test cases increases.**

struggle to handle nested conditions and critical transition points. Furthermore, the State dimension, which monitors variable histories across deep execution paths, shows an average W_S of only 46.02, revealing an inability to maintain a persistent internal memory of variable updates. This poor performance stems from the fact that the State dimension frequently interacts with the other three dimensions as illustrated in Fig.7, exposing their fundamental inability to maintain a persistent memory of variable updates.

6.4 Impact of Prompting Method

Although CoT, CoC, and RHDA enhance implementation invariance and process transparency evaluation, they still fail to address robustness gaps and superficial execution thoroughly. Specifically, considering RHDA is an iterative reflection framework, we investigate why RHDA-2 sometimes underperforms RHDA-1 as observed in Tab.1. We find that when a correct final output is mistakenly judged as incorrect in RHDA, models readily revise previously correct intermediate reasoning simply in response to this erroneous feedback, highlighting the existence of superficial execution.

6.5 Soundness of Test Cases

To improve computational efficiency, we employ a coverage-based reduction strategy to identify the minimal subset of test cases that maintains maximal execution coverage. This optimization results in an average of 7.8 test cases per instance, which significantly raises the evaluation ceiling compared to prior benchmarks. The exposure trend illustrated in Fig.6 demonstrates that model performance initially varies but eventually converges to saturation as the number of test cases increases.

7 Conclusion

We introduce CoRE, a benchmark designed to evaluate code reasoning through implementation invariance and process transparency. Our evaluation of eight leading LLMs identifies a pronounced robustness gap and the phenomenon of superficial execution in existing frontier LLMs. To quantify this, we propose RCS, a composite protocol that penalizes heuristic shortcuts and rewards genuine code reasoning. Our experimental findings highlight the fragility of current LLMs, underscoring the critical role of CoRE in facilitating a deeper exploration for code reasoning.

8 Limitations

Despite its rigor, the CoRE benchmark has several limitations that provide directions for future work. First, our current implementation and execution behavior tracing are primarily focused on the Python programming language. While CoRE derives instances from well-established benchmarks like HumanEval and LiveCodeBench, these source datasets may still be susceptible to potential training data leakage. Second, although we incorporate a human verification stage to ensure logical validity, the reliance on expert verification limits the rapid and automated scaling of the dataset. Finally, the distribution of probing questions across the four dimensions, Arithmetic, Logic, State, and Boundary, is not perfectly uniform due to the inherent complexity of intermediate execution.

References

Anthropic. 2025. Introducing claude sonnet 4.5. Technical report.

Claas Beger and Saikat Dutta. 2025. Coconut: Structural code understanding does not fall out of a tree. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 128–136. IEEE.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*.

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 *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1869–1881. IEEE.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Jun Gao, Yongqi Li, Ziqiang Cao, and Wenjie Li. 2025a. Interleaved-modal chain-of-thought. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 19520–19529.

Jun Gao, Yun Peng, and Xiaoxue Ren. 2025b. \texttt {ReMind}: Understanding deductive code reasoning in llms. *arXiv preprint arXiv:2511.00488*.

Jun Gao, Qian Qiao, Tianxiang Wu, Zili Wang, Ziqiang Cao, and Wenjie Li. 2025c. Aim: Let any multimodal large language models embrace efficient in-context learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 3077–3085.

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

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. oppt-4o system card. *arXiv preprint arXiv:2410.21276*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*.

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

Tamera Lanham, Anna Chen, Ansh Radhakrishnan, Benoit Steiner, Carson Denison, Danny Hernandez, Dustin Li, Esin Durmus, Evan Hubinger, Jackson Kernion, and 1 others. 2023. Measuring faithfulness in chain-of-thought reasoning. *arXiv preprint arXiv:2307.13702*.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and 1 others. Chain of code: Reasoning with a language model-augmented code emulator. In *Forty-first International Conference on Machine Learning*.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. 2024b. Codemind: A framework to challenge large language models for code reasoning. *arXiv e-prints*, pages arXiv–2402.

OpenAI. 2025a. Gpt-5 system card: Capabilities and safety evaluations. Technical report.

OpenAI. 2025b. Openai o3 and o4-mini system card. Technical report.

Zhensu Sun, Xiaoning Du, Fu Song, Shangwen Wang, Mingze Ni, Li Li, and David Lo. 2025. Don't complete it! preventing unhelpful code completion for productive and sustainable neural code completion systems. *ACM Transactions on Software Engineering and Methodology*, 34(1):1–22.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel Bowman. 2023. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. *Advances in Neural Information Processing Systems*, 36:74952–74965.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. 2022. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, and 1 others. 2024. Large language models are not fair evaluators. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9440–9450.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Wikipedia. 2026. Jaccard index. Technical report.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Yuze Zhao, Tianyun Ji, Wenjun Feng, Zhenya Huang, Qi Liu, Zhiding Liu, Yixiao Ma, Kai Zhang, and Enhong Chen. 2025. Unveiling the magic of code reasoning through hypothesis decomposition and amendment. In *The Thirteenth International Conference on Learning Representations*.

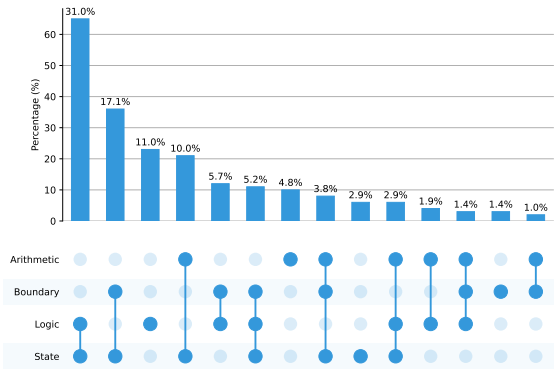


Figure 7: The interaction of intermediate probes.

Table 3: Overview of Large Language Models used in different stages of the study. **II** indicates implementation invariance, and **IP** represents intermediate probing.

Model	II	IP	Eval
GPT-5		✓	✓
o3			✓
o1	✓		
4o-mini	✓		
DeepSeek-V3	✓		
DeepSeek-V3.2		✓	✓
DeepSeek-R1	✓		✓
Claude-3.7	✓		✓
Claude-4.5		✓	✓
Qwen-3-235B	✓	✓	✓
Gemini-2.5	✓	✓	✓

A Benchmark Construction Details

A.1 Details of Identical Implementation Generation

We leverage an ensemble of state-of-the-art LLMs to generate diverse but functionally equivalent code implementations for problems sourced from HumanEval and LiveCodeBench. As shown in Table 3, seven models, such as o1, 4o-mini, DeepSeek-V3, and Qwen-3, were primarily utilized for this stage. During this stage, we set the temperature to 0.7. In the implementation cleaning process, the cyclomatic complexity and Jaccard similarity threshold τ_{cc} and τ_{sim} are set to 3.0 and 0.7, respectively.

Code Generation Prompt

You are an expert Python programmer. You will be given a question (problem specification) and will generate a correct Python program that matches the specification and passes all tests.

Question:
{question_content}

Format: You will use the following starter code to write the solution to the problem and enclose your code within delimiters.
“‘python
{starter_code}
“‘

Answer: (use the provided format with backticks)

A.2 Details of Intermediate Probing

To alleviate the impact of LLM bias, we employ GPT-5, DeepSeek-V3.2, Claude-4.5, DeepSeek-R1, and Qwen-3 in a cycle to formulate probing questions in this stage. We encourage questions to be diverse to challenge LLMs, and we thus slightly increase the temperature to 0.8. The following few-shot examples are provided to the model to demonstrate the desired reasoning depth and output format.

Reference Examples for Code Reasoning

Example 1
Input Source Code:

```
def update_arr(arr, factor):
  for i in range(len(arr)):
    if i > 0:
      arr[i] = (arr[i] *
      factor + arr[i-1]) % 10
  return arr
```

Input Execution Trace:
[Step 1] Line 2: for i in range(len(arr)) (Iter 1: i=0) | locals: ’arr’: [1, 5, 9], ’factor’: 2
...
[Step 5] Line 3: if i > 0 (Evaluated: True)
[Step 6] Line 5: arr[i] = (arr[i] * factor + arr[i-1]) % 10 | locals: ’arr’: [1, 1, 9]

Output (JSON):

```
{
  "target_step_index": 6,
  "complexity_type": "Arithmetic + State",
  "question": "In the 2nd iteration (i=1),
      what is the exact value assigned to
      'arr[1]' on line 5?",
  "ground_truth": "1",
}
```

Example 2
Input Source Code:

```
def matrix_sum(matrix):
  total = 0
  for row in matrix:
    for val in row:
      if val == -1: break
      total += val
    return total
```

Input Execution Trace:
...
[Step 10] Line 3: for row in matrix (Iter 2) | locals: {'row': [10, -1, 5], 'total': 6}
[Step 11] Line 4: for val in row (Iter 1) | locals: {'val': 10}
[Step 12] Line 5: if val == -1 (Evaluated: False)
[Step 13] Line 6: total += val | locals: {'total': 16}
[Step 14] Line 4: for val in row (Iter 2) | locals: {'val': -1}
[Step 15] Line 5: if val == -1 (Evaluated: True)
[Step 16] Line 5: break | locals: {'total': 16}
Output (JSON):

```
{
  "target_step_index": 16,
  "complexity_type": "Boundary + State",
  "question": "Given the input row [10, -1,
       5], what is the value of variable '
      total' immediately after the break
      statement executes on line 5?",
  "ground_truth": "16",
}
```

Then we combine the above reference examples with the following instruction, providing LLMs with sample questions targeting four dimensions: Arithmetic, Logic, State, and Boundary

Intermediate Probing Prompt

 # System Instruction
You are an expert Code Logic Auditor. Your goal is to create a "Challenging Code Reasoning Benchmark" by analyzing execution traces.

Mission:
Identify the **most error-prone** step in the provided execution trace and formulate a question about it. We want to test the model's ability to handle **complexity**.

Selection Criteria:

1. **Complex Arithmetic:** Lines with multiple operators (e.g., `res = (a + b) * c % d`) or list indexing with math.
2. **Loop Boundaries:** The last iteration of a loop, or the state immediately after a loop finishes.
3. **Nested Logic:** Steps inside a nested loop or a nested `if` block where context is deep.
4. **Compound Conditions:** Boolean evaluations involving `and`, `or`, `not`.
5. **State Accumulation:** A variable modified multiple times (e.g., `total` after the 5th iteration).

Rules:

- **Answer Integrity:** The 'ground_truth' must be extracted EXACTLY from the provided trace. Do not compute it yourself.
- **Precision:** The question must specify the exact context (e.g., "At the end of the 3rd iteration...", "In the evaluation of the condition on line 5...").

 # Few-Shot Examples
⟨Reference Examples from Appendix A.2 are inserted here⟩

 # Current Task
Input Source Code:
"'python
{source_code}

“‘

Input Execution Trace:
{execution_trace}

A.3 Details of Human Verification

To ensure the logical validity and diversity of the benchmark, five experts executed a multi-stage verification protocol. During the implementation variance phase, the experts manually refined samples with a Jaccard similarity within the range of $[0.7, 0.9]$ to recover implementations that exhibited high 1-gram similarity but featured distinct structural logic or algorithmic strategies.

Regarding intermediate reasoning traces, the experts evaluated logical correctness and the rationality of state transitions while simultaneously removing redundant questions to maximize diversity. This rigorous calibration process required approximately 110 human-hours and resulted in a finalized pool of 391 expert-verified implementations and 400 intermediate questions across 91 unique questions. The effort achieved high inter-annotator agreement with Cohen's Kappa $\kappa = 0.82$ for the implementation variance phase and $\kappa = 0.86$ for the intermediate probing phase.

B Details of Experimental Baselines

B.1 Prompting Methods

We evaluated the models using three primary strategies: Input-Output (IO), Chain-of-Thought (CoT), and Chain-of-Code (CoC). For CoT, we specifically implemented a two-shot approach to standardize the reasoning "scratchpad" across all frontier models.

Standard Output Prompt

You are given a Python function and an assertion containing an input to the function. Complete the assertion with a literal (no unsimplified expressions, no function calls) containing the output when executing the provided code on the given input, even if the function is incorrect or incomplete. Provide the full assertion with the correct output, following the examples.

```
def f(s):
  return s + "a"
assert f("x9j") == ??
# Answer:
assert f("x9j") == "x9ja"

{code}
assert {func_name}({x}) == ??
# Answer:
```

Chain-of-Thought Prompt

You are given a Python function and an assertion containing an input to the function. Complete the assertion with a literal (no unsimplified expressions, no function calls) containing the output when executing the provided code on the given input, even if the function is incorrect or incomplete. Execute the program step by step before arriving at an answer, and provide the full assertion with the correct output, following the examples.

```
def f(s):
    s = s + s
    return "b" + s + "a"
assert f("hi") == ??
```

Let's execute the code step by step:

- 1. The function f is defined, which takes a single argument s.
- 2. The function is called with the argument "hi", so within the function, s is initially "hi".
- 3. Inside the function, s is concatenated with itself, so s becomes "hihi".
- 4. The function then returns a new string that starts with "b", followed by the value of s (which is now "hihi"), and ends with "a".
- 5. The return value of the function is therefore "bhihia".

```
Answer:
assert f("hi") == "bhihia"

{code}
assert {func_name}({x}) == ??
```

Let's execute the code step by step:

Chain-of-Code Prompt

You are given a Python function and an assertion containing an input to the function. Complete the assertion with a literal (no unsimplified expressions, no function calls) containing the output when executing the provided code on the given input, even if the function is incorrect or incomplete. Execute the program step by step before arriving at an answer, and provide the full assertion with the correct output, following the examples.

```
def f(s):
    s = s + s
    result = "b" + s + "a"
    return result
assert f("hi") == ??
```

[TRACE]
state: {}
line: f("hi")
explanation: Python execution.
delta state: 's': 'hi'
line: s = s + s
explanation: Python execution.
delta state: 's': 'hihi'
line: result = "b" + s + "a"
explanation: Python execution.
delta state: 'result': 'bhihia'
line: return result
explanation: Python execution.
delta state:
[/TRACE]
Answer:

```
assert f("hi") == "bhihia"
```

```
{code}
assert {func_name}({x}) == ??
```

[TRACE]

Case Study 1:

```
def specialFilter(nums):
  count = 0

  for num in nums:
    if num > 10:
      num_str = str(abs(num))
      first_digit = int(num_str[0])
      last_digit = int(num_str[-1])

      if first_digit % 2 != 0 \
         and last_digit % 2 != 0:
        count += 1
  return count
```

Envs: {nums: [71, -2, -33, 75, 21, 19], count: 0}

Step	num	num>10?	first_digit	last_digit	Both odd?	count	TRACE(count)
1	71	True	7	1	True	1	✅Satisfy
2	-2	False	-	-	-	1	✅Keep: -2 < 10
3	-33	True	3	3	True	2	Satisfy: abs(-33) > 10
4	75	True	7	5	True	3	✅Satisfy
5	21	True	2	1	False	3	✅Keep: 2 is even
6	19	True	1	9	True	4	✅Satisfy
7	**Return 4** (3)						

Case Study 2:

```
def minSubArraySum(nums):
  min_sum = float('inf')
  cur_sum = 0

  for num in nums:
    cur_sum += num
    if cur_sum < min_sum:
      min_sum = cur_sum
    if cur_sum > 0:
      cur_sum = 0

  return min_sum
```

Envs: {nums: [100, -33, 32, -1, 0, -2], min_sum: inf, current_sum: 0}

Step	num	cur_sum	min_sum	cur_sum < min_sum?	cur_sum > 0?	TRACE(min_sum)
1	100	100→0	inf	False	True	Keep:100 > inf
2	-33	0→-33	-33	True	False	✅Update: -33 < inf
3	32	-33→-1	-33	False	False	✅Keep: -1 > -33
4	-1	-1→-2	-33	False	False	✅Keep: -2 > -33
5	0	-1→-2	-33	False	False	✅Keep: -2 > -33
6	-2	-2→-4	-33→-4	True	False	Update: -4 < -33
7	**Return -4**(-33)					

Figure 8: **Case studies of LLMs on the CoRE benchmark.** These emperical examples illustrate the model's reasoning process in output and intermediate state reasoning, highlighting common failures in handling numbers and logical conditions.