Do Code Semantics Help? A Comprehensive Study on Execution Trace-Based Information for Code Large Language Models

Jian Wang[1,2], Xiaofei Xie[1], Qiang Hu[3*], Shangqing Liu[4], Yi Li[2]

[1]Singapore Management University, Singapore
[2]Nanyang Technological University, Singapore
[3]Tianjin University, China
[4]State Key Laboratory for Novel Software Technology, Nanjing University, China

Abstract

Code Large Language Models (Code LLMs) have opened a new era in programming with their impressive capabilities. However, recent research has revealed critical limitations in their ability to reason about runtime behavior and understand the actual functionality of programs, which poses significant challenges for their post-training and practical deployment. Specifically, Code LLMs encounter two principal issues: (1) a lack of proficiency in reasoning about program execution behavior, as they struggle to interpret what programs actually do during runtime, and (2) the inconsistent and fragmented representation of semantic information, such as execution traces, across existing methods, which hinders their ability to generalize and reason effectively. These challenges underscore the necessity for more systematic approaches to enhance the reasoning capabilities of Code LLMs. To address these issues, we introduce a generic framework to support integrating semantic information (e.g., execution trace) to code task-relevant prompts, and conduct a comprehensive study to explore the role of semantic information in enhancing the reasoning ability of Code LLMs accordingly. Specifically, we focus on investigating the usefulness of trace-based semantic information in boosting supervised fine-tuning (SFT) and post-phase inference of Code LLMs. The experimental results surprisingly disagree with previous works and demonstrate that semantic information has limited usefulness for SFT and test time scaling of Code LLM.

1 Introduction

Code large language models (Code LLMs) have emerged as prominent programming assistants, demonstrating remarkable performance across various coding tasks, including program repair (Xia et al., 2022), code generation (Liu et al., 2023b), and code summarization (Jain et al., 2020). Recently, several Code LLMs have been introduced, each characterized by distinct training schemes. For instance, Llama3.1 (Dubey et al., 2024; Roziere et al., 2023), is fine-tuned with code infilling tasks and long code input contexts, complemented by an instruction fine-tuning process. Similarly, DeepSeek-Coder (Guo et al., 2024) is trained on more than 2 trillion tokens using a fill-in-the-blank task to enhance its code generation capabilities. These models focus on learning contextual information from code and docstrings, advancing their general understanding of code (Chen et al., 2024; Ni et al., 2024; Ding et al., 2024).

However, these approaches predominantly capture the static dimensions of code (e.g., tokens and context), while neglecting the dynamic semantics crucial for a comprehensive understanding of code. Recent studies have highlighted this limitation, revealing that even state-of-the-art models like GPT-4 struggle to reason about runtime behaviors of code (Chen et al., 2024). Understanding code semantics and accurately predicting runtime behavior is critical, particularly for practical coding applications that require semantic understanding (Ni et al., 2024; Ding et al., 2024). This underscores the urgent need to investigate methodologies that enhance the reasoning capabilities of Code LLMs for better semantic understanding, such as coverage prediction (Chen et al., 2024) and output prediction (Chen et al., 2024; Liu et al., 2023a) abilities.

To address these challenges, two primary methods have been proposed to bolster the reasoning capabilities of Code LLMs in code generation tasks: 1) *inference with semantic information* – integrating feedback, such as error information to the inputs (or outputs) at inference time, and 2) *fine-tuning with semantic information* – directly enhancing the models' intrinsic reasoning by integrating semantically enriched fine-tuning data, thereby enabling improved predictions (Ni et al., 2024; Ding

*Corresponding Author: qianghu@tju.edu.cn

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 10367–10385

et al., 2024). In this work, we consider both approaches and explore whether and how incorporating code semantic information can enhance the performance of Code LLMs. Specifically, we investigate evaluate their effectiveness across various code-related tasks.

The core challenge lies in identifying and collecting appropriate semantic data for training or inference to improve the semantic comprehension of Code LLMs. Recent efforts (Ding et al., 2024; Ni et al., 2024) have begun to address this by fine-tuning models using dynamic data, such as execution behaviors, to enhance semantic understanding. While these approaches have shown promise, they employ diverse semantic representations, such as natural language descriptions of programs or execution traces. There remains a lack of 1) a unified study and tool that supports all semantic representations, and 2) systematic understanding regarding how different training data compositions, particularly in terms of semantic representations, impact code reasoning and generation capabilities.

To this end, we propose and implement a generic framework that facilitates the generation of multiple types of semantic representations, supporting post-training, one-time inference, and the scaling of test-time computation during inference. Based on this framework, we conduct a systematic study to explore the efficacy of semantic information in boosting code generation. Specifically, we integrate different semantic representations (i.e., different code execution traces) into the input data (prompt) during both SFT time and inference time to assess their impact. Additionally, we examine the influence of different training strategies (e.g., parameter-efficient fine-tuning) on the effectiveness of semantic information. Different from previous research findings (Ni et al., 2024; Ding et al., 2024), our experimental results demonstrate that integrating existing semantic information into the input prompt has limited benefits to the performance of tuned Code LLMs.

To summarize, the main contributions of this paper are:

- We introduce the first generic framework that supports different types of code semantic representations. Based on this framework, we construct and open-source a high-fidelity dataset featuring diverse execution behavior representations, including bug-patch function pairs, unit tests, and multiple semantic layers. The dataset and all related implementations are publicly available on our website [1].
- We conduct a comprehensive study to explore the effectiveness of semantic information in enhancing both SFT and inference of code LLMs.
- We summarize multiple findings, such as semantic information integrated in the input does not positively contribute to the inference.

2 Problem Statement

We address the problem of enhancing the code generation capabilities of Code LLMs by integrating code semantic information into the input prompts. Current Code LLMs primarily rely on static text data, which often fails to capture the nuanced semantics crucial for thorough code understanding (Wei et al., 2023, 2024; Abdin et al., 2024). Inspired by the practices of human developers, who iteratively refine code through reasoning and semantic assessment rather than relying solely on dynamic testing or runtime feedback, we note that such reasoning and refinement processes are largely absent in existing models (Ni et al., 2024; Ding et al., 2024). There are two main sub-problems we are interested in for Code LLMs, 1) fine-tuning with semantic information, and 2) inference with semantic information.

(1) Fine-tuning with semantic information. Our Code LLM fine-tuning paradigm focuses on a *repair-based* fine-tuning framework that leverages a carefully curated dataset $\mathcal{D} = \{(x, y) \mid x = (b, r),\ y = a\}$ where (x, y) is the input-output pair for model fine-tuning, b denotes the buggy code fragment, r is execution-trace rationale, and a represents the patched code. We selected the code repair task because models often fail to produce correct code in a single attempt, requiring the refinement of the output until it is correct. A central question is how to encode the reasoning signals r so that the Code LLM can learn better code generation capability.

(2) Scaling Inference through Semantic Refinement. Emulating human "work-and-check" practices at inference time—iteratively refining candidate solutions and verifying each step—can substantially improve an LLM's accuracy under a fixed but non-trivial test-time compute budget N (Li et al., 2025). The key question is: *Given a fixed inference-time compute budget N, to what extent*

[1] https://github.com/tracewise-probing/tracewise_probing

can an LLM improve its performance when prompts are enhanced with semantic representations? In this paradigm, a search-based computation strategy θ specifies how to (1) propose candidate solutions incrementally, (2) verify or score each partial output (e.g., via code execution or a reward model), and (3) refine solutions based on feedback—all within the budget N. Formally, following (Snell et al., 2024), for a given query q, the final output y is drawn from

$$y \sim \text{Target}(\theta,\ q,\ N,\ \text{Verify}) \quad (1)$$

where $\text{Target}(\theta, q, N, \text{Verify})$ is a test-scaling framework which iterates through proposing, verifying, pruning, and refining partial solutions until the budget N is exhausted. If $y^*(q)$ denotes the ground-truth correct answer for q, we measure accuracy via the indicator $\mathbb{1}_{\{y=y^*(q)\}}$. In general, we define the test-time compute-optimal strategy $\theta_q^*(N)$ as the one that maximizes the expected probability of generating the correct answer:

$$\theta_q^*(N) = \arg\max_{\theta}\Big(\mathbb{E}_{y\sim\text{Target}(\theta,N,q,\text{Verify})}\big[\mathbb{1}_{y=y^*(q)}\big]\Big) \quad (2)$$

Here, θ may control how many refinement steps to run, which candidate paths to verify, subject to the budget N. Verification signals (such as code execution) enable the model to discard incorrect paths or improve partially correct ones, while iterative refinement uses feedback to converge on better outputs. By strategically allocating test-time resources, a trace-based verify-and-refine loop can substantially boost accuracy without additional training.

3 Evaluation Framework

3.1 Overall Design

This work aims to investigate the impact of various execution-trace based semantics and their representations, denoted as r, on the performance of Code LLMs. Following recent works (Ding et al., 2024; Chen et al., 2024; Ni et al., 2024), we consider high-level program descriptions and low-level execution traces as potential semantic components of r. Different trace representations can significantly influence fine-tuning and inference performance, prompting us to explore which representations best enhance Code LLMs. To achieve the goal, we design and build a framework for the generation and evaluation of automatic semantic representations, which is outlined in Figure 1. The framework consists of three main components, data construction, Code LLM fine-tuning, and inference.

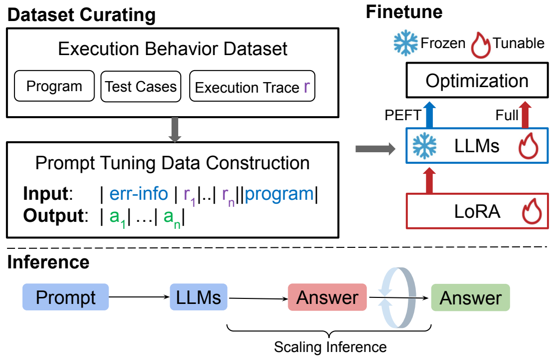

Figure 1: Paradigm of our framework. Initially, we curate prompt-tuning data from the Execution Behavior Dataset by extracting runtime execution messages, which are then formalized using a trace adapter. Subsequently, we employ parameter-efficient fine-tuning techniques, such as LoRA, or opt for full parameter fine-tuning to train the foundation model. In the output above, the purple text denotes the rationale, while the green text represents the answer. During the inference phase, the framework supports Scaling Inference to enhance the capability of LLMs.

Fine-Tuning Data Construction. Our dataset contains two parts, a program repair dataset and other downstream datasets. The program repair dataset is used to help Code LLMs learn semantic information that is integrated into the buggy/correct code pairs. Specifically, the input b includes *program descriptions*, *test cases*, which are associated with the buggy code (e.g., an instruction to generate code, a test case fails on the buggy code), and semantic information r. For r, our framework automatically generates *execution traces* as self-explanations for bug fixes (i.e., patch code a) based on the Trace Adapter component. Specifically, Trace Adapter first runs the code using a compiler to collect the raw execution trace. After that, it transfers the raw trace to various trace representations. Currently, Trace Adapter supports five types of representative reasoning-based code semantic information, Scratchpad (Nye et al., 2021), NExT (Ni et al., 2024), SemCoder (Ding et al., 2024), CodeExecutor (Liu et al., 2023a)), and Concise which is a new variant of CodeExecutor designed by us. Based on our data construction pipeline, we prepare and open-source a new dataset encompassing buggy/patched code, unit tests, program descriptions, and various trace types, as existing datasets (Ni et al., 2024; Ding et al., 2024) do not meet these requirements.

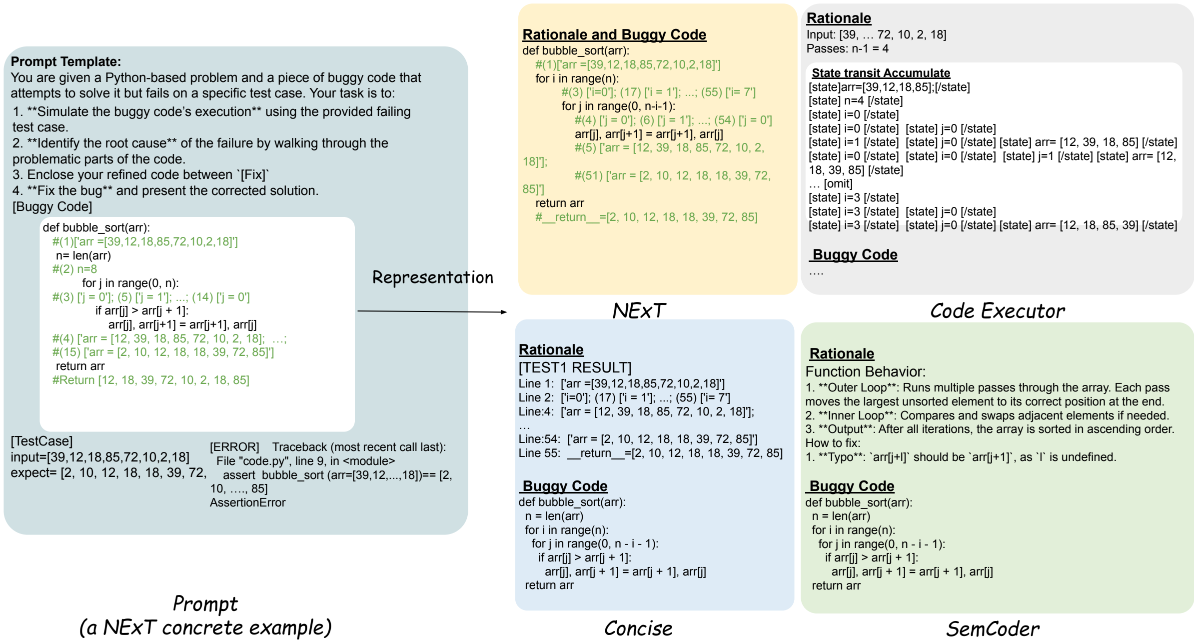

Figure 2: A concrete prompt example (left panel) and examples of different semantic representations (right panel).

Supervised Fine-Tuning. Our framework supports a two-stage Code LLM fine-tuning process. Concretely, it first fine-tunes Code LLMs using a repair-based workflow (as introduced in Section 2) to force models to learn the semantic information hidden in the difference between the execution traces of the buggy and correct codes. Then, other downstream task datasets, such as code generation datasets are used in the second phase to help Code LLMs learn domain-specific knowledge.

Test Time Scaling. Our framework supports two test time scaling strategies, Sequential Scaling and Parallel Scaling (Khattab et al., 2024; Li et al., 2025; Wang et al., 2025b; Shi and Jin, 2025).

Sequential Scaling iteratively generates outputs based on the feedback from the previous round. Given an input prompt, the Code LLM first samples N candidate programs and executes each with an external checker (e.g. a Python interpreter or trace-format adapter). If any candidate passes all public test cases, that program is returned immediately; otherwise, the checker emits trace-based diagnostics for every failing candidate. These diagnostics are added to the prompt, prompting the LLM to generate a new revised candidate in the next round. This self-debug (Chen et al., 2023) cycle repeats until a correct solution is found or a predefined budget of $R_{\max}$ rounds is reached, exploring at most $N \times R_{\max}$ candidate programs while continuously steering the model with execution-trace feedback. Different from *Sequential Scaling*, *Parallel Scaling* generates multiple solutions at once and selects one accordingly. More implementation details can be found in the Appendix A.

3.2 Trace Representation Adapters

The key component in the framework is the trace adapter. Execution traces can be represented in various ways. Our adapter supports various distinct execution representations collected from existing works. Figure 2 illustrates examples of each execution representation.

NExT integrates execution traces directly within the code as inline comments. It identifies variables present in each line of code and appends changes in these variables as comments following the respective line, providing a seamless integration of code and state information.

SemCoder utilizes natural language to describe execution traces. It provides a line-by-line explanation of code execution, including aspects such as execution status, variable changes, and input-output relationships. For instance, as shown in Figure 2, SemCoder describes the function signature of 'bubble_sort' and specifies that the 'arr' argument accepts only a list of integers, offering a detailed, human-readable explanation.

Code Executor records the state changes of variables in each line, similar to NExT, but presents these execution traces separately from the code, emphasizing a clear distinction between code and execution states.

Concise is a variant of Code Executor, which records the value changes of variables line-by-line and presents the trace separately from the code context as shown in Figure 2. Unlike Code Executor, Concise ignores variables whose values remain unchanged during the execution of a specific line, simplifying the representation. For example, in line 4, the variable 'n=10' is omitted in Concise.

4 Experiment Design

Based on our framework, we conduct a comprehensive study to explore the usefulness of code semantic information for Code LLMs during fine-tuning and test-time scaling, respectively. Specifically, for the fine-tuning part, we utilize our constructed datasets to fine-tune Code LLMs first, and then evaluate their capabilities on different programming tasks using the basic evaluation paradigm. For the test-time scaling evaluation, we employ different test-time scaling strategies to help assess the ability of fine-tuned Code LLMs from the previous step to investigate the usefulness of semantic information.

Datasets. Table 1 summarizes datasets used in our study. As fine-tuning contains two stages, it requires two types of datasets. For the first stage, we utilize the framework to prepare different types of semantic information-covered datasets. For the second stage, we use the datasets provided by Semcoder (Ding et al., 2024) for the fine-tuning of downstream tasks. Regarding the evaluation, we employ widely used datasets to assess the capability of Code LLMs, including Code-Synthesis tasks (HumanEval (Chen et al., 2021a), MBPP (Liu et al., 2023b), LiveCodeBench(LCB) (Jain et al., 2024), BigcodeBench (Zhuo et al., 2024)), two repair tasks (HE-R (Muennighoff et al., 2023a) and MBPP-R collected by us from EvalPlus's MBPP release, regarding their test-failure generation as a source of buggy code.), and two reasoning tasks (CRUXEval-I and CRUXEval-O) (Gu et al., 2024). For the test-time scaling evaluation, we use LiveCodeBench in our experiments, following (Li et al., 2025). We adapt different trace representations into the prompts while keeping the same dataset and codebase.

Models. For fine-tuning, our study considers three representative LLMs: DeepSeek-Coder (deepseek-6.7b-base), LLaMA (Llama3.1-8B), and Gemma2 (gemma2-9b). For the inference, we cover two more closed-source models, GPT-4o and Deepseek-Chat(V3). In addition, for the evaluation of reasoning ability, we include two more models oriented to reasoning, microsoft/phi-4 (Abdin et al., 2024) and AIDC-AI/Marco-o1 (Zhao et al., 2024).

Configuration. Input prompts are produced automatically by the DSPY framework (Khattab et al., 2024); full templates can be found in Appendix E. All code executes inside a sandbox following the safety procedures of (Chen et al., 2021a) to guard against malicious generations. For detailed experimental configurations, please refer to Appendix A. Besides, we put the results of HumanEval and HE-R in Appendix C D due to the page limitation.

5 Result Analysis

5.1 Fine-Tuning with Semantic Information

Comparison between fine-tuning with and without semantic information. Table 2 summarizes the performance of Code LLMs after fine-tuning. Surprisingly, the results demonstrate that fine-tuning with trace information cannot enhance the performance of Code LLMs. Specifically, for *Program Repair* tasks, compared to models trained without traces (w/o trace), only SemCoder contributes to fine-tuning but with limited improvements (from 0.3 to 1.4). Similarly, the results of *Code Synthesis* tasks show that semantic information cannot significantly enhance the code generation ability of Code LLMs. In more than half of the cases (7 out of 9 cases), fine-tuning without trace information achieves the best results. Besides, there is also a similar phenomenon in the *Reasoning* tasks.

Takeaway: Integrating trace-based semantic information into the fine-tuning datasets cannot significantly enhance the code generation capability of Code LLMs.

Comparison between different trace representations. We then investigate whether there is a trace representation that is relatively better than others. Unfortunately, the results demonstrate that no single trace representation consistently outperforms others. Considering different tasks separately, *SemCoder* is the best choice for program repair tasks, and *SemCoder (GPT4o)* can consistently enhance the reasoning ability of Code LLMs. *Takeaway: SemCoder* and *SemCoder (GPT4o)* are recommended representations used in fine-tuning for program repair and code reasoning tasks.

Fine-Tuning	Task	Program Repair (27.8K samples per each representation)						Code Synthesis	Code Reasoning
		Concise	CodeExecutor	NExT	SemCoder(GPT4o)	SemCoder	w/o trace	32.4K samples	32.4K samples
	Token Size (M)	23.3	23.8	19.6	33.4	32.0	12.5	14.4	27.8
Evaluation	Task	Code Synthesis				Program Repair		Code Reasoning	
		HumanEval	MBPP	LiveCodeBench(easy)	BigcodeBench(full)	huamnevalpack(HE-R)	MBPP-R	CRUXEval-I	CRUXEval-O
	Sample Size	64	378	880	1140/148	164	378	800	800

Table 1: Details of datasets used in our study.

BaseModel	TrainCorpus	Finetune		Code Repair	Code Synthesis			Code Reasoning	
		downstream	trace	MBPP-R	MBPP	BigcodeBench	LiveCodeBench	CRUXEval-I	CRUXEval-O
DeepSeek-Coder	-	✗	✗	17.7	71.9	41.5	40.8	40.0	40.4
	only NL2Code	✓	✗	25.4	72.9	43.7	12.6	60.1	55.4
	w/o trace	✓	✗	39.2	75.9	45.4	35.7	61.9	56.6
	Concise	✓	✓	39.2	74.4	44.3	29.4	61.6	55.0
	CodeExecutor	✓	✓	38.4	77.2	44.6	31.5	60.4	56.1
	NeXT	✓	✓	37.6	76.7	44.0	36.1	61.3	54.2
	SemCoder(GPT4o)	✓	✓	37.0	75.7	45.4	31.5	62.0	58.1
	SemCoder	✓	✓	40.5	76.4	45.7	29.0	59.5	55.4
LLaMA	-	✗	✗	20.1	58.6	31.4	27.3	42.6	36.2
	only NL2Code	✓	✗	24.9	73.7	44.1	18.1	60.1	55.9
	w/o trace	✓	✗	29.1	59.1	31.6	8.4	58.8	54.0
	Concise	✓	✓	27.0	59.4	30.4	14.7	55.8	57.6
	CodeExecutor	✓	✓	24.9	59.4	32.6	9.7	57.0	55.2
	NeXT	✓	✓	29.1	61.4	30.6	16.0	56.9	52.8
	SemCoder(GPT4o)	✓	✓	22.2	59.4	31.4	10.9	58.6	58.0
	SemCoder	✓	✓	29.4	61.9	33.4	14.7	59.9	55.4
Gemma2	-	✗	✗	20.9	63.7	29.8	32.8	49.2	41.5
	only NL2Code	✓	✗	19.8	61.4	26.8	12.6	57.9	55.6
	w/o trace	✓	✗	24.9	58.4	25.1	6.7	57.8	57.5
	Concise	✓	✓	22.8	60.2	28.8	8.0	57.6	57.2
	CodeExecutor	✓	✓	22.8	59.4	27.3	8.8	58.9	58.2
	NeXT	✓	✓	26.2	58.1	26.9	8.8	59.5	55.8
	SemCoder(GPT4o)	✓	✓	24.1	62.9	29.5	8.4	58.9	56.5
	SemCoder	✓	✓	26.2	62.2	27.6	13.0	58.9	56.8

Table 2: Evaluation results for full-parameter fine-tuning with semantic information on three different base models across three downstream tasks (code repair, code synthesis, and code reasoning). The "trace" setting indicates whether the LLM output includes semantic information. "only NL2Code": fine-tuning using only code generation data without code repair data. "w/o trace": fine-tuning with both code generation data (i.e., NL2Code) and code repair data, where the execution trace is not included in the code repair data. We report pass@1 under greedy decoding, following each benchmark's recommended settings. BigCodeBench measured on the full set and LiveCodeBench is on the easy subset. The best scores per model are underlined

5.2 Parameter-Efficient Fine-Tuning

Parameter-efficient fine-tuning (e.g., LoRA) is widely applied for LLMs. In this part, we explore the influence of LoRA on the fine-tuning of Code LLMs considering semantic information.

Figure 3 depicts the performance of fine-tuned Code LLMs, the detailed results can be found in Appendix C. The results indicate that the effectiveness of parameter-efficient fine-tuning is model-dependent. Concretely, fully fine-tuning performs the best for DeepSeek model (in 5 out of 6 cases), but LoRA enhances the model performance of LLaMA and Gemma2 in most cases (11 out of 12 cases). Similarly, LoRA8 and LoRA64 perform inconsistently across different models, and it is hard to justify which strategy is better. Furthermore, training methods highly affect the ability of code LLM trained, the performance gap between different methods can be up to 21.6 (model LLaMA). This reminds us that choosing proper training methods is crucial for Code LLMs.

Considering different trace representations, the results confirm our previous conclusion that trace-based semantic information cannot significantly enhance the performance of Code LLMs through fine-tuning. However, we found that Gemma2-9B is better adapted to traces, achieving competitive results with strategies such as Semcoder (10.34% pass@1) while maintaining repair improvements. Besides, LoRA64 without trace information is best for general code generation, while LoRA64 + repair-focused traces (e.g., Semcoder_GPT4) maximizes repair capabilities.

Takeaway: Parameter-efficient training methods significantly affect the performance of Code LLMs. However, the effectiveness of each method is highly model-dependent. Besides, fine-tuning without semantic information is still the best choice for preparing Code LLMs with better performance when considering these methods.

5.3 Inference Test-Scaling Computation

Table 3 summarizes the results of test-time scaling. It is clear that, compared to open-source LLMs,

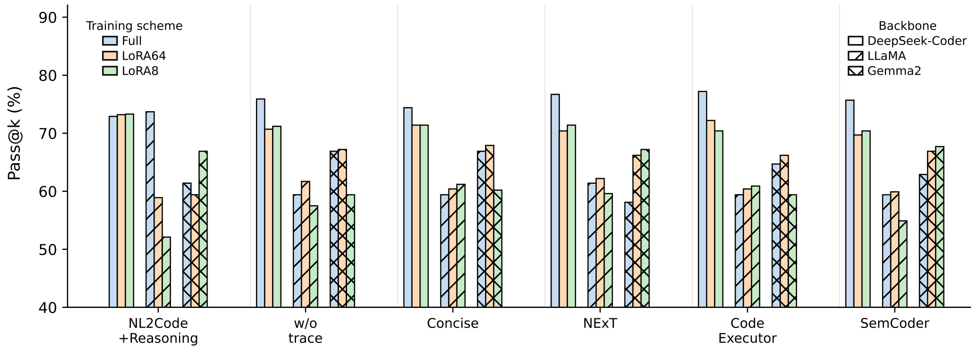

Figure 3: Fine-tuning using different training methods, i.e., Full, LoRA64, and LoRA8.

	Greedy	COT	Sequential Scaling					Parallel Scaling				
			w/o trace	CodeExcutor	Concise	NExT	SemCoder	w/o trace	CodeExcutor	Concise	NExT	SemCoder
GPT-4o-mini	73.08	73.08	98.46	98.46	99.23	99.23	99.23	88.46	80.77	80.77	84.62	80.77
deepseek-chat(V3)	84.62	100.00	100.00	100.00	100.00	100.00	100.00	96.15	96.15	96.15	96.15	92.3
Reasoning Compatible Model												
AIDC-AI/Marco-o1	53.85	50.00	76.92	69.23	76.92	73.08	73.08	61.54	53.85	69.23	61.54	57.69
microsoft/phi-4	53.85	73.08	100.00	96.15	100.00	91.54	100.00	80.77	76.92	84.62	80.77	84.62
Instruction of Foundation Model												
Llama-3.1-8B-Inst	34.62	34.62	67.69	66.92	74.62	74.62	65.38	46.15	42.31	57.69	57.69	57.69
deepseek-coder-6.7b-Inst	42.31	46.15	68.46	61.54	69.23	76.15	69.23	53.85	50.00	57.69	61.54	50.00
Qwen2.5-Coder-7b-Inst	61.54	34.62	83.85	87.69	80.77	90.77	86.92	53.85	61.54	65.38	50.00	53.85

Table 3: Pass@1 accuracy on the **LiveCodeBench** (*easy*) private test set under equal compute budgets. Values are the percentage of prompts whose *final* completion passes *all* private test cases. **Greedy**: one-shot, highest-probability decode. **CoT**: answer plus natural-language rationale. **Sequential Scaling**: 8 parallel samples (T = 0.7) followed by R_{max}=4 self-debugging rounds on public tests, selecting the best candidate. **Parallel Scaling**: 16 candidates ranked by votes from an LLM-as-a-Judge on execution result along with its trace representation. The "*w/o trace*" variants rely only on the initial execution output, whereas trace-based variants leverage execution traces representations during self-debugging or voting. Sequential Revision benefits most from trace-aware signals. Double underlining marks the overall best LiveCodeBench private-set score.

closed-source LLMs perform significantly better at test time.

Impact of test-scaling strategies. First, the results demonstrated that test-scaling consistently enhances the code generation ability of Code LLM. Concretely, in 65 out of 70 cases, test scaling strategies achieved higher Pass@1 scores than Greedy and COT. **Impact of semantic representation.** The results suggest that, similar to SFT, the usefulness of semantic information is blurred. In more than half cases (36 out of 56 cases), integrating semantic information into the input prompt cannot help Code LLM to produce correct code compared to without adding semantic information. However, one semantic representation (*Concise*) stands out, which achieved Pass@1 no worse than *w/o trace* in 11 out of 14 cases.

Takeaway: Similar to fine-tuning, most of the trace-based semantic representations cannot enhance the performance of Code LLMs at test time except for *Concise*.

5.4 Hyperparameter Study

We further explore the impact of hyperparameters on Sequence scaling, which significantly boosts Code LLMs at test time.

We first investigate the impact of model temperature. Figure 5 illustrates the results, where Pass@1 scores fluctuate under different temperatures. One conclusion we can draw is that a small temperature (T=0.2) negatively affects the performance of Code LLMs, higher temperature performs relatively better.

Sequence scaling has two parameters, the iteration number (*rounds*) and the generated samples (*samples*) during each iteration. The results in Figure 5 show that the more samples generated, the higher Pass@1 scores achieved by Code LLMs. However, there is a trade-off between the sample numbers and the performance of Code LLMs. *samples*=8 is the default setting in our framework. The results of *rounds* study can be found in Table 9a, where more rounds lead to better code generation capability of Code LLMs.

6 Related works

Chain-of-Thought (CoT) and Tool-Integrated Reasoning (TIR). Beyond execution traces, recent advances emphasize explicit reasoning steps and tool usage. *Chain-of-Thought (CoT)* (Wei et al.,

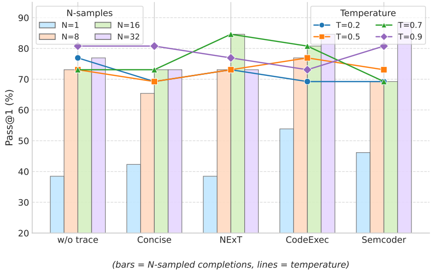

Figure 4: Pass@1 of Qwen-7B on the LCB "easy" split as a function of sampled completions N (bars) and decoding temperature T (lines) across five trace formats. Two trends emerge: (i) increasing N yields substantial gains across all formats; (ii) higher temperatures (T > 0.7) generally outperform lower ones. NExT and CodeExecutor achieve the best results (88.5% at N=32, T=0.9), followed by Semcoder, while the baseline w/o trace consistently underperforms.

2022) enables LLMs to decompose complex problems into intermediate reasoning steps, improving accuracy in tasks like mathematical problem-solving. However, some tasks require computational precision beyond language reasoning. *Tool-Integrated Reasoning (TIR)* (Gou et al., 2023) addresses this by integrating LLMs with external tools (e.g., Python interpreters) for specialized computations, excelling in symbolic computation and high-complexity algorithms. These approaches highlight the trend of augmenting LLMs with runtime observations or external tools, which our work builds upon by systematically evaluating their impact on reasoning and code generation.

LLMs for Software Engineering and Execution Behavior. Code execution behavior encompasses runtime information (e.g., program state, execution paths) and pre-/post-execution details. Recent studies leverage these behaviors to enhance LLM performance. For instance, (Chen et al., 2023) introduced Self-Debugging, where LLMs generate explanations to guide debugging; (Ni et al., 2024) proposed NExT, representing execution behaviors as inline comments for fine-tuning; and (Ding et al., 2024) described runtime behaviors in natural language for LLM training. While prior works typically use a single representation, our study explores multiple execution-based representations and their impact on code generation and reasoning tasks.

LLMs are widely applied in software engineering, including vulnerability detection (Shestov et al., 2024), bug repair (Xia and Zhang, 2023; Wang et al., 2024b, 2025a), and code generation (Hong et al., 2023; Wu et al., 2023; Tang et al., 2024). Evaluation frameworks (e.g., EvalPlus (Liu et al., 2023c)) and datasets (e.g., ClassEval (Du et al., 2024), SWE-bench (Jimenez et al., 2023)) have been developed to benchmark these capabilities. While prior efforts focus on task-specific performance, our work investigates how execution-centric signals, inspired by human debugging, enhance LLMs' proficiency in code generation and reasoning.

Scaling Up Inference-Time Computing Recent advances in inference-time computing have improved the verification of mathematical reasoning in LLMs. (Cobbe et al., 2021) introduced token-level reward models to score individual steps, while (Xiao et al., 2024) refined these with process reward models (PRM) for granular feedback. (Snell et al., 2024) demonstrated that scaling inference-time computing is more cost-effective than retraining models. Building on these, we sample multiple solutions from LLM reasoners and explore verifier training approaches. Our framework, adapted from (Li et al., 2025), systematically evaluates how the semantic information impacts the post-training, one-time inference, and scaling inference test-time with runtime behavior can achieve strong performance in code-related tasks.

7 Conclusion

This paper introduces a generic framework for generating trace-based code semantic information. Based on this framework, we systematically evaluate the usefulness of trace-based code semantic information for fine-tuning and inference of Code LLMs. The results highlight that existing code semantic information does not enhance fine-tuning and test-time scaling of Code LLMs.

This work can serve as the new baseline for the study of leveraging semantic information to enhance Code LLMs. This opens up several avenues for future research. First, it is essential to design new forms of semantic representations that are more aligned with how Code LLMs process and understand code, potentially incorporating higher-level abstractions or contextual cues. Second, future work should explore more effective strategies for integrating semantic information into model training and inference pipelines – such as architectural modifications, specialized pretraining objectives, or more adaptive prompting techniques.

8 Data and Source Code availability

All source code, datasets, and intermediate data for reproduction are available at https://github.com/

tracewise-probing/tracewise_probing.

9 Acknowledgements

This work was partially supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), the CyberSG R&D Cyber Research Programme Office, the Singapore Ministry of Education Academic Research Fund Tier 1 (RG12/23). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore, CyberSG R&D Programme Office as well as MOE.

Limitations

Limited Programming language supported. Currently, our framework only supports Python, other programming languages, such as Java and C++, are not supported. Even though, we believe our findings can provide insights to developers who plan to enhance their Code LLMs via semantic information interaction. Besides, we plan to actively maintain our framework to cover more programming languages.

Limited LLM size. Due to constraints on computational resources, we only conduct experiments on LLMs with around 7B size. Experiments with larger LLMs could be our future work.

Ethical Considerations

Research purpose and societal impact. This project seeks to deepen scientific understanding of how dynamic program semantics influence Code-LLM reasoning, with the ultimate goal of producing safer, more reliable coding assistants. All artefacts—datasets, code, and models—are released solely for research and evaluation; they are **not** intended for autonomous deployment in production settings.

Provenance, licensing, and consent. Source code used for fine-tuning and evaluation is drawn exclusively from repositories under OSI-approved permissive licences (e.g., MIT, Apache-2.0). No proprietary material is incorporated without explicit permission.

Privacy preservation in execution traces. Because runtime logs can inadvertently expose credentials or personally identifiable information (PII), every trace passes a three-stage sanitation pipeline: (i) static pattern-based redaction of common secret/PII formats; (ii) dynamic taint tracking that masks values originating from environment variables, network sockets, or file I/O; and (iii) manual review of a random 2% sample per release. Traces failing any check are discarded.

Safeguards against malicious use. Enhanced semantic reasoning could facilitate the generation of vulnerable or harmful code. We therefore (i) withhold weights fine-tuned on explicitly security-sensitive benchmarks, (ii) deploy server-side filters that block outputs matching exploit-related patterns (shell execution, SQL/command injection, path traversal), and (iii) document residual unsafe generations in an appendix to encourage community development of stronger guards.

Bias and inclusivity in dataset design. Although our empirical study concentrates on mainstream languages, we provide *PySnooper* for Python, and Solidity to spur broader, community-driven extension. We encourage downstream researchers to audit biases that may arise when applying our framework to new ecosystems or developer populations.

Responsive governance. We publish our redaction and inspection pipeline under an open-source licence, provide a dedicated security-contact email for vulnerability disclosures, and commit to removing or revising any resource within 30 days of a substantiated harm report.

In summary, we have instituted licensing checks, consent agreements, privacy filters, controlled release mechanisms, and transparent governance to ensure that the benefits of semantics-aware Code-LLMs are realised responsibly while potential harms are proactively mitigated.

References

2023-3. vllm, a high-throughput and memory-efficient inference and serving engine for llms.

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J Hewett, Mojan Javaheripi, Piero Kauffmann, and 1 others. 2024. Phi-4 technical report. *arXiv preprint arXiv:2412.08905*.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2024. Evaluating large language models with runtime behavior of program execution. *arXiv preprint arXiv:2403.16437*.

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. 2021a. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

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

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Tora: A tool-integrated reasoning agent for mathematical problem solving. *arXiv preprint arXiv:2309.17452*.

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, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, and 1 others. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

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*.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973*.

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

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. Dspy: Compiling declarative language model calls into self-improving pipelines.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E Gonzalez, and Ion Stoica. 2025. S*: Test time scaling for code generation. *arXiv preprint arXiv:2502.14382*.

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

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. 2023c. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023a. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023b. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.

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

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, and 1 others. 2021. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Alexey Shestov, Anton Cheshkov, Rodion Levichev, Ravil Mussabayev, Pavel Zadorozhny, Evgeny Maslov, Chibirev Vadim, and Egor Bulychev. 2024. Finetuning large language models for vulnerability detection. *arXiv preprint arXiv:2401.17010*.

Wenlei Shi and Xing Jin. 2025. Heimdall: test-time scaling on the generative verification. *arXiv preprint arXiv:2504.10337*.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Daniel Tang, Zhenghan Chen, Kisub Kim, Yewei Song, Haoye Tian, Saad Ezzini, Yongfeng Huang, and Jacques Klein Tegawende F Bissyande. 2024. Collaborative agents for software engineering. *arXiv preprint arXiv:2402.02172*.

Jian Wang, Shangqing Liu, Xiaofei Xie, and Yi Li. 2024a. An empirical study to evaluate AIGC detectors on code content. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 844–856.

Jian Wang, Shangqing Liu, Xiaofei Xie, Jing Kai Siow, Kui Liu, and Yi Li. 2024b. RATCHET: Retrieval augmented transformer for program repair. In *Proceedings of the 35th International Symposium on Software Reliability Engineering (ISSRE)*, pages 427–438.

Jian Wang, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jiongchi Yu, Jiaolong Kong, and Yi Li. 2025a. Defects4C: Benchmarking large language model repair capability with C/C++ bugs. In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

Junxiong Wang, Wen-Ding Li, Daniele Paliotta, Daniel Ritter, Alexander M Rush, and Tri Dao. 2025b. M1: Towards scalable test-time compute with mamba reasoning models. *arXiv preprint arXiv:2504.10449*.

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.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro Von Werra, Arjun Guha, and Lingming Zhang. 2024. Selfcodealign: Self-alignment for code generation. *arXiv preprint arXiv:2410.24198*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical program repair in the era of large pre-trained language models. *arXiv preprint arXiv:2210.14179*.

Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*.

Chaojun Xiao, Jie Cai, Weilin Zhao, Guoyang Zeng, Xu Han, Zhiyuan Liu, and Maosong Sun. 2024. Densing law of llms. *arXiv preprint arXiv:2412.04315*.

Fanghua Ye, Mingming Yang, Jianhui Pang, Longyue Wang, Derek F Wong, Emine Yilmaz, Shuming Shi, and Zhaopeng Tu. 2024. Benchmarking llms via uncertainty quantification. *arXiv preprint arXiv:2401.12794*.

Yu Zhao, Huifeng Yin, Bo Zeng, Hao Wang, Tianqi Shi, Chenyang Lyu, Longyue Wang, Weihua Luo, and Kaifu Zhang. 2024. Marco-o1: Towards open reasoning models for open-ended solutions. *Preprint*, arXiv:2411.14405.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Experiment

A.1 Experiment Detail.

Finetune We conducted SFT experiments on three different base models using two distinct configurations. First, we perform full-parameter tuning with DeepSpeed ZeRO-3 using learning rates of 2.0e-5; training batch sizes of 8; bfloat16 precision; a maximum sequence length of 2048; and two epochs. Next, we explore LoRA-based tuning with various LoRA ranks, disabling DeepSpeed while maintaining the same bfloat16 precision, maximum sequence length, and epoch count.

Scaling inference. *Sequential Revisions* uses an external tool, a Python interpreter, to verify each predicted solution. As introduced in Section 3.1, the feedback from tools is appended to the prompt for the next generation iteration. This iterative process continues until a successful solution emerges or the compute budget of $R_{\max} = 8$ rounds is reached.

Parallel Scaling generates N diverse candidate solutions in parallel to increase the likelihood of finding a correct solution. We synthesize test inputs, execute all candidates, and collect their execution outputs and trace representations as scoring prompts. An LLM-as-a-judge then ranks solutions based on these scoring prompts and pre-trained knowledge. This Parallel Scaling complements Sequential Revisions to maximize code generation capabilities without additional training. Specifically, we sample $N = 16$ answers independently from the specified LLM and then select the best answer according to the reward model's final judgment. We use Phi-4 as the reward LLM to score each candidate with a reward function (e.g., static analysis, unit tests, or a learned model, similar to Best-of-N sampling), and retain only the top $\frac{N}{M}$ candidates. We score these again, prune to the top-ranked solutions, and repeat iteratively until all buggy lines are addressed. The result is up to N complete repaired-code solutions, from which we select the best via a final evaluation, always choosing the highest score as judged by the reward model.

Greedy sampling generates only 1 answer independently from the specified LLM by setting the temperature to 0.

Implementation and Environment. We implement all Code LLMs based on Hugging Face APIs, the implementation of the fine-tuning process is modified from the official project of (Hu et al., 2021). We use OpenAI's official APIs to access gpt-3.5-turbo-0125/gpt-4o-mini-2024-07-18/gpt-4o-2024-05-13 models. Finetune experiments have been conducted on eight NVIDIA A100 GPUs using the Distributed Data Parallel (DDP) module. Inference jobs utilize the vLLM (vll, 2023-3), which is a unified library for LLM serving and inference.

B Dataset

B.1 Finetune Dataset Quality Filtering and Decontamination

We follow the (Wei et al., 2024) to conduct Decontamination and Refinement.

Removing Benchmark Data To ensure the integrity of our evaluation process, we rigorously decontaminated the dataset by removing any functions that resembled prompts or solutions from the benchmarks used for evaluation. This step is critical to prevent data leakage and ensure a fair assessment of our method. Specifically, we checked for the presence of substrings from benchmark prompts or solutions within the dataset. Any function containing such substrings was excluded. This process guarantees that the dataset remains unbiased and does not inadvertently include examples that could skew evaluation results.

Docstring Quality Filtering We observed that many Python functions, while containing docstrings, often had poor or misleading documentation. To address this, we employed StarCoder2-15B, a state-of-the-art language model, to perform binary classification on the docstrings. The model was tasked with identifying functions with low-quality or misleading documentation. Functions flagged as having poor docstrings were removed from the dataset. This step ensures that the retained functions are not only functional but also well-documented, enhancing their usability for downstream tasks such as code understanding and generation.

In sum up, the decontamination and refinement process, particularly the removal of benchmark-related data and the filtering of low-quality docstrings, plays a pivotal role in ensuring the quality and reliability of our dataset. By meticulously removing functions that could compromise evaluation fairness and those with inadequate documentation, we have created a robust dataset of 248,934 high-quality Python functions. This dataset is well-suited for a wide range of applications, including code generation, evaluation, and analysis, while maintaining a high standard of integrity and usability.

B.2 Evaluation dataset

Evaluation on Code Generation and Reasoning Tasks In this study, we fine-tune Code LLMs using the refinement dataset described in Section B.3. For the experiment, we deploy the fine-tuned models in two code generation tasks: program repair and code synthesis. We further fine-tune the Code LLMs specifically for reasoning tasks to assess whether their reasoning capabilities are enhanced. We evaluate the performance of the fine-tuned models using open-source test datasets.

Note that to mitigate potential data leakage risks, we adhere to established methods as outlined in (Muennighoff et al., 2023b) by conducting a thorough decontamination process. This ensures that there is no overlap between our fine-tuning dataset and the evaluation datasets utilized.

- *Program Repair.* To evaluate program repair capabilities, we construct datasets of buggy code using benchmarks from HumanEval, MBPP, and CRUXEval, maintaining a consistent sample size of 164, which aligns with HumanEval. For HumanEval, we directly use buggy code from the existing dataset HumanEvalPack (Muennighoff et al., 2023a). For the other two datasets, which contain larger sample sizes, we randomly select 164 samples from each. Following the methodology described in (Ni et al., 2024), we employ GPT-4, GPT-3.5-Turbo, and CodeLlama-34B to generate solutions for each problem. From these, we select one incorrect solution per problem based on test case validation. This process results in collections of 164 buggy code samples for each dataset, denoted as Human-R, MBPP-R, and CRUXEval-R. For the repair evaluation, the prompts provided include the buggy code, the corresponding failed test case, and the execution traces of that test case [2].
- *Code Synthesis.* We evaluate the code synthesis capabilities of the tuned Code LLMs using two widely-used datasets, HumanEval and MBPP, both provided by EvalPlus (Liu et al., 2023c). To ensure consistency in our assessment, we employ the same prompts and pre-processing methods as outlined in (Liu et al., 2023c). Additionally, we differentiate between two sets of test cases from EvalPlus, referred to as *base* and *plus*, in our evaluations.
- *Code Reasoning.* We follow the existing works (Chen et al., 2024; Gu et al., 2024; Wang et al., 2024a) to conduct the reasoning tasks, i.e., input prediction, output prediction, state prediction, and coverage prediction. Input/output prediction indicates fulfilling the corresponding input or output, given a block of code and a partially completed assertion statement as a prompt. State prediction refers to predicting what the next line statement will be after an intermediate statement is executed. Coverage prediction means that after randomly picking a line of code, we ask the LLM to predict whether it will be executed for a given specific test case. For input and output prediction, we directly use the existing datasets (Gu et al., 2024), named as CRUXEval-I and CRUXEval-O. For the evaluation, we follow the same prompts from (Chen et al., 2024; Gu et al., 2024).

B.3 Fine-tune dataset(Refinement Dataset)

We found that there are no datasets that fully support our study, i.e., the repair-based training mode and all types of execution representation that we collected. Hence, we construct a new dataset that covers buggy code, its corresponding patch, test cases, and other semantic information such as execution traces.

Our dataset is constructed using APPs (Hendrycks et al., 2021), a dataset provided by codeparrot for the generation of code at the competition level. This dataset includes essential elements such as basic buggy code, correct code, test cases, and the human refinement trajectories from buggy to correct versions.

The steps to construct the dataset are as follows:

1. *Buggy and Patch Pair Collection*: Each problem in apps includes multiple solutions, both correct and incorrect, provided by various users. A key challenge in extracting (buggy code, patch code) pairs is the difficulty in

[2] Due to space constraints, we have made all prompt templates, including those used for fine-tuning and evaluation, available on our website.

matching incorrect code with corresponding correct solutions due to anonymized author information for privacy. To overcome this, we employ a similarity-based matching approach, as the buggy code and its refined version from the same author typically exhibit significant similarities. Specifically, we employ UniXcoder (Guo et al., 2022) to extract embeddings of both correct and incorrect programs and calculate their embedding similarity using Cosine similarity measurement. We include pairs with a similarity score above 0.8 in our dataset as (B_X, P_Y).

2. *Test Case Extraction*: After collecting the code pairs, we execute both the buggy and patched code using their accompanying test cases from apps. We retain test cases that fail with the buggy code but pass with the patched code, designating them as failing test cases, which are then incorporated into R_X.

3. *Execution Trace Extraction*: We execute both the buggy and patched codes under the failing test cases and use Trace-Tracker, a Python debugging tool, to gather runtime information, including trace coverage and states. We develop converters to translate this runtime information into various trace representations (detailed in Section 3.1). These traces are added to R_X and R_Y, respectively.

4. *Code Reasoning Related Data Extraction*: Leveraging the execution traces, we enrich our dataset with features specifically designed to evaluate various aspects of code reasoning: input prediction, output prediction, state prediction, and coverage prediction. For input and output predictions, we adhere to the methods outlined in (Gu et al., 2024), inserting assert statements to validate the inputs and outputs effectively.

5. *Program Description Extraction*: Following previous work (Chen et al., 2021b), we utilize each contest problem's brief description as a base. We then employ GPT-4o to enrich these descriptions by generating implementation constraints and incorporating test cases.

C Experiment results of Full and PEFT

C.1 Prediction uncertainty analysis.

Finally, we explore the reason why fine-tuning with trace-based semantic information cannot enhance LLM's performance from the perspective of prediction uncertainty. We employ tasks and datasets provided by (Ye et al., 2024) to measure the prediction uncertainty of fine-tuned models. Table 7a presents the uncertainty scores and Table 7b summarizes the correlation between prediction uncertainty and model performance. The correlation result demonstrates that for code generation tasks, there is a clear positive correlation between the uncertainty scores and the model performance. This indicates that the model is more certain about its prediction but has lower code generation capability. *Takeaway:* Semantic information in fine-tuning datasets can truly boost the prediction confidence of Code LLMs, but fail to increase the prediction correctness. How to utilize semantic information to help fine-tune Code LLMs is still an open problem.

D Experiment results of Test-scaling on MBPP-R

Table 8 summarizes the results of test-time scaling. It is clear that compared to open-source LLMs, closed-source LLMs perform significantly better at test time. Comparison between inference with and without trace-based semantic information. First, we can see that, different from the findings from the fine-tuning investigation, inference with trace-based semantic information consistently boosts the performance of Code LLMs. In most cases (98 out of 112), adding trace information enhances Code LLMs with an improvement by up to 10.85, a max improvement in NeXT with GPT-4o. This indicates that semantic information can guide LLMs in generating more correct code. Comparison between different trace representations. Similar to previous findings, no single semantic representation consistently performs better than the others. SemCoder, which performs relatively better during fine-tuning, cannot stand out considering inference only. Concise, a variant of CodeExecutor designed by us performs the best under the instruction version of LLMs. Different from fine-tuning, trace-based semantic information significantly enhances the performance of Code LLMs at test time. Sequential Revisions and Parallel are two optimal search strategies for test-time scaling.

		repair		NL2Code		reasoning		bigcodebench		Livecodebench	
		HE-R/(+)	MBPP-R/(+)	HE/(+)	MBPP/(+)	in_predict	out_predict	full	hard	easy pass@1	overall pass@1
DeepSeek-Coder		26.2(22.0)	17.7(15.6)	49.4(43.9)	71.9(57.6)	40	40.4	41.50	12.20	40.80	17.40
only nl2code	Full	59.1(52.4)	25.4(22.0)	64.6(56.1)	72.9(60.9)	60.1	55.4	43.70	16.90	12.60	4.60
	LoRA64	48.2(42.1)	25.4(22.0)	57.3(48.8)	73.2(59.1)	44.8	47.9	44.70	14.90	44.10	18.50
	LoRA8	44.5(37.2)	22.2(19.3)	52.4(44.5)	73.3(58.1)	42.9	47.2	44.50	12.20	41.20	17.40
concise	Full	45.7(39.0)	39.2(33.1)	61.6(54.3)	74.4(61.9)	61.6	55	44.30	17.60	29.40	12.60
	LoRA64	46.3(41.5)	33.9(28.8)	53.7(47.6)	71.4(59.1)	48.1	48.8	42.50	12.20	40.30	16.70
	LoRA8	42.7(38.4)	35.7(30.4)	54.3(48.2)	71.4(59.1)	54.1	48	45.60	16.20	38.70	16.30
CodeExecutor	Full	38.4(34.8)	38.4(33.6)	60.4(53.7)	77.2(62.9)	60.4	56.1	44.60	20.30	31.50	13.60
	LoRA64	45.1(40.9)	36.0(31.2)	54.3(48.2)	72.2(60.9)	52.5	48.8	45.00	15.50	38.20	17.10
	LoRA8	49.4(43.3)	33.9(29.1)	54.9(47.0)	70.4(57.6)	46.2	47.8	45.20	16.90	39.90	17.10
w/o trace	Full	43.9(39.0)	39.2(34.1)	58.5(51.8)	75.9(61.9)	61.9	56.6	45.40	16.20	35.70	15.60
	LoRA64	49.4(45.7)	35.2(29.4)	53.7(46.3)	70.7(58.6)	54.2	49.1	46.40	18.90	43.70	18.80
	LoRA8	47.0(40.9)	30.4(24.9)	53.0(44.5)	71.2(58.6)	49.4	48.4	46.10	14.90	41.60	17.70
NeXT	Full	39.6(36.6)	37.6(31.7)	61.6(54.9)	76.7(62.4)	61.3	54.2	44.00	16.90	36.10	14.90
	LoRA64	47.6(42.1)	34.7(29.6)	54.9(48.8)	70.4(58.4)	47.4	49.6	45.40	16.20	42.40	17.10
	LoRA8	47.0(41.5)	34.4(28.8)	55.5(47.6)	71.4(57.6)	47.4	47.6	45.00	17.60	40.30	17.00
concise2	Full	43.3(37.2)	38.9(33.1)	59.1(51.8)	76.7(63.2)	60	55.6	43.90	18.90	35.70	15.30
	LoRA64	48.8(43.9)	35.7(30.4)	52.4(46.3)	70.2(59.1)	52.8	49.5	45.40	16.90	38.20	16.30
	LoRA8	48.2(42.7)	32.8(28.6)	53(45.1)	70.9(57.6)	46.4	49	43.60	13.50	41.20	17.50
SemcoderGPT4o	Full	53.7(47.6)	37.0(32.0)	59.1(52.4)	75.7(63.4)	62	58.1	45.40	18.20	31.50	13.70
	LoRA64	51.8(46.3)	34.9(29.9)	55.5(48.8)	69.7(58.9)	53.1	50.7	46.10	18.20	39.10	17.10
	LoRA8	51.2(44.5)	30.4(26.5)	52.4(45.1)	70.4(56.9)	50.5	49.5	47.20	18.90	42.00	17.40
SemcoderGPT4o_y	Full	47.0(40.9)	38.9(32.0)	61.6(55.5)	74.7(62.9)	61.3	57.4	45.50	18.90	42.00	16.70
	LoRA64	53.0(47.6)	32.3(28.6)	54.3(48.8)	72.9(60.4)	52.6	49.2	44.70	16.90	44.50	18.20
	LoRA8	53.7(45.7)	31.7(27.5)	52.4(45.7)	72.7(58.4)	46.1	48.9	46.10	17.60	39.90	17.10
Semcoder	Full	45.7(39.6)	40.5(35.2)	58.5(51.8)	76.4(63.2)	59.5	55.4	45.70	20.90	29.00	12.80
	LoRA64	48.2(43.3)	34.7(30.2)	53.0(46.3)	70.2(58.9)	52.4	50.5	45.90	17.60	42.90	18.10
	LoRA8	48.8(43.3)	32(27.5)	53.7(45.7)	70.7(57.1)	47	49.8	46.10	16.20	41.20	17.50
Semcoder_y	Full	46.3(39.6)	37.6(33.1)	60.4(54.3)	75.4(61.9)	59	57.8	44.10	16.90	39.10	16.40
	LoRA64	48.2(42.7)	32.8(29.1)	56.1(49.4)	72.9(60.2)	54.1	49.8	44.80	19.60	40.80	17.10
	LoRA8	51.2(44.5)	31.7(27.2)	53.0(45.7)	72.2(58.9)	49.6	47.6	45.20	15.50	41.60	17.50

Table 4: A extend version of Table in DeepSeek-Coder(deepseek-6.7B-base) after finetuning with semantic information. The _y token indicates that the rationale r should be generated, rather than fed into the prompt.

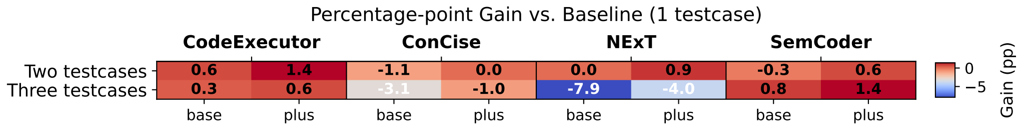

Figure 5: Heat-map of percentage-point gains over the (one-test case) baseline. *Rows* mark the aggregated test-case sizes. *Columns* are grouped by trace representations each with its MBPP-R(base, plus) benchmark. Warm shades (reds) indicate positive gains.

E More Prompt Example

E.1 Prompt templates

We also provide detailed prompts used in our experiments in website. Please refer to our github repository [3]

		repair		NL2Code		reasoning		bigcodebench		Livecodebench	
		HE-R/(+)	MBPP-R/(+)	HE/(+)	MBPP/(+)	in_predict	out_predict	full	hard	easy pass@1	overall pass@1
LLaMA		28.0(26.2)	20.1(18.0)	38.4(32.3)	58.6(49.1)	42.6	36.2	31.40	6.08	27.30	9.50
only nl2code	Full	58.5(52.4)	24.9(22.2)	65.9(56.7)	73.7(61.2)	60.1	55.9	44.10	18.20	18.10	7.40
	LoRA64	47.6(42.1)	25.7(21.7)	44.5(40.9)	58.9(46.4)	54.4	52.9	35.10	10.10	29.80	11.10
	LoRA8	43.9(38.4)	23.8(20.1)	39.6(33.5)	52.1(42.9)	52.8	51.9	33.70	9.50	34.50	11.90
concise	Full	30.5(29.3)	27.0(23.8)	47.0(43.3)	59.4(48.9)	55.8	57.6	30.40	8.80	14.70	5.20
	LoRA64	34.8(31.1)	33.6(29.9)	40.2(35.4)	60.4(51.4)	54.4	51.7	35.20	10.10	21.80	8.00
	LoRA8	39.6(32.3)	31.0(27.2)	36.6(32.3)	61.2(49.9)	53.2	51.9	33.90	7.40	32.80	11.60
CodeExecutor	Full	29.9(29.3)	24.9(22.2)	53.7(50.0)	59.4(47.9)	57	55.2	32.60	9.50	9.70	3.60
	LoRA64	31.1(28.0)	32.8(28.8)	39.6(34.8)	58.9(49.6)	53.9	51.2	33.90	6.80	18.90	6.70
	LoRA8	37.8(30.5)	31.0(27.0)	36.0(32.9)	60.9(49.9)	53.4	52.2	33.30	7.40	34.00	11.60
w/o trace	Full	28.0(24.4)	29.1(26.2)	52.4(49.4)	59.1(47.1)	58.8	54	31.60	8.10	8.40	3.10
	LoRA64	31.7(29.3)	32.0(27.5)	42.7(40.9)	61.7(52.6)	53.5	50.5	34.70	10.80	22.70	8.00
	LoRA8	47.0(40.9)	29.1(25.7)	36.6(32.9)	57.5(47.1)	50.2	49.8	32.50	9.50	31.50	11.20
NeXT	Full	28.7(26.2)	29.1(25.4)	49.4(46.3)	61.4(49.1)	56.9	52.8	30.60	6.80	16.00	5.60
	LoRA64	28.7(25.0)	32.5(28.8)	43.9(39.6)	62.2(53.9)	54	51.1	34.60	7.40	24.80	9.00
	LoRA8	42.1(34.8)	31.5(27.2)	37.2(33.5)	59.6(48.4)	53.4	53.1	31.80	8.80	34.90	12.20
concise2	Full	29.9(27.4)	27.2(23.3)	50.6(47.0)	55.4(46.4)	56.6	55.9	32.37	8.11	10.50	3.80
	LoRA64	30.5(28.0)	32.5(27.8)	44.5(39.0)	63.2(53.6)	54	51.2	35.50	8.10	21.80	7.70
	LoRA8	37.2(31.1)	31.0(27.2)	39.0(35.4)	60.4(49.1)	50.5	51.2	33.20	7.40	31.90	10.90
SemcoderGPT4o	Full	38.4(34.8)	22.2(19.8)	51.8(46.3)	59.4(47.9)	58.6	58	31.40	10.80	10.90	4.10
	LoRA64	37.2(32.9)	26.7(24.1)	42.7(38.4)	59.9(50.1)	54.8	50.4	34.40	8.80	23.10	8.60
	LoRA8	37.8(32.3)	25.7(22.0)	39.6(35.4)	54.9(44.9)	52.8	52.1	33.60	7.40	32.80	11.50
SemcoderGPT4o_y	Full	31.7(28.7)	31.5(28.0)	51.8(48.2)	62.7(50.1)	59	60.2	30.60	7.40	14.70	5.60
	LoRA64	36.6(30.5)	29.4(25.9)	37.8(34.1)	59.1(50.9)	53.5	52.9	34.56	12.84	27.96	9.32
	LoRA8	39.0(33.5)	26.7(23.5)	37.2(32.3)	55.9(44.9)	52.9	51.9	32.20	5.40	33.60	11.80
Semcoder	Full	34.1(29.9)	29.4(24.6)	51.8(48.8)	61.9(48.6)	59.9	55.4	33.40	14.20	14.70	5.20
	LoRA64	28.7(25.6)	31.5(27.2)	42.7(36.6)	59.9(50.4)	54.9	50.4	34.70	6.80	24.80	8.70
	LoRA8	37.2(31.7)	29.1(24.6)	37.2(32.9)	56.1(45.6)	50.6	50.9	32.70	6.80	29.80	10.40
Semcoder_y	Full	29.3(27.4)	28.8(25.1)	51.8(48.2)	58.4(46.4)	59	58.1	33.00	12.20	28.20	10.10
	LoRA64	33.5(29.3)	27.8(24.6)	42.1(36.6)	55.6(44.9)	54.1	53.9	35.61	10.81	26.52	8.75
	LoRA8	37.8(32.0)	24.6(22.2)	42.1(36.6)	55.1(43.4)	51.7	52	33.40	5.40	34.00	11.80

Table 5: A extend version of Table in LLaMA(llama3.1-8B) after finetuning with semantic information.

		repair		NL2Code		reasoning		bigcodebench		Livecodebench	
		HE-R/(+)	MBPP-R/(+)	HE/(+)	MBPP/(+)	in_predict	out_predict	full	hard	easy pass@1	overall pass@1
Gemma2		37.2(33.5)	20.9(19.6)	40.2(34.1)	63.7(51.9)	49.2	41.5	29.80	6.80	32.80	11.40
only nl2code	Full	38.4(35.4)	19.8(16.7)	59.1(54.3)	61.4(50.4)	57.9	55.6	26.80	12.80	12.60	4.90
	LoRA64	50.0(42.1)	24.9(22.8)	54.9(47.0)	69.4(55.1)	57.2	60.1	37.81	11.49	35.84	12.95
	LoRA8	49.4(44.5)	24.6(22.8)	47.0(41.5)	66.9(54.1)	53.9	51.9	37.11	12.16	36.56	13.07
concise	Full	34.1(30.5)	22.8(20.4)	47.0(42.7)	60.2(50.6)	57.6	57.2	28.80	10.80	8.00	3.10
	LoRA64	35.4(31.7)	33.1(28.3)	50.0(45.1)	66.9(52.6)	59.8	51.2	38.86	10.81	26.88	9.09
	LoRA8	37.8(34.1)	33.3(28.8)	48.2(42.1)	67.9(56.1)	55.9	56.1	38.77	8.78	29.03	10.00
CodeExecutor	Full	28.7(25.6)	22.8(19.3)	42.1(37.2)	59.4(48.9)	58.9	58.2	27.30	13.50	8.80	3.10
	LoRA64	37.2(32.9)	32.3(28.0)	48.2(44.5)	64.7(52.1)	59.2	57.5	38.86	11.49	23.66	7.84
	LoRA8	36.6(31.7)	33.9(29.1)	45.1(39.6)	66.2(55.9)	51.5	49.2	37.63	8.11	25.81	9.20
w/o trace	Full	30.5(25.0)	24.9(22.0)	48.2(44.5)	58.4(48.6)	57.8	57.5	25.10	8.80	6.70	2.70
	LoRA64	31.7(28.7)	29.9(26.7)	48.2(42.7)	66.9(54.6)	58.8	56.4	38.77	11.49	23.30	8.30
	LoRA8	35.4(32.9)	31.5(28.0)	45.1(40.2)	67.2(54.9)	51.1	49	38.95	9.46	24.01	8.41
NeXT	Full	31.7(29.3)	26.2(23.3)	48.2(43.9)	58.1(48.4)	59.5	55.8	26.90	8.80	8.80	3.10
	LoRA64	38.4(34.8)	32.0(28.0)	47.6(42.1)	66.2(53.6)	59.5	54.5	38.42	12.16	28.32	9.32
	LoRA8	35.4(32.3)	33.9(29.1)	48.8(42.7)	67.2(55.9)	51.2	51	39.47	12.16	23.30	7.84
concise2	Full	26.2(24.4)	25.1(21.4)	48.2(43.9)	56.9(47.6)	58.9	56.2	29.00	7.40	8.00	2.90
	LoRA64	32.3(29.3)	32.5(28.6)	51.2(45.7)	66.9(53.6)	57.9	57.5	38.77	12.84	29.75	9.55
	LoRA8	40.2(34.8)	31.7(27.8)	44.5(37.8)	67.4(55.1)	46.5	51.7	38.86	6.76	29.03	10.00
SemcoderGPT4o	Full	35.4(31.1)	24.1(21.7)	51.8(48.8)	62.9(50.1)	58.9	56.5	29.50	7.40	8.40	2.90
	LoRA64	45.7(40.9)	29.4(25.4)	47.0(42.1)	66.9(53.6)	57.8	54.4	38.42	14.19	25.81	8.98
	LoRA8	45.1(39.0)	28.6(24.9)	47.6(42.7)	67.7(55.1)	41.1	52.5	38.95	10.14	26.16	9.32
SemcoderGPT4o_y	Full	31.7(30.5)	22.2(18.8)	48.8(44.5)	57.6(46.9)	58.9	52.1	27.63	13.51	9.50	4.90
	LoRA64	32.3(29.9)	29.1(25.9)	47.0(40.2)	68.7(55.9)	59	57.6	39.04	10.81	30.11	10.11
	LoRA8	41.5(37.8)	27.2(24.9)	51.2(44.5)	66.4(53.9)	58.5	51.4	38.95	12.84	23.66	8.30
Semcoder	Full	31.1(27.4)	26.2(22.2)	53(50.0)	62.2(53.1)	58.9	56.8	27.60	13.50	13.00	4.80
	LoRA64	32.9(31.1)	31.0(27.2)	48.8(44.5)	66.4(53.6)	59.2	59.9	39.74	15.54	24.01	7.95
	LoRA8	39.6(35.4)	29.9(26.2)	43.9(38.4)	66.2(55.6)	56.9	55.4	39.74	10.14	21.86	7.61
Semcoder_y	Full	26.8(24.4)	24.3(20.9)	49.4(45.7)	59.4(48.9)	58.5	52.1	23.25	5.41	17.00	5.80
	LoRA64	38.4(35.4)	24.9(22.8)	48.8(42.7)	67.4(54.9)	59.2	59.6	39.12	14.19	30.47	10.34
	LoRA8	40.2(33.5)	24.6(22.2)	43.3(37.2)	67.9(56.1)	51.1	48.5	38.77	13.51	27.96	10.00

Table 6: A extend version of Table in Gemma2(gemma2-9b) after finetuning with semantic information.

(a) Prediction uncertainty of LLMs.

	Van.	NL2C	w/o T	Conc.	CodeEx.	NExT	Sem (GPT4)	Sem
LLaMA	1.90	3.66	2.43	2.98	2.95	2.74	2.44	2.32
DeepSeek-Coder	3.25	3.74	3.60	3.47	3.74	3.49	3.62	3.47
Gemma2	2.99	3.05	3.28	3.01	2.92	2.92	3.37	3.65

(b) Spearman rank correlation between prediction uncertainty and model performance.

	MBPP-R	MBPP	BigCode	LiveCB	CRUX	CRUX-O
Coef.	0.356	0.759	0.440	0.357	0.598	0.228
p**-val**	0.088	1.7e-5	0.031	0.087	0.002	0.284

	Sequential Revisions						BoN	BeamSearch
	w/o trace(greedy)	Concise	CodeExecutor	NExT	SemCoder	SemCoder(GPT4o)		
Close source Model								
GPT-4o	50.79(44.71)	60.05(50.79)	59.79(50.79)	61.64(50.53)	58.73(50.0)	59.41(51.10)	54.46(42.23)	61.29(51.71)
deepseek-chat(V3)	53.17(47.88)	61.11(54.23)	60.85(52.38)	61.11(53.7)	61.11(52.4)	61.31(52.68)	56.37(57.45)	63.17(53.88)
Reasoning Compatible Model								
Marcon-o1	25.40(22.20)	27.00(23.00)	27.00(23.30)	24.10(20.90)	28.80(24.90)	29.10(25.10)	26.30(23.10)	29.40(25.10)
phi-4	39.15(33.86)	43.65(38.10)	45.5(39.15)	42.86(38.10)	44.71(39.68)	44.71(40.12)	41.23(36.45)	45.15(41.86)
Instruction version of Foundation Model								
CodeLlama-7b-Instruct-hf	19.58(18.25)	20.11(18.52)	19.84(19.05)	19.58(17.72)	18.52(16.93)	19.12(17.00)	19.13(17.43)	19.58(18.25)
Llama-3.1-8B-Instruct	28.84(27.51)	36.51(32.28)	32.01(29.89)	32.01(28.57)	33.86(31.48)	34.21(32.14)	29.21(28.43)	33.54(31.42)
deepseek-coder-6.7b-instruct	24.87(23.54)	23.81(22.49)	25.66(23.54)	23.54(22.22)	30.16(27.51)	30.56(28.31)	25.87(24.34)	30.87(28.14)

Table 8: `Pass@1`Comparing compute-optimal approaches on the code repair benchmark MBPP-R at test time, the numbers outside and inside parenthesis "()" indicate the base and plus versions of EvalPlus, respectively. w/o trace (greedy) only interacts with the LLM via its initial (potentially buggy) code. In contrast, other sequential revision methods benefit from trace-based semantic information. The best results of EvalPlus' base highlights with underline

Round	1	2	3	4	5
Pass@1	61.54	84.62	84.62	84.62	88.46
SyntaxError	50.96	9.13	6.25	3.85	4.33
ExecuteFail	34.62	34.62	33.65	34.62	34.62
TestcaseFail	7.21	24.52	25.00	25.96	25.00

(a) Pass-rate improvement and error-type distribution over five self-debugging rounds. The overall pass rate climbs from 61.54 % in Round 1 to 88.46 % by Round 5, while syntax errors drop sharply and an increasing share of examples transitions into the *testcase-pass* category. Percentages are shown for each round; blank cells indicate zero occurrences..