TRANSFORMING LANGUAGE MODELS INTO PROGRAM INTERPRETERS VIA EXECUTION TRACE CHAIN OF THOUGHT

Anonymous authors
Paper under double-blind review

ABSTRACT

Code execution reasoning (CER), the ability to predict code execution on a given input, has emerged as an important aspect of language models' (LMs) coding capabilities. However, many open-source small- to medium-sized LMs continue to perform poorly on simple code snippets, and effective methodologies to enhance CER capability have not yet been established. In this context, we first highlight the limitations of LMs in basic operations in CER. Through our custom tests, including a test that measures the understanding of individual grammar rules, we indicate that code understanding in natural language does not imply actual procedural understanding of code, and that it is necessary to accumulate reasoning steps at a granularity finer than a line in a structured manner. Motivated by these insights, we investigate *ET-CoT* (Execution Trace Chain of Thought), a method in which execution traces are generated with our custom code interpreter *PyTracify* and used as chain-of-thought rationales, in order to transform 8B-class LMs to code interpreters specialized for CER. After fine-tuning with 127k examples, we demonstrate the effectiveness of ET-CoT, improving Qwen2.5-7B-Instruct to 70.0% on CruxEval-O and to 88.3% on LiveCodeBench (execution), thereby setting new baselines for the class.

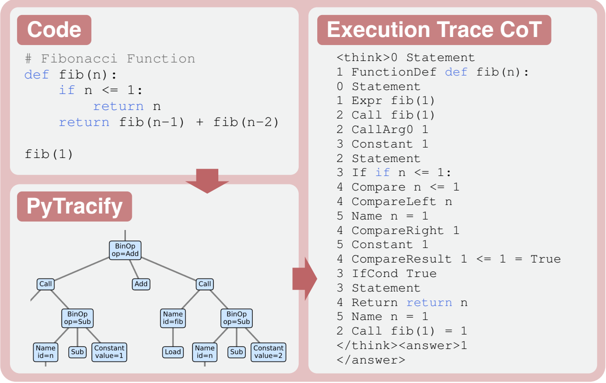
1 INTRODUCTION

With growing expectations for language models (LMs) to handle the entire cycle of code generation, debugging, and optimization autonomously (Islam et al., 2024; Novikov et al., 2025), the coding-related capabilities required of them now extend beyond generation alone (Hou et al., 2024).

Code execution reasoning (CER) (Austin et al., 2021; Nye et al., 2021), the prediction of how a piece of code actually operates on specific input variables, constitutes one such capability. Such procedural understanding of code is a natural competence of skilled human programmers, and it is critical for debugging and repairing errors in generated code (Gu et al., 2024a). Therefore, CER tasks have been incorporated into the evaluation suites for state-of-the-art general-purpose (Yang et al., 2025) and code-specific (Hui et al., 2024b) LMs.

Despite its importance, CER remains challenging for many open-source models (Ding et al. (2024a); Table 2 (a)). Also, the bottlenecks and effective methodologies for this task remain underexplored (La Malfa et al., 2024), because datasets are often black boxes despite the availability of strong open-source LMs (Groeneveld et al., 2024), and because many prior works regard CER as one of the coding tasks rather than focusing on it. Existing approaches typically fine-tune LMs on natural language datasets containing fine-grained explanations of code execution (Ni et al., 2024; Ding et al., 2024a; Li et al., 2025). However, Wang et al. (2025) reported that they showed small improvement over direct-output fine-tuning, questioning if true semantic understanding is achieved.

In principle, however, CER should be solved deterministically with a sufficiently detailed chain of thought (CoT). Indeed, theoretical results suggest that CoT reasoning expands the class of problems that Transformers can compute (Li et al., 2024; Merrill & Sabharwal, 2024). This naturally raises the question of why many current LMs fail to do so, and whether systematic methods can be developed to endow them with the capability to solve CER deterministically with CoT.



```python
# Fibonacci Function
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

fib(1)
```

```
<think>0 Statement
1 FunctionDef def fib(n):
0 Statement
1 Expr fib(1)
2 Call fib(1)
2 CallArg0 1
3 Constant 1
2 Statement
3 If if n <= 1:
4 Compare n <= 1
4 CompareLeft n
5 Name n = 1
4 CompareRight 1
5 Constant 1
4 CompareResult 1 <= 1 = True
3 IfCond True
3 Statement
4 Return return n
5 Name n = 1
2 Call fib(1) = 1
</think><answer>1
</answer>
```

Figure 1: We demonstrate that execution traces can turn small LMs into strong interpreters specialized for CER.

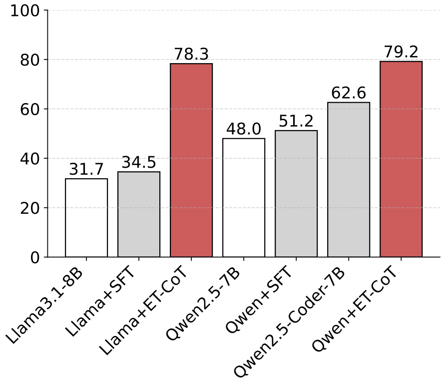


Figure 2: Average score of CruxEval and LiveCodeBench (execution).

In this paper, we focus on solving CER with small-sized ($\leq$8B) LMs. We begin by introducing dedicated tests focusing on basic components of CER, which suggest the necessity of step-by-step reasoning at a granularity finer than a line in a structured manner. This motivates the use of execution traces and points to the intriguing possibility of transforming small-sized LMs into code interpreters *specialized* for CER. To this end, we investigate *ET-CoT* (Execution Trace Chain of Thought), an approach for utilizing execution traces as CoT rationales. The results show the effectiveness of this approach, establishing new baselines for small-sized LMs (Figure 2).

The contributions of this work are summarized as follows:

- To evaluate the CER capability of small-sized LMs, we design two tests targeting fundamental components of CER (Section 3). The first test assesses line-level grammatical understanding employed in CER benchmark (Section 3.1), while the second test evaluates the iterative application of simple procedures on a per-step basis (Section 3.2). The findings demonstrate not just the limitations of moderate-sized LMs, but also offer new perspectives on the difficulty of CER. For example, natural language explanation ability does not guarantee actual procedural understanding, substantial reasoning steps are necessary for a single line, and access to procedural understanding in the model is unstable even when present.
- We collect and create Python snippets to address shortcomings of LMs in CER, develop our custom Python interpreter *PyTracify*, construct synthetic CoT rationales of execution traces using it, and fine-tune various 8B-class LMs (Section 4). In particular, with 127k samples we bring them to a level at which they can function as code interpreters specialized for CER, while most studies targeted multiple coding tasks simultaneously (Ding et al., 2024a; Li et al., 2025; Li & Wang, 2025).
- As a result, we establish new baselines for 8B-class models (Section 5.1), raising the performance of Qwen2.5-7B-Instruct (Qwen team, 2025) to 70.0% on CruxEval (Gu et al., 2024b) and 88.3% on LiveCodeBench (execution) (Jain et al., 2024), surpassing Qwen2.5-Coder-7B-Instruct (Hui et al., 2024a). Furthermore, we observe that ET-CoT can mitigate the earlier CER bottlenecks (Section 5.2).

2 RELATED WORKS

Benchmark and challenges of CER. Apart from general-purpose datasets such as MBPP (Austin et al., 2021), datasets dedicated to CER have only recently appeared. LiveCodeBench (execution) (Jain et al., 2024) collects competitive programming problems, while CruxEval (output prediction) (Gu et al., 2024b) is a synthetic dataset generated with Code Llama and recently extended to multi-languages (Xu et al., 2024). Both contain code snippets of about ten lines with input–output pairs. As for the challenges of CER, prior work has reported difficulties in handling snippets with multiple operators and control flow, as well as error accumulation with increased critical path (Chen et al., 2024; La Malfa et al., 2024; Liu et al., 2025; Liu & Jabbarvand, 2025). However, those discussions mainly consider problems where even commercial models fail (though sometimes presented as simple

(Gu et al., 2024b)), and the analyses of 8B-class models have often been limited to observing that they generally fail on these problems. Finally, we note that CER performance is only weakly correlated with code generation ability (Austin et al., 2021; Gu et al., 2024a; Luo et al., 2023; Wei et al., 2024). Consequently, recent technical reports report CER performance alongside generation metrics (Hui et al., 2024b; Yang et al., 2025), highlighting the importance of CER as an independent task.

Fine-tuning to improve the CER ability. Fine-tuning with direct code, input, and output pairs was attempted in Austin et al. (2021); Gu et al. (2024b), yielding only limited improvements. To further improve CER capability, recent work has attempted to incorporate intermediate reasoning steps. In natural language, NExT (Ni et al., 2024) fine-tuned LMs on execution-aware rationales, while SemCoder (Ding et al., 2024a) enhanced the semantic understanding of code with step-wise explanations. Li et al. (2025) scaled this approach by developing an automatic pipeline to generate such CoT data. On the other hand, interpreter-style execution trace was utilized by Scratchpad (Nye et al., 2021) (but they used the same dataset for both training and evaluation). Such execution trace data was also used by CodeExecutor (Liu et al., 2023) and Ding et al. (2024b) in pretraining. However, Wang et al. (2025) reproduced Scratchpad, CodeExecutor, NeXT, and SemCoder and suggested that intermediate reasoning steps may not more effective than SFT without traces to enhance semantic understanding of the code. This might be because many previous studies addressed multiple tasks simultaneously, which could limit the effect of step-wise simulation data.

In this context, we focus on CER capability and train LMs as specialized interpreters, demonstrating that execution trace itself is sufficient to set a new baseline for CER. Finally, we remark that there is an independent work that also uses execution trace (Armengol-Estapé et al., 2025) (see Appendix A).

Theoretical backgrounds. Transformers' ability to act as interpreters is suggested by theoretical results establishing their Turing completeness (Pérez et al., 2019; Pérez et al., 2021). In particular, by repeatedly performing generation and computation as in CoT, Transformers can solve problems requiring serial computation (Xu & Sato, 2025; Schuurmans et al., 2024; Bhattamishra et al., 2020; Merrill & Sabharwal, 2024). In this spirit, Giannou et al. (2023) constructed a 13-layer looped Transformer that executes general programs. However, without CoT of sufficient length, their expressivity collapses to low-level circuit classes (Merrill & Sabharwal, 2023). We also remark that Zhai et al. (2024) proved that Transformers can efficiently process compiler tasks such as AST construction, symbol resolution, and type analysis. Overall, these theoretical results motivate approaches to code reasoning with CoT, particularly those that mechanically simulate code execution.

3 WHY IS CODE EXECUTION REASONING DIFFICULT FOR LMS?

Our primary goal is to enhance the code execution reasoning ability of small-sized LMs ($\sim$8B). However, most existing datasets for code execution reasoning are designed to challenge advanced commercial LLMs by combining multiple operations (Ma et al., 2023; Gu et al., 2024b; Jain et al., 2024). Such complexity may obscure the specific issues faced by small-sized LMs. In this section, we introduce our own tests and highlight two overlooked failure modes in this context: (i) limitations of basic syntax understanding and (ii) initial instability in iterative operations.

3.1 LIMITATIONS OF BASIC SYNTAX UNDERSTANDING

Line-by-line syntax understanding is the minimal unit of code execution reasoning ability. We made original 800 examples to measure this, based on CruxEval (Gu et al., 2024b). Specifically, from each original snippet of up to 13 lines, we selected the most central line, redefine its operation as a function, and prepared corresponding inputs and outputs. An example is shown in Figure 3 (a case mispredicted by GPT-4.1), with pipeline details provided in Appendix G.1.

Function:

```
def f(o):
    return o[-2::-1]
```

Input:

```
o = 'bab'
```

Figure 3: Sample problem.

We tested nine models on this dataset (Figure 4). Red denotes predictions with answer-only output, while blue denotes predictions with CoT reasoning. Green indicates the correctness of natural language function descriptions. Concretely, each model was asked to describe the function's operation in natural language, and GPT-4.1 then applied this description to the inputs and checked whether the output matched. See Appendix G.2 for more details.

From the results, we observe the following: **(i) Even the simulation of a single operation requires substantive CoT**—for many models, CoT improves accuracy by more than 10% over direct prediction. **(ii) Most 8B-class models have limitations in basic syntax understanding even with CoT**—except for Qwen3-8B, their accuracy stays below 75%. **(iii) Declarative understanding in natural language does not imply procedural understanding**—models can provide accurate natural language descriptions of operations, sufficient to derive the correct output, but their ability to apply these operations to inputs substantially diverges from their declarative understanding.

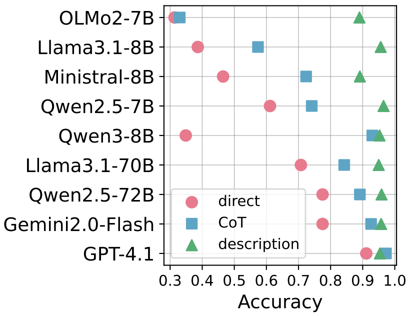


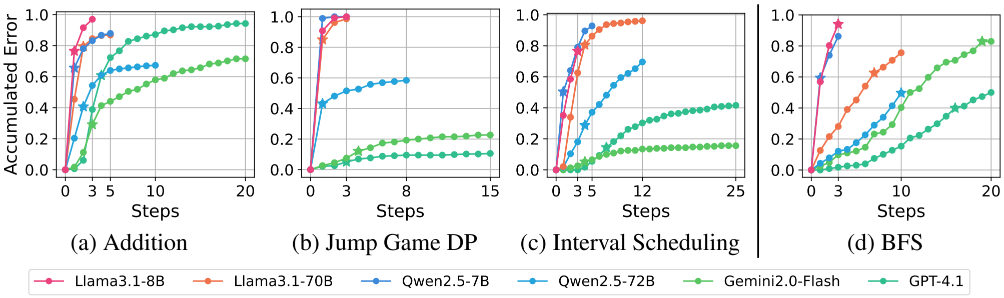
Figure 4: Accuracy of execution simulation for single-syntax functions.

3.2 EVALUATION OF STEP-WISE FAILURE RATES IN ITERATIVE OPERATIONS

Another basic component of CER is the repeated application of operations. Focusing on this, simulating example algorithms with controllable complexity is a common means of evaluation (Liu et al., 2025; Chen et al., 2024; Liu & Jabbarvand, 2025), also seen in natural language reasoning (Shojaee et al., 2025). A common observation is that errors accumulate with repeated steps, eventually leading models to fail in reasoning beyond a certain level of complexity (La Malfa et al., 2024).

We prepared four example programs to evaluate the repetition capability of small-sized LMs. For this purpose, (i) we designed the code so that each step involved simple operations. Furthermore, (ii) we inserted `print` statements within the code and required the models to predict these values, thereby enforcing step-wise reasoning. The four algorithms we selected are: (a) **digit-wise addition**, (b) **dynamic programming for the jump game** (where one moves from the start of the sequence to the end by steps of size one or two, minimizing the total sum of differences along the path), (c) **interval scheduling** of sorted jobs, and (d) **breadth-first search**. See Appendix H for details.

The results are shown in Figure 5. We indicate with a star the maximum step-wise error rate ((failure at step t)/(success up to step $t-1$)). First, (d) BFS exhibits the familiar failure due to accumulated errors, and 8B-class LMs generally perform poorly. However, interestingly, (a), (b), and (c) exhibit a behavior where errors concentrate in the initial steps, followed by stable reasoning in later steps. Extensive results in Appendix H shows that this is not cherry picking. This tendency, observed across both open and commercial models, implies a new failure mode that **even when accurate procedural understanding exists within the models their utilization of it can be unstable**.



(a) Addition (b) Jump Game DP (c) Interval Scheduling (d) BFS

Figure 5: Step-wise cumulative error rate for iterative code simulations.

4 TRAINING WITH EXECUTION TRACE COT

4.1 UTILIZING EXECUTION TRACE TO OVERCOME THE DIFFICULTIES OF CER

The preceding discussion implies that strengthening CER in 8B-class LMs requires a stable and systematic method for accumulating sequential reasoning steps, at a granularity finer than a single line, that align with actual program execution. To this end, we investigate training models on execution traces with the aim of enabling them to function as interpreters. We expect the use of execution traces enables the model to solidify fundamental syntax and control-flow reasoning, generate an appropriate amount of CoT for each instance, and perform CER in an execution-aligned manner. We refer to this approach as *Execution-Trace Chain of Thought* (ET-CoT).

In order to efficiently generalize small-sized LMs as interpreters within the scope of fine-tuning, we carefully select data and define the trace format. Below, we construct a high-quality and non-duplicated code dataset with custom problems, and our original Python interpreter *PyTracify*.

4.2 ET-COT DATASET CONSTRUCTION

We started with creating a high-quality dataset of code, input, and output pairs. While some prior work generates multiple input–output pairs from the same code (Li et al., 2025), we prioritized data diversity to encourage generalization, and therefore constructed a dataset without code duplication.

Specifically, our dataset comprises five sources: **AtCoder** and **LeetCode** subsets from the **Nan-Do** dataset (Nan-Do, 2023) as competitive programming problems, **APPS** (Hendrycks et al., 2021) and **MBPP** (Austin et al., 2021) as more general code generation problems, **PyX** as a mixed dataset with guaranteed quality from a leading prior work of Ding et al. (2024a), and finally, **Custom Dataset** described in Section 4.3.

To ensure that programs are correctly executable, we imposed a 5-second execution limit and retained only those that successfully passed. Furthermore, we removed the top 20% of samples to filter out excessively long traces, based on the trace length defined later by PyTracify.

Table 1: Distribution of the training dataset.

Dataset Source	**#Samples**	**Pct. (%)**
Nan-Do	50,426	39.6
AtCoder contests	*(33,290)*	
LeetCode contests	*(17,136)*	
APPS	25,908	20.3
Custom Dataset	38,879	30.5
String Functions	*(12,000)*	
Tokenizer Vocabulary	*(26,879)*	
PyX	10,958	8.6
MBPP	1,242	1.0
Total	**127,413**	**100.0**

Because our training data included the LeetCode subset of Nan-Do but our evaluation benchmark LiveCodeBench is also based on it, we decontaminated the overlapping problems. We adapted the decontamination script from Open-R1 (Face, 2025) with 8-gram matching and removed all problems corresponding to LiveCodeBench's evaluation set, following Sections 3.3 and A.3 of Jain et al. (2024). After these procedures, we finally obtained 127,413 samples of code, input, and output, and the distribution is shown in Table 1.

4.3 CUSTOM DATASET

As discussed in Section 3.1, 8B-class models exhibit limitations in basic syntax understanding. Moreover, LMs process text token by token and often struggle with strings operations and position identification within strings and lists. CER benchmarks contain such problems, but they are relatively scarce in code generation datasets which we mainly depend in the previous section. To address these issues, we developed customized problems as follows:

String Functions Dataset. To enhance capability of string manipulations, we created a dataset focusing on eight string functions: `slicing`, `replace`, `rpartition`, `find`, `join`, `len`, `removeprefix`, and `rstrip`. These are simple operations for both humans and computers but are known to surprisingly mislead LMs (Gu et al., 2024b). For each function, we generated 1,500 samples by applying it to randomly generated strings of length 3–20 characters, resulting in a total of $1{,}500 \times 8 = 12{,}000$ samples. An example is provided in Appendix D.

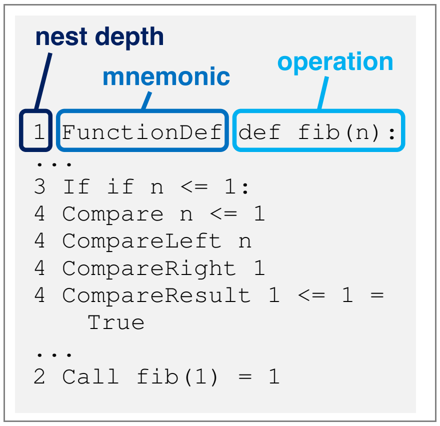
Tokenizer Vocabulary Length Dataset. Another cause of the limitations of string manipulation ability would be the fact that LMs do not correctly recognize the concept of character length due to tokenization. To give models correct understanding of string length, using Llama3.1-8B-Instruct (Dubey et al., 2024b) as an example, we identified vocabulary items for which the model failed to predict the correct token length. From this, we collected 26,879 samples.

Section 5.3.2 reports the performance impact of removing these custom datasets, highlighting their contribution to improving CER accuracy.

4.4 GENERATING EXECUTION TRACES WITH PYTRACIFY

We convert these problems into execution traces using our custom Python code interpreter, *PyTracify*. When designing the trace format of PyTracify, we configured it to output variable updates and control flow at the most basic level. Each line in the trace is represented as a triplet of **nest depth**, **mnemonic**, and **operation**. **Nest depth** encodes the depth of current call or loop nesting so the model can easily track recursion and control flow. **Mnemonic** tell the name of the operations, which follows those used in Python AST node. Finally, **operation** records either the code fragment to be evaluated or the resulting action and value. Figure 6 shows the example of the trace (excerpted).

In addition to unrolling all loops, we decompose each syntax to promote the model's understanding at the line level. For example, when comparing two values in a conditional branch, we first indicate the start of the comparison with `Compare`, then check the left and right values with `CompareLeft` and `CompareRight`, and finally output the result with `CompareResult`. We apply such thorough breakdown to other operations such as evaluation of the while loop condition and binary operations. Moreover, operations such as `len`, whose complexity increases with input length, are rewritten into for-loops for evaluation, even though they are single-line operations. Such designs are motivated by the insight from Section 3 that solving CER requires step-by-step reasoning at a granularity finer than a single line of code. Although this level of detail might seem unnecessary at first glance, Section 5.3.1 quantifies the impact of removing these details on model accuracy.



```
1 FunctionDef def fib(n):
...
3 If if n <= 1:
4 Compare n <= 1
4 CompareLeft n
4 CompareRight 1
4 CompareResult 1 <= 1 =
   True
...
2 Call fib(1) = 1
```

Figure 6: Example of the PyTracify trace.

Regarding the implementation of PyTracify, PyTracify first parses source code with the `ast` module, and then evaluates statements and expressions recursively while maintaining explicit stack frames. The rule for evaluation largely follows CPython. However, given that the code under consideration is relatively simple, there are several deviations such as variable scope. We refer readers to Appendix B for details of syntactic differences between CPython and PyTracify.

Because the whole pipeline executes programs without using any LLM, generation of execution traces requires minimal computational cost. In our case, by launching 100 parallel PyTracify processes, we generated the entire ET-CoT dataset of 127,413 samples in just 7 mins.

4.5 IMPLICATIONS FROM THEORY OF COMPUTATION

Our approach generates execution traces that grow with the input, which may appear inefficient at first glance. However, we emphasize that even for simple code, such traces cannot always be compressed, by stating a theoretical result. As seen in Section 2, theoretical analyses have shown that the presence and length of CoT strongly influence the program-execution capabilities of autoregressive transformer models. Specifically, L and NL denote the classes of problems solvable by deterministic and nondeterministic Turing machines, respectively, using logarithmic space in the input size n. A canonical NL-complete problem (one of the hardest problems in the class NL) is reachability in directed graphs (Sipser, 1996). For such a familiar class of problems, the following property can be established.

Proposition 1. *Assuming* $\mathsf{L} \neq \mathsf{NL}$*, solving an* NL*-hard problem requires a number of CoT steps that scales with the input size* n*.*

Given that it is generally believed that $\mathsf{L} \neq \mathsf{NL}$, this proposition suggests that an LM capable of consistently executing a reasonably wide range of programs must be able to generate CoT traces whose length scales with the required computational steps depending on each problem. See Appendix F for the formal presentation and its proof, which is a straightforward consequence from Theorem 4 of Merrill & Sabharwal (2024).

Table 2: Performance on code-execution benchmarks (all reported with pass@1). ET-CoT consistently improves CER ability across model–dataset pairs.

(a) ET-CoT compared to baseline, SFT, GRPO.

Model	**Size**	**CruxEval-O**	**LCB-Exec**
Llama3.1-8B-Inst (Baseline)	8B	28.0	35.4
Llama3.1-8B-Inst + SFT	8B	36.1 (+8.1)	32.9 (-2.5)
Llama3.1-8B-Inst + GRPO	8B	28.2 (+0.2)	30.3 (-5.1)
Llama3.1-8B-Inst + ET-CoT	8B	**67.8** (+39.8)	**88.5** (+53.1)
Qwen2.5-7B-Inst (Baseline)	7B	45.0	50.9
Qwen2.5-Coder-7B-Inst	7B	65.9 (+20.9)	59.3 (+8.4)
Qwen2.5-7B-Inst + SFT	7B	44.8 (-0.2)	57.6 (+6.7)
Qwen2.5-7B-Inst + GRPO	7B	43.0 (-2.0)	54.2 (+3.3)
Qwen2.5-7B-Inst + ET-CoT	7B	**70.0** (+25.0)	**88.3** (+37.4)
Qwen3-8B (Baseline)	8B	51.3	88.9
Qwen3-8B + SFT	8B	51.4 (0.1)	53.4 (-35.5)
Qwen3-8B + ET-CoT	8B	**73.9** (+22.6)	**91.2** (+2.3)
OLMo2-7B-Inst (Baseline)	7B	28.6	9.1
OLMo2-7B-Inst + SFT	7B	27.5 (-1.1)	30.5 (+21.4)
OLMo2-7B-Inst + ET-CoT	7B	**54.3** (+25.7)	**75.4** (+66.3)

(b) Comparison with other baselines

Model	**Size**	**CruxEval-O**	**LCB-Exec**
StarCoder2 (Lozhkov et al., 2024)	15B	46.2	33.6
StarCoder2-Inst (Lozhkov et al., 2024)	15B	50.9	29.6
CodeLlama-Python (Rozière et al., 2024)	13B	36.0	23.2
CodeLlama-Inst (Rozière et al., 2024)	13B	41.2	25.7
CodeLlama-Python (Rozière et al., 2024)	7B	34.0	23.0
CodeLlama-Inst (Rozière et al., 2024)	7B	36.8	30.7
StarCoder2 (Lozhkov et al., 2024)	7B	34.5	26.3
Magicoder-CL (Wei et al., 2024)	7B	35.5	28.6
Magicoder-S-CL (Wei et al., 2024)	7B	35.8	30.0
DeepSeekCoder (Guo et al., 2024)	6.7B	41.2	36.1
DeepSeekCoder-Inst (Guo et al., 2024)	6.7B	43.2	34.0
Magicoder-DS (Wei et al., 2024)	6.7B	41.9	38.8
Magicoder-S-DS (Wei et al., 2024)	6.7B	43.5	38.4
SemCoder (Ding et al., 2024a)	6.7B	**65.1**	59.7
SemCoder-S (Ding et al., 2024a)	6.7B	63.9	**61.2**

5 EXPERIMENTAL RESULTS

In this section, we present the fine-tuning results with ET-CoT and compare the performance with other training methods and models. For this purpose, we selected four representative 8B-class models: Llama3.1-8B-Instruct, Qwen2.5-7B-Instruct (Qwen team, 2025), Qwen3-8B (Yang et al., 2025), and OLMo2-7B-Instruct (OLMo 2 team, 2025). All models were trained on the 127k ET-CoT dataset for 4 epochs using AdamW (Loshchilov & Hutter, 2019) as an optimizer (β_1=0.9, β_2=0.95, ϵ=1e-8) and cosine decay learning rate (2e-5 $\rightarrow$ 4e-6). The batch size was 64 and context length was 8192 tokens, except that we utilized 4096 for OLMo2-7B-Instruct due to the inherent limit on the context length. The prompt format we used is described in Appendix C.

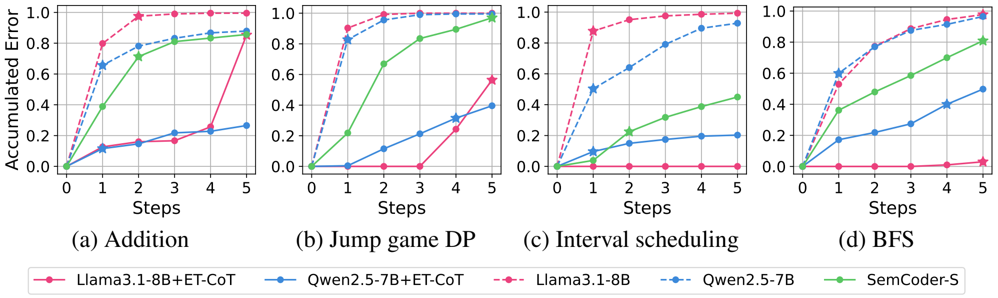
5.1 PERFORMANCE ON CODE-EXECUTION BENCHMARKS

Following SemCoder (Ding et al., 2024a), we chose CruxEval-O (Gu et al., 2024b) and LiveCodeBench (execution, LCB-Exec) (Jain et al., 2024) as CER benchmarks. We report the ET-CoT results in Table 2a, as well as those of the original models, direct-output fine-tuning (denoted as SFT) and Group Relative Policy Optimization (GRPO) (Shao et al., 2024). SFT and GRPO were trained on pairs of code and answer of the ET-CoT dataset, eliminating execution traces. For SFT, the models were trained on all 127,413 examples of ET-CoT, whereas for GRPO we report the results of the best checkpoint within 2,000 steps, as GRPO required longer training time even within this range. Table 2b shows the results of other methods using different base models.

Effectiveness of ET-CoT. From Table 2a, we observe that ET-CoT substantially improved over baselines across all pairs of the models and benchmarks. For example, ET-CoT gains $+39.8$ on CruxEval-O and $+53.1$ on LCB-Exec versus the baseline for Llama3.1-8B-Instruct, which cannot be achieved by SFT/GRPO. Moreover, for Qwen2.5-7B-Instruct, ET-CoT achieved 70.0% on CruxEval-O and 88.3% on LCB-Exec, and notably better than the results of Qwen2.5-Coder-7B-Instruct, a code-specific model trained on black-box data. These results indicate the effectiveness of ET-CoT for enhancing CER ability.

Table 2b reports the results of previous models from Ding et al. (2024a). Among that, SemCoder family is representative within fine-tuning methods using intermediate reasoning steps, and also the most performant. ET-CoT also surpasses this baseline, which we attribute to these factors: our specialization in CER, the finer granularity of reasoning steps, and the reduced variability of execution traces generated by PyTracify, compared to SemCoder, which relies on LMs to generate traces.

Limitations of the training on direct outputs The results of SFT and GRPO yielded only limited improvements. Regarding SFT, the failure reflects a fundamental limitation of approaches without



(a) Addition (b) Jump game DP (c) Interval scheduling (d) BFS

Figure 7: ET-CoT mitigates the initial instability of iterative code simulation.

intermediate reasoning steps, as we showed in Section 3.1 that even a single syntax requires CoT, and argued in Section 4.5 that the length of CoT must adapt to the problem. This observation is consistent with Gu et al. (2024b), who showed that direct-output fine-tuning on CruxEval was not effective to improve the score on CruxEval. On the other hand, regarding GRPO, this failure cannot be attributed to the lack of intermediate steps as the generation length increased during training. A plausible explanation is that when the base model capability is insufficient, the reward based on outputs becomes sparse and noisy, which is a well-known issue in RL (Lightman et al.).

Finally, we remark that the proportion of cases in which the execution traces generated by fine-tuned models with ET-CoT match the traces produced by PyTracify are analyzed in Appendix E.

5.2 DID ET-COT MITIGATE THE LIMITATIONS IN BASIC OPERATIONS?

After confirming the effectiveness of ET-CoT on standard benchmarks, a finer-grained question is whether ET-CoT actually mitigates the limitations observed in Section 3. Therefore, we applied the same experimental setup as in Section 3 to models fine-tuned with ET-CoT.

Basic syntax understanding. Table 3 presents the results. We observed substantial improvements on basic syntax understanding through ET-CoT in three out of four models. In contrast, the performance of Qwen3 decreased. Regarding this, we found generations that skipped the CoT, suggesting that the model judged that no reasoning effort was required for this simple task even after the fine-tuning. This suggests that training reasoning models as interpreters is difficult, but starting from base models may help avoid this issue.

Table 3: Comparison of the basic syntax understanding before and after ET-CoT.

Model	Original	ET-CoT
Llama3.1-8B-Inst	34.5	**67.4**
Qwen2.5-7B-Inst	45.3	**70.3**
Qwen3-8B	**84.8**	73.4
OLMo2-7B-Inst	16.3	**53.3**

Iterative code simulation. The results for Llama3.1-8B and Qwen2.5-7B with $n = 5$ are presented in Figure 7. Full results are in Appendix H. Alongside the original models, we include a comparison with SemCoder-S (Ding et al., 2024a), as a representative natural language–based fine-tuning method.

Here, while the original models were unable to solve any of the tasks, ET-CoT resulted in improved accuracy for both models across all tasks. The improvement is significantly greater than that achieved by SemCoder. Moreover, the initial instability, namely a failure in the early steps despite the ability to stably conduct later repetitions, was generally mitigated, except in the case of Qwen2.5-7B on (c) Interval scheduling. Therefore, we conclude that ET-CoT mitigates the problem of initial instability in iterative code simulation.

5.3 ABLATION STUDIES

We conducted an ablation study to identify critical aspects of ET-CoT. Specifically, we focused on the trace format and the dataset composition. We used Llama3.1-8B-Instruct for all experiments.

Table 4: Token length statistics and accuracy of different execution trace formats.

Format	Min	Max	Mean	CruxEval-O	LCB-Exec
Full Trace	163	36,274	923.56	67.75	88.52
No-Loop Internals Trace	152	24,610	694.22	60.50 (-7.25)	84.34 (-4.18)
Minimal Trace	152	20,133	648.94	50.88 (-9.62)	63.25 (-21.09)

Table 5: Effect of dropping different subsets of the Custom Dataset (pass@1)

Training Variant	#Samples (Total)	#Samples (Custom)	CruxEval-O	LCB-Exec
Full data	127,413	38,879	67.75	88.52
No Tokenizer Vocabulary	100,534	12,000	66.25 (-1.5)	88.41 (-0.09)
No String Functions	115,413	26,879	62.88 (-4.87)	87.27 (-1.25)
No Custom Dataset	88,534	0	57.49 (-10.26)	85.38 (-3.14)

5.3.1 EFFECT OF THE EXECUTION TRACE FORMAT

To isolate the effect of trace granularity, we compared three formats. (1) **Full Trace**, as used in the above experiments; (2) **No-Loop Internals Trace**, which omits repeated internal entries generated inside loops to avoid redundancy; (3) **Minimal Trace**, which omits explicitly checking operand values through entries (e.g., `BinOpLeft` and `CompareRight`) during binary operations and comparisons.

As shown in Table 4, the full trace yielded the highest scores on both datasets, and performance deteriorates as traces are shortened. This suggests that accurate CER requires explicit unrolling of iterations, and that small-sized LMs need multi-step CoT even when processing a single syntax. The former observation is consistent with the discussion in Section 4.5, and the latter with Section 3.1.

5.3.2 DATASET ABLATION

In Section 4.3, we created Custom Dataset to address typical weakness of LMs. To quantify the contribution of it, we tested four variants of the dataset: (1) **Full Dataset** — the complete dataset; (2) **No String Functions**, which deletes 12,000 string function examples from the custom subset; (3) **No Tokenizer Vocabulary**, which omits the Llama-token vocabulary subset from the custom subset; (4) **No Custom Dataset**, which removes all custom subsets (string functions and tokenizer vocabulary).

Table 5 shows that the full dataset yields the highest accuracy, supporting the usefulness of the two custom subsets. A more fine-grained view reveals that removing string functions has a larger impact than removing the tokenizer vocabulary. However, removing both simultaneously results in more than double the degradation caused by removing string functions alone. This trend suggests that the effects of the tokenizer vocabulary and string functions are not independent.

6 CONCLUSION

This work emphasized the effectiveness of systematically accumulating fine-grained reasoning steps for code execution reasoning (CER). We investigated ET-CoT, which generates execution traces using our custom code interpreter and uses them as CoT rationales. ET-CoT improved various 8B-class LMs, establishes new baselines for the class, and mitigates failures in the basic components of CER.

Finally, we note limitations of this work arising from its specialization to CER. First, we relied on CruxEval and LiveCodeBench as evaluation datasets, but we were unable to identify other widely accepted benchmarks for CER. Second, in practice, CER only becomes meaningful when integrated with other coding-related capabilities such as code generation, whereas this work specialized the models as interpreters. Nevertheless, we believe that deepening understanding on a single aspect of coding abilities and setting baselines that outperform a code model trained on black-box data from the same base model provide a useful stepping stone toward building strong coding models.

REPRODUCIBILITY STATEMENT

We provide source code and configuration files as part of the supplementary materials, where instructions for running the code and constructing the dataset are also included. The dataset used for training and evaluation is shared via an anonymous Google Drive link. Information on data preprocessing, model hyperparameters (learning rate, batch size, random seed), and library versions can be found in the scripts and instruction text within the supplementary materials. Our main experiments were conducted on 8 NVIDIA H100 GPUs, each equipped with 80 GB of memory.

THE USE OF LARGE LANGUAGE MODELS

The authors used an LLM to correct grammatical mistakes and polish the text into more natural expressions during the preparation of this paper. Therefore, the use of the LLM was limited to line-level generation.

REFERENCES

Jordi Armengol-Estapé, Quentin Carbonneaux, Tianjun Zhang, Aram H Markosyan, Volker Seeker, Chris Cummins, Melanie Kambadur, Michael FP O'Boyle, Sida Wang, Gabriel Synnaeve, et al. What i cannot execute, i do not understand: Training and evaluating llms on program execution traces. *arXiv preprint arXiv:2503.05703*, 2025.

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

Satwik Bhattamishra, Arkil Patel, and Navin Goyal. On the computational power of Transformers and its implications in sequence modeling. In *Proceedings of the 24th Conference on Computational Natural Language Learning (CoNLL)*, pp. 455–475. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.conll-1.37. URL `https://aclanthology.org/2020.conll-1.37/`.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. Reasoning runtime behavior of a program with LLM: How far are we? *arXiv preprint arXiv:2403.16437*, 2024.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

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

Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. Traced: Execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–12, 2024b.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 2024a.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 2024b.

Hugging Face. Open r1: A fully open reproduction of deepseek-r1, January 2025. URL `https://github.com/huggingface/open-r1`.

Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. In *International Conference on Machine Learning*, pp. 11398–11442. PMLR, 2023.

Dirk Groeneveld, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Harsh Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, et al. Olmo: Accelerating the science of language models. *arXiv preprint arXiv:2402.00838*, 2024.

Alex Gu, Wen-Ding Li, Naman Jain, Theo X Olausson, Celine Lee, Koushik Sen, and Armando Solar-Lezama. The counterfeit conundrum: Can code language models grasp the nuances of their incorrect generations? *arXiv preprint arXiv:2402.19475*, 2024a.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 16568–16621. PMLR, 21–27 Jul 2024b. URL `https://proceedings.mlr.press/v235/gu24c.html`.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL `https://arxiv.org/abs/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. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024a. URL `https://arxiv.org/abs/2409.12186`.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024b.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.

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, 2024. URL `https://arxiv.org/abs/2403.07974`.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

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

Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. Codei/o: Condensing reasoning patterns via code input-output prediction. *arXiv preprint arXiv:2502.07316*, 2025.

Qian Li and Yuyi Wang. Constant bit-size Transformers are Turing complete, 2025. URL `https://arxiv.org/abs/2506.12027`.

Zhiyuan Li, Hong Liu, Denny Zhou, and Tengyu Ma. Chain of thought empowers transformers to solve inherently serial problems. *arXiv preprint arXiv:2402.12875*, 1, 2024.

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

Changshu Liu and Reyhan Jabbarvand. A tool for in-depth analysis of code execution reasoning of large language models. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pp. 1178–1182, 2025.

Changshu Liu, Yang Chen, and Reyhaneh Jabbarvand. Assessing large language models for valid and correct code reasoning, 2025. URL `https://openreview.net/forum?id=2umZVWYmVG`.

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

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL `https://arxiv.org/abs/1711.05101`.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024. URL `https://arxiv.org/abs/2402.19173`.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. Lms: Understanding code syntax and semantics for code analysis. *arXiv preprint arXiv:2305.12138*, 2023.

Marina Mancoridis, Bec Weeks, Keyon Vafa, and Sendhil Mullainathan. Potemkin understanding in large language models. *arXiv preprint arXiv:2506.21521*, 2025.

William Merrill and Ashish Sabharwal. A logic for expressing log-precision Transformers. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, 2023. URL `https://papers.nips.cc/paper_files/paper/2023/hash/a48e5877c7bf86a513950ab23b360498-Abstract-Conference.html`.

William Merrill and Ashish Sabharwal. The expressive power of Transformers with chain of thought. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL `https://openreview.net/forum?id=NjNGlPh8Wh`.

Nan-Do. Atcoder contests dataset. `https://huggingface.co/datasets/Nan-Do/atcoder_contests`, 2023. Accessed: 2025-09-05.

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

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. 2021.

OLMo 2 team. 2 OLMo 2 Furious, 2025. URL `https://arxiv.org/abs/2501.00656`.

Jorge Pérez, Javier Marinković, and Pablo Barceló. On the Turing completeness of modern neural network architectures. In *International Conference on Learning Representations (ICLR)*, 2019. URL `https://openreview.net/forum?id=HyGBdo0qFm`.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is turing-complete. *Journal of Machine Learning Research*, 22(75):1–35, 2021.

Qwen team. Qwen2.5 technical report, 2025. URL `https://arxiv.org/abs/2412.15115`.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL `https://arxiv.org/abs/2308.12950`.

Dale Schuurmans, Hanjun Dai, and Francesco Zanini. Autoregressive large language models are computationally universal, 2024. URL `https://arxiv.org/abs/2410.03170`.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL `https://arxiv.org/abs/2402.03300`.

Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity. *arXiv preprint arXiv:2506.06941*, 2025.

Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.

Jian Wang, Xiaofei Xie, Qiang Hu, Shangqing Liu, and Yi Li. Do code semantics help? a comprehensive study on execution trace-based information for code large language models. In *Submitted to ACL Rolling Review - May 2025*, 2025. URL `https://openreview.net/forum?id=j1HUJ5glpx`. under review.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: empowering code generation with oss-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

Kevin Xu and Issei Sato. To CoT or to loop? a formal comparison between chain-of-thought and looped Transformers, 2025. URL `https://arxiv.org/abs/2505.19245`.

Ruiyang Xu, Jialun Cao, Yaojie Lu, Ming Wen, Hongyu Lin, Xianpei Han, Ben He, Shing-Chi Cheung, and Le Sun. Cruxeval-X: A benchmark for multilingual code reasoning, understanding and execution. *arXiv preprint arXiv:2408.13001*, 2024.

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

Zeping Yu and Sophia Ananiadou. Interpreting arithmetic mechanism in large language models through comparative neuron analysis. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3293–3306, 2024.

Xiyu Zhai, Runlong Zhou, Liao Zhang, and Simon Shaolei Du. Transformers are efficient compilers, provably. *arXiv preprint arXiv:2410.14706*, 2024.

Ziqian Zhong, Ziming Liu, Max Tegmark, and Jacob Andreas. The clock and the pizza: Two stories in mechanistic explanation of neural networks. *Advances in neural information processing systems*, 36:27223–27250, 2023.

A REMARK ON ARMENGOL-ESTAPÉ ET AL. (2025)

This paper was originally presented at the ICML 2025 *Programmatic Representations for Agent Learning* workshop. After that, we found that there was independent work which also leverages execution traces, namely Armengol-Estapé et al. (2025). In their study, 1.5M execution traces were generated using `sys.settrace`, and Llama3.1-8B-Instruct (Dubey et al., 2024b) was fine-tuned. They reported 79.7% accuracy on CruxEval (Gu et al., 2024b) and further described the successful simulation of 14k execution steps for a specific function (a 4-bit binary counter), which is indeed impressive.

Nevertheless, for the following reasons, a fair comparison between their results and ours cannot be established. Their paper states only that the 1.5M data points originate from “unrestricted Python code,” without any details regarding the nature of the data (e.g., source and length distribution). Moreover, the method for generating training data from execution traces is not made explicit. In particular, the reported 79.7% result on CruxEval is achieved with the *Compact Scratchpad* method, but their explanation consists solely of the following sentence: *“Inspired by Ni et al. (2024)’s trace representation, we also consider a diff-based scratchpad, in which the model only needs to predict the variables that change with respect to the previous state. This should help at long executions by decreasing the token count. Note, though, that in Ni et al. (2024) this representation was not used as a scratchpad, but to annotate code.”* We also note that the baseline they report for Llama3.1-8B-Instruct on CruxEval differ from ours, but we reported pass@1 results using the CoT prompt used in CruxEval (Listing 21 of Gu et al. (2024b)).

Apart from the shared emphasis on execution traces, the directions of the two studies diverge. They trained a language model on 1.5M execution trace-based examples, a scale that goes beyond typical fine-tuning datasets, with an intention to enable long-horizon simulations. By contrast, the present work concentrates on enabling language models to solve short code executions, as exemplified by CruxEval (Gu et al., 2024b) and LiveCodeBench (Execution) (Jain et al., 2024). Specifically, we started from the investigation of the bottlenecks of 8B-class models, revealing limitations in syntax-level prediction ability (Section 3.1) and identifying fluctuations in CoT reasoning that hinder precise operation execution (Section 3.2). These findings underscore the necessity of achieving fine-grained and fluctuation-free CoT, and the use of execution traces is positioned as a means to address this challenge.

B INTERPRETER DIFFERENCES BETWEEN CPYTHON AND PYTRACIFY

While CPython and PyTracify share the same surface syntax, they differ in how variable names inside nested functions are *bound on assignment*. In particular:

- **Reads (name lookup).** PyTracify follows the same **LEGB** rule as CPython—**L**ocal → **E**nclosing → **G**lobal → **B**uilt-in. Thus, free-variable reads in an inner function can see bindings from an enclosing function scope.
- **Writes (assignment binding).** Unlike CPython, where an assignment in an inner function creates a *new local* binding unless the name is explicitly declared `nonlocal` or `global`, PyTracify uses *default-`nonlocal`* semantics: if a name exists in the nearest *enclosing function scope*, an inner assignment *updates that enclosing binding by default*. If there is no such enclosing binding, the assignment creates/uses a local as usual. The behavior of `global` matches Python.

Intuitively, PyTracify behaves *as if* `nonlocal` were implicitly in effect for names that already exist in the nearest enclosing function scope. This choice simplifies the interpreter/tracer while leaving idiomatic programs unaffected.

Listing 6 Identical behavior under CPython and PyTracify

```python
def outer():
    x = "enclosing"
    def inner():
        print(x)    # CPython: "enclosing" | PyTracify: "enclosing"
```

```
    inner()
outer()
```

In Listing 6, both CPython and PyTracify resolve `x` in `inner()` via the *Enclosing* scope.

Listing 7 Writes in an inner function: CPython vs. PyTracify

```
x = "global"

def outer():
    x = "enclosing"
    def inner():
        x = "local"
        print(x)   # CPython: "local"  | PyTracify: "local"
    inner()
    print(x)       # CPython: "enclosing"
                  # PyTracify: "local"   (inner updated the enclosing x
                      )

outer()
print(x)           # CPython: "global"  | PyTracify: "global"
```

As shown in Listing 7, CPython treats `x` assigned in `inner()` as a new *local* binding (unless `nonlocal x` is declared), so `outer()` prints `"enclosing"`. PyTracify instead *rebinds the enclosing* `x` by default, so `outer()` prints `"local"`.

Additionally, to make character-level semantics more transparent to the model, PyTracify overrides Python's built-in function `len` as shown below.

Listing 8 Pedagogical override of `len` to expose character-level iteration

```
def len(target: object) -> int:
    cnt = 0
    for element in target:
        cnt += 1
    return cnt
```

With the override in Listing 8, calling `len("hello")` produces the execution trace in Listing 9.

Listing 9 Execution trace produced by `len("hello")` under the override

```
0 Statement
1 Expr len("hello")
2 Call len("hello")
2 CallArg0 "hello"
3 Constant 'hello'
2 Statement
3 Assign cnt = 0
4 Constant 0
3 Assign cnt = 0
2 Statement
3 For for element in target:  # type:
4 Name target = hello
3 Statement
4 AugAssign cnt += 1
5 Constant 1
4 AugAssign cnt = 1
3 Statement
4 AugAssign cnt += 1
5 Constant 1
4 AugAssign cnt = 2
3 Statement
4 AugAssign cnt += 1
```

```
5 Constant 1
4 AugAssign cnt = 3
3 Statement
4 AugAssign cnt += 1
5 Constant 1
4 AugAssign cnt = 4
3 Statement
4 AugAssign cnt += 1
5 Constant 1
4 AugAssign cnt = 5
2 Statement
3 Return return cnt
4 Name cnt = 5
2 Call len('hello') = 5
```

Because LLMs typically process strings as sequences of subword tokens rather than individual characters, they can be brittle on strictly character-level reasoning. By overriding `len` to surface the per-character iteration in the execution trace, PyTracify makes the character-level semantics explicit and easier for the model to learn.

C Training and Inference Prompt for ET-CoT

Listing 10 Training prompt format for the ET-CoT training dataset

```
System:
You are a highly capable assistant. Your task is to estimate the output
     of the given Python code.
The reasoning process and output are enclosed within <think> </think>
    and <answer> </answer> tags, respectively,
i.e., <think> reasoning process here </think><answer> output here </
    answer>
User:
<code>code</code> <input>input</input>.
Assistant:
<think>trace</think><answer>output</answer>
```

Listing 10 is the training and inference prompt format for the ET-CoT trainig dataset. The user supplies the Python code for execution and any required standard input values. The assistant then provides the execution trace (generated using PyTracify), encapsulated within `<think>` tags, followed by the code’s final output, encapsulated within `<answer>` tags.

At inference time, we provide only the code and input (the “User” portion of the training format; the model generates both the reasoning and the final `<answer>`. For ET-CoT models, we prefix the prompt with the token sequence `0\n Statement` to trigger the ET-CoT procedure. Correctness is evaluated by matching the model’s generated answer to the ground-truth output—this is the same evaluation protocol used in CruxEval and LiveCodeBench. Because there is no need to introduce diversity in the outputs, we use a temperature of 0 and top-k sampling with $k = 1$.

D Custom Dataset Example

Listing 11 Example of String Functions Dataset

```
<code>
# removeprefix: Remove the specified prefix from the start of the
    string if present.
# Example: \"unhappy\".r emoveprefix(\"un\") -> \"happy\"
print('koalawatermelonslow'.removeprefix('koa'))
</code>
```

Listing 11 is an example from the **String Functions Dataset** described in Section 4.3. It shows the case of the `removeprefix` function, where the behavior of `removeprefix` is explained in comments inside the `<code></code>` block, along with an example of applying the function to a random string. The same structure—commented explanation of the function's behavior plus an example of string manipulation—is used for other functions such as `len`, `slice`, `replace`, `rpartition`, `find`, `join`, and `rstrip`.

E MATCH RATE BETWEEN ET-COT AND PYTRACIFY

Table 12: ET-CoT Llama3.1-8B performance and trace-matching statistics. The table reports the pass@1 accuracy on CruxEval-O, the overall agreement between ET-CoT and PyTracify traces, and the proportions of correct and incorrect predictions among the trace-matched subset.

Metric	**Value (%)**
CruxEval-O Accuracy (ET-CoT Llama3.1-8B)	67.75
ET-CoT / PyTracify Trace Agreement	52.63
Correct & Trace-Matched	52.38
Incorrect & Trace-Matched	0.25

Table 12 shows the proportion of cases in which the execution traces generated by Llama 3.1-8B with ET-CoT match the traces produced by PyTracify. From the table, we can see that the ET-CoT–fine-tuned Llama 3.1-8B achieves a pass@1 accuracy of 67.75%. Overall, 52.63% of outputs had traces that matched PyTracify's. Among these trace-matched cases, 52.38% were correct predictions, while 0.25% were incorrect despite the traces matching. The 0.25% of cases where the trace matched perfectly but the prediction was incorrect correspond to just two samples: one where the correct answer consisted of seven consecutive spaces but the model predicted an empty string, and another where the expected answer was `'  4 2 '` (with leading and trailing spaces) but the model predicted `'4 2'` without those spaces.

F PROOF OF PROPOSITION 1

There are several theoretical works on the computational power of Transformers with and without CoT. Li et al. (2024) proved that CoT corresponds to the size of boolean circuits which can be solved by the Transformer with that CoT length. On the other hand, without CoT, its computational power is bounded above by uniform TC^0, according to Merrill & Sabharwal (2024). Here TC^0 is the class of problems efficiently solvable by basic (constant-depth and polynomial-size Boolean) circuits. Also, according to Merrill & Sabharwal (2023), without CoT of sufficient length, the expressivity of Transformers collapses to low-level circuit classes. These results highlight the crucial role of CoT design if we want LMs to "think like a computer" (to run programs of certain complexities).

Here, by borrowing the result of Merrill & Sabharwal (2024), we emphasize that CoT of adaptive length is necessary even to simulate simple programs. Specifically, we discuss the problem class of NL, which is the class decidable in logarithmic space nondeterministically. NL-hard denotes the set of problems to which every problem in NL reduces within log space. Example of such problems include reachability in directed graphs, 2-SAT, and NFA simulations (Sipser, 1996). On the other hand, L is the class of problems decidable deterministically in logarithmic space, simpler than NL. It is generally believed that $\mathsf{L} \neq \mathsf{NL}$.

Proposition 1. *Assume that* $\mathsf{L} \neq \mathsf{NL}$. *Then, solving an* NL*-hard problem with a log-precision (*$O(logm)$ *precision for* m *decoding steps) decoder-only transformer with strict causal masking (each position attends only to earlier tokens), saturated attention (idealized hard attention), and projected pre-norm (apply linear projection before layer normalization for each sublayer) requires* $\omega(\log n)$ *intermediate decoding steps as a function of the input length* n*.*

Proof. Suppose that there exists an NL-hard problem A that can be solved by such a Transformer with $O(\log n)$ intermediate decoding steps. Every problem B in NL reduces to A via a log-space reduction by the definition of NL-hardness, and from Theorem 4 of Merrill & Sabharwal (2024) with $t(n) = O(\log n)$, composing this reduction with an L algorithm for A keeps us in L, so $B \in \mathsf{L}$. Hence we obtain that $\mathsf{NL} \subseteq L$.

Since by definition $\mathsf{L} \subseteq \mathsf{NL}$, we would have $\mathsf{L} = \mathsf{NL}$, contradicting the standing assumption $\mathsf{L} \neq \mathsf{NL}$. Therefore, NL-hard problems cannot be solved with only $O(\log n)$ intermediate decoding steps. □

This proposition suggests that to solve NL problems we need to adapt the number of CoT steps, and we cannot abbreviate the intermediate reasoning steps into a constant number of steps. Due to an residual factors in Merrill & Sabharwal (2024) which cannot be ignored for the NL class, we could only state the growth speed of $\omega(\log n)$. Thus it is a future work to state stronger claims, while keep focusing on a simple class of problems realistic to be included in actual CER benchmarks.

G DETAILS OF THE EXPERIMENT IN SECTION 3.1

Here, we describe the data generation and evaluation pipeline in detail. Specifically, the complete intermediate process to transform CruxEval data into problems such as that in Figure 3, including the prompts used, as well as examples of the generations, can be found at the end of this section in "Example of the data generation and evaluation pipeline." The experiments were conducted using models available on OpenRouter (`https://openrouter.ai/`), except for OLMo-2-1124-7B-Instruct (OLMo 2 team, 2025), which we ran locally with vLLM (Kwon et al., 2023).

G.1 DATA GENERATION PIPELINE

The original CruxEval dataset (Gu et al., 2024b) consists of functions with 3–13 lines and corresponding input–output examples. From this, we constructed functions to evaluate line-level execution simulation. First, using Gemini 2.5 Pro (Comanici et al., 2025), we extracted the line deemed most essential from each original function. Next, we created a minimal function containing only that line (with minimal scaffolding if strictly necessary). For each such function, we generated input–output pairs, ensuring that the new function's behavior matched the execution of the extracted line in the original function.

From the 800 original samples, we thus produced new functions and input–output pairs. Among them, about ten were either non-executable or had mismatched pairs; we applied minimal manual fixes to these cases, resulting in 800 executable functions with correct input–output pairs.

G.2 EVALUATION

On this dataset, we have evaluated OLMo-2-1124-7B-Instruct (OLMo 2 team, 2025), Llama-3.1-8B-Instruct (Dubey et al., 2024a), Llama-3.1-70B-Instruct (Dubey et al., 2024a), Qwen2.5-7B-Instruct (Qwen team, 2025), Qwen2.5-72B-Instruct (Qwen team, 2025), Qwen3-8B (Yang et al. (2025), think mode for CoT and explanation, non-think mode for direct prediction), Ministral-8B, Gemini-2.0-Flash, and GPT-4.1. In our design, the models are divided into three groups. The first group is a collection of recent models of 7–8B scale, corresponding to OLMo-2-1124-7B-Instruct, Llama-3.1-8B-Instruct, Qwen2.5-7B-Instruct, Qwen3-8B, and Ministral-8B. The second group comprises larger models from the same series as those in the first group, namely Llama-3.1-70B-Instruct and Qwen2.5-72B-Instruct. Finally, we selected as popular commercial models Gemini-2.0-Flash, and GPT-4.1. Our choice to focus on non-thinking models is motivated by the fact that closed-source thinking models cannot be forced to provide direct answers (without reasoning), which would lead to missing data.

We add a note on natural language descriptions. This experiment is based on the hypothesis that a known failure mode of LMs—being able to explain a concept but failing to apply it to concrete instances (Mancoridis et al., 2025)—also occurs in code understanding. There are several possible ways to assess the correctness of code descriptions generated by LMs, but we regard the strictest criterion as whether such descriptions alone allow simulation of code execution to produce the correct output. To this end, after having each LM describe the code, we asked GPT-4.1, which demonstrated strong code execution simulation ability, to generate output predictions by simulating execution solely based on the descriptions (without seeing the function itself). We then compared these predictions against the ground-truth outputs for evaluation.

During GPT-4.1's execution simulation, occasional errors occurred, which we believe caused some downward bias in accuracy. On the other hand, the chance of obtaining the correct output from an incorrect description is very low, making upward bias unlikely. Therefore, for the purpose of showing that the accuracy of natural language descriptions exceeds that of direct execution simulation, the inherent error in this evaluation method does not become a problem.

Listing 13 Example of the data generation and evaluation pipeline

Original problem from CruxEval:

Function:

```
def f(s, o):
    if s.startswith(o):
        return s
    return o + f(s, o[-2::-1])
```

Input:

```
s = 'abba'
o = 'bab'
```

Output:

```
'bababba'
```

Prompt for Gemini 2.5-Pro when transforming the code:

System prompt:

```
You are a dataset transformation assistant.

You will work with problems from Cruxeval, a code execution dataset:
- Each problem is (code, input, output).
- Given code and input, the original task is to predict the execution
    output.

Before transforming, simulate the original code step by step to locate
    the first moment when the single most essential line is executed and
     to capture the exact variable state at that moment.

Transformation rules:
1) Choose exactly one "most essential" line.
2) Write a minimal function that contains only that line (plus minimal
    scaffolding if strictly necessary).
3) Creating the Input: use the exact variable state when that line
    first executes as the new inputs.
   Write them in the same comma-separated style shown in the examples.
   Keep it minimal but sufficient to execute the line once.
4) Creating the Output: the return value produced by the new simplified
     function after executing the essential line exactly once.

You may output reasoning before the transformed data, but always end
    with the following format:

code:
<transformed code>

input:
<transformed input>

output:
<transformed output>

---

### Example 1

Original:
code:
```

```
def f(text, value):
    text_list = list(text)
    text_list.append(value)
    return ''.join(text_list)

input:
'bcksrut', 'q'

output:
'bcksrutq'

Transformed:
code:
def f(text_list, value):
    text_list.append(value)
    return text_list

input:
text_list = ["b", "c", "k", "s", "r", "u", "t"]
value = "q"

output:
["b", "c", "k", "s", "r", "u", "t", "q"]

---

### Example 2

Original:
code:
def f(names):
    numberOfNames = 0
    for i in names:
        if i.isalpha():
            print(i, numberOfNames)
            numberOfNames += 1
    return numberOfNames

input:
['sharron', 'Savannah', 'Mike Cherokee']

output:
2

Transformed:
code:
def f(name):
    if name.isalpha():
        return 1
    else:
        return 0

input:
name = 'sharron'

output:
1
```

User prompt:

```
Transform the following Cruxeval problem according to the System
   instructions.

code:
def f(s, o):
```

```
    if s.startswith(o):
        return s
    return o + f(s, o[-2::-1])

input:
'abba', 'bab'

output:
'bababba'
```

Trace of Gemini 2.5-Pro when transforming the code:

```
`f` is a recursive function. Let's trace its execution with the given
    inputs.

1.  **`f(s='abba', o='bab')`**:
    *   The condition `s.startswith(o)` (`'abba'.startswith('bab')`) is
         `False`.
    *   The `else` block is executed. It must compute `o + f(s, o
        [-2::-1])`.
    *   This is the first time the most essential part of the logic is
        reached. The crucial operation here is the calculation of the
        new value for `o` for the recursive call.
    *   The expression `o[-2::-1]` is evaluated. `o` is `'bab'`. The
        slice starts at the second-to-last character ('a') and goes
        backward to the beginning. The result is `'ab'`.
    *   The execution continues with a recursive call: `f('abba', 'ab')
        `.

2.  **`f(s='abba', o='ab')`**:
    *   The condition `s.startswith(o)` (`'abba'.startswith('ab')`) is
        `True`.
    *   The function returns `s`, which is `'abba'`.

3.  **Returning to the first call**:
    *   The expression from step 1 is now resolved: `'bab' + 'abba'`.
    *   The final return value is `'bababba'`.

The most essential line is `return o + f(s, o[-2::-1])`. The single
    most critical operation within that line, executed at the first
    opportunity, is the slice `o[-2::-1]`, which generates the argument
    for the subsequent recursive call and drives the algorithm forward.
    We will isolate this operation.

-   **Essential line chosen:** The slice operation `o[-2::-1]` from the
     line `return o + f(s, o[-2::-1])`.
-   **State at first execution:** The variables available when this
    slice is first computed are `s = 'abba'` and `o = 'bab'`. Only `o`
    is needed for the slice.
-   **Minimal function:** A function that takes `o` and returns the
    result of the slice.
-   **Input:** The value of `o` at that moment: `'bab'`.
-   **Output:** The result of `'bab'[-2::-1]`, which is `'ab'`.

code:
def f(o):
    return o[-2::-1]

input:
o = 'bab'

output:
'ab'
```

Transformed problem:

Function:

```
def f(o):
    return o[-2::-1]
```

Input:

```
o = 'bab'
```

Output:

```
'ab'
```

Prompt for prediction:

System prompt:

```
You are a highly capable assistant. Your task is to predict the exact
    value that the final function call evaluates to in the given Python
    code. Please describe your thought process briefly but clearly
    enough so that someone can verify your thought process, enclosed in
    <reason>...</reason>, and put the prediction of the code output in <
    answer>...</answer>.

Strictly follow the format:
<reason>
(thought process here)
</reason>
<answer>
(only the final output here)
</answer>
```

User prompt:

```
Predict the return value of the final function call of the code,
    according to the system instructions.

code:
def f(o):
    return o[-2::-1]

o = 'bab'

f(o)
```

Generated predictions:

[GPT-4.1] (incorrect)

Answer:

```
'ba'
```

Reasoning:

```
The function f(o) takes a sequence o and returns o[-2::-1]. In Python
    slice notation, [start:stop:step], so o[-2::-1] starts at the second
    -to-last element and goes backwards to the beginning. For o = 'bab',
     o[-2] is 'a', so the slice is from index 1 to before the beginning,
     stepping by -1: o[1::-1], which is 'ab' reversed, i.e., 'ba'.
```

[Gemini 2.0 Flash] (correct)

Answer:

```
'ab'
```

Reasoning:

```
The function `f` takes a string `o` as input.
It returns a slice of the string `o`. The slice `o[-2::-1]` starts from
     the second to last element (index -2), goes to the beginning of the
     string, and reverses the string.
In this case, o = 'bab'. So the slice starts from index -2 which is 'a
    ', and goes to the beginning with a step of -1. So the reversed
    slice is 'ab'.
```

[Llama3.1-8B] (incorrect)

Answer:

```
'abab'
```

Reasoning:

```
Upon executing the function call f(o) with the input string 'bab', we
    first need to reverse the entire string except for the last two
    characters. Since the indexing in Python is 0-based, '-2::-1' means
    start from the second last character ('b') and move backwards,
    stepping by -1 each time. However, since we're looking to exclude
    the last character and only remove the second last digit then
    overwrite the first 3rd digit of the 'o' list with the rest, a
    simpler method would be to get the first 1 character and put it last
     here while the first 2 characters then get reversed leading to
    reverse offirst three digits of string except last two, return the
    rest: ('a','b',''), and appending them at the start afteroverwriting
     '' leaving ('a','b','b') excluding the first & second part of the
    repeating ('b') block that were moved away substituting for their
    normal counterparts of index location {0, }, return the reversed
    list resulting sting: 'abab'.
```

[Llama3.1-70B] (incorrect)

Answer:

```
'aba'
```

Reasoning:

```
The function f(o) takes a string o as input and returns the substring
    from the second last character to the beginning of the string in
    reverse order. The slicing operation o[-2::-1] starts from the
    second last character (index -2), moves backwards to the beginning
    of the string (step -1), and includes all characters in between.

In this case, the input string o is 'bab'. Applying the slicing
    operation, we start from the second last character 'a', move
    backwards to the beginning of the string, and include all characters
     in between. This results in the substring 'aba'.

However, since the slicing operation moves backwards, the resulting
    substring is in reverse order, so the actual return value of the
    function is 'aba' spelled backwards, which is 'aba' itself since it
    is a palindrome.
```

[Qwen2.5-7B] (incorrect)

Answer:

```
'b'
```

Reasoning:

```
The function `f(o)` operates on the string `o` which is 'bab'.
1. In the function, the slice `o[-2::-1]` is used. This starts from the
     second last character (i.e., the index -2, which is 'b') and goes
    backwards to the start of the string (index -3, which would be out
    of range, so it starts from 'b') step by -1.
2. So, it reverses the string from the second last character to the
    start, which results in 'b'.
```

[Qwen2.5-72B] (correct)

Answer:

```
'ab'
```

Reasoning:

```
The function `f` takes a single argument `o` which is expected to be a
    string. It returns a slice of the string starting from the second
    last character to the beginning of the string, in reverse order.

Given the input `o = 'bab'`:
- `o[-2]` is 'a' (the second last character).
- `o[-3]` is 'b' (the first character, accessed in reverse order).

So, the slice `o[-2::-1]` will return the substring 'ab'.
```

Prompt for generating explanations:

System prompt:

```
You will be given a Python function along with input and output
    examples. Explain what the function does, briefly but clearly enough
     so that someone could write the same code after reading it. Your
    response should consist only of the explanation.

---

### Example 1

function:
def f(text_list, value):
    text_list.append(value)
    return text_list

example input:
text_list = ["b", "c", "k", "s", "r", "u", "t"]
value = "q"

example output:
["b", "c", "k", "s", "r", "u", "t", "q"]

explanation:
This function takes as input a list of single-character strings called
    text_list and a single-character string called value, and it outputs
     a list of single-character strings. The output is text_list with
    value appended to the end.

### Example 2

function:
def f(name):
    if name.isalpha():
        return 1
    else:
        return 0

example input:
name = "sharron"

example output:
1

explanation:
This function takes as input a string called name and outputs an
    integer. The output is 1 if the string name consists only of
    alphabetic characters (a-z, A-Z) and is non-empty; otherwise, the
    output is 0.
```

User prompt:

```
Generate an explanation from the following, according to the system
    instructions.

function:
def f(o):
    return o[-2::-1]

example input:
o = 'bab'

example output:
f(o)
```

Generated explanations:

[GPT-4.1] (correct)

```
This function takes as input a string called o and outputs a string.
    The output is the substring of o starting from the second-to-last
    character and moving backward to the beginning of the string,
    effectively reversing all characters of o except for the last one.
```

[Gemini 2.0 Flash] (correct)

```
This function takes as input a string called o and returns a new string
     consisting of all characters in o except for the last one, in
    reverse order.
```

[Llama3.1-8B] (correct)

```
This function takes as input a string o and returns a substring
    consisting of all characters of o except the last one, in reverse
    order.
```

[Llama3.1-70B] (correct)

```
This function takes as input a string o and outputs a string. The
    output is the substring of o starting from the second-to-last
    character and moving backwards to the beginning of the string.
```

[Qwen2.5-7B] (correct)

```
This function takes as input a string o and outputs a string. The
    output is the substring of o containing all characters except the
    last one, taken in reverse order.
```

[Qwen2.5-72B] (correct)

```
This function takes as input a string o and outputs a string. The
    output is the substring of o starting from the second-to-last
    character (inclusive) to the beginning of the string, in reverse
    order.
```

H DETAILS OF THE EXPERIMENT IN SECTION 3.2

H.1 PREPARATION OF THE EXAMPLE CODES

Selection of the algorithms. We first selected four iterative algorithms to be used as examples. In doing so, we considered several factors. (i) Since our goal is to measure the iterative code simulation capability of small-sized LMs, each step should be sufficiently simple to be solvable even by such models. (ii) To ensure that the step count is meaningful, the complexity of each step should be approximately uniform. These correspond to conditions (i) and (ii) in Section 3.2. Furthermore, while some prior works (La Malfa et al., 2024; Shojaee et al., 2025) intentionally employed algorithms with superlinear complexity to inputs in order to elicit failures in LMs, we focus here exclusively on algorithms with linear complexity in the input size.

First, we adopted addition, which has been widely used as a case study for LLMs (Zhong et al., 2023; Yu & Ananiadou, 2024). Next, from popular algorithms, we selected dynamic programming algorithms, greedy algorithms, and graph algorithms. After writing out their code and ensuring conciseness, we chose jump game DP, interval scheduling of sorted jobs, and BFS, one for each group. The explanations of these algorithms are provided below.

Extraction and simplification of iterative parts. Since our goal is to measure the step-wise error rate, we need to eliminate preprocessing and postprocessing so that the step count is meaningful. For example, in interval scheduling, we directly provide a list of jobs already sorted by their finishing times for this reason.

We also took into account the difficulty of each operation for LMs and adjusted the implementation and input/output formats to maximize their likelihood of success at each step. For instance, as in Section 4.3, where we introduced additional training on string manipulations, LLMs generally struggle with string operations. Therefore, in the implementation of addition, we pass digits as lists rather than as plain numbers. Similarly, because LMs often struggle with extracting elements from specified positions in a list, in the addition task we remove the used element from the list at each digit step. Further details are provided in the respective algorithm paragraphs.

(a) Digit-wise addition (Listing 14). Digit-wise addition takes two numbers as input and compute their sum. In implementing this task, we considered the following points.

- LMs generally struggle to extract a specific digit from a number, which causes the difficulty to increase as the computation proceeds step by step. To address this, we decompose the numbers into lists of digits and remove each digit once it has been processed, thereby constructing an equivalent algorithm that is easier for LMs to handle.
- Although `a = A[-1]` and `A = A[1:]` can be written more compactly as `a = A.pop()`, we keep them separated for clarity. We also avoided to use `.append` and used `[c] + C` instead.
- At every step we print the intermediate result as well as the lists for original numbers `A` and `B`, which ensures that the necessary information remains close to the output even as the computation progresses further away from the input.
- To ensure that the function body consists only of a `while` loop, we include not only the input numbers A and B to be added, but also the output C and the carry used in the algorithm as part of the inputs.

The inputs are generated by randomly sampling n-digit numbers, where n is specified as a complexity of the problem. For completeness of the algorithm, we include an exception handler for the final carry immediately before the output. However, even if a carry occurs in the last digit, we do not count it as a failure. In n-digit plus n-digit addition, we evaluate steps only up to n, since not every case produces a final carry and including it would make the calculation of the maximum failure rate complicated.

Listing 14 Python program for (a) digit-wise addition

```python
def add_equal_length_numbers(A: list[int], B: list[int], C: list[int],
    carry: int) -> list[int]:
    while A and B:
        a = A[-1]
        b = B[-1]
        c = a + b + carry
        carry = c // 10
        c = c % 10
        A = A[:-1]
        B = B[:-1]
        C = [c] + C
        print(A, B, C, carry)
    if carry:
        C = [carry] + C
    return C
```

(b) Dynamic Programming for the jump game (Listing 15). We first describe the jump game. Given a list of integers `heights`, starting from the leftmost element one repeatedly ”jumps” either one or two positions forward until reaching the right end. At each step, the cost is defined as the absolute difference between the numbers, and the goal is to find a path that minimizes the total cost. This problem can be solved using a standard one-dimensional dynamic programming, where we iteratively update the minimum cost to each position from the left in `DP`.

In implementing this task, we considered the following points.

- To avoid exception handling around the start and end positions, we pre-filled the DP list at positions 0 and 1. This eliminated the need for special cases within the function and ensured that the workload at each step remained uniform. Therefore, a `heights` list of length $n+2$ corresponds to n steps.

- To eliminate the need for the LM to retrieve values from the middle of the sequence, we removed each element from the `heights` list once it was accessed, since the element was no longer needed thereafter.

The inputs are generated by randomly sampling from $\{0, \ldots, 9\}^{n+2}$, where n is specified as a complexity of the problem. This function receives `heights` as well as an empty list `DP`, and fill and output this `DP`.

Listing 15 **Python program for (b) dynamic programming for the jump game**

```python
def jump_DP_easy(heights: list[int], DP: list[int]) -> list[int]:
    while len(heights) >= 3:
        one_step = DP[-1] + abs(heights[2] - heights[1])
        two_steps = DP[-2] + abs(heights[2] - heights[0])
        d = min(one_step, two_steps)
        DP = DP + [d]
        heights = heights[1:]
        print(heights, DP, one_step, two_steps, d)
    return DP
```

(c) Interval scheduling of sorted jobs (Listing 16). Given a list of intervals (jobs) specified by `start` and `end` times, the task is to find the maximum number of non-overlapping intervals. This can be solved by first sorting the intervals by their finishing times and then greedily selecting them in order of earliest finishing time. However, since sorting requires $O(n \log n)$ time, it is not suitable for our setting. Therefore, we isolate only the selection step and use it as the subject of code execution. Here, the output is a bool list, indicating whether each interval is selected or not.

In implementing this task, as before, we removed each job once it was examined so that the LM did not need to access the middle of the list.

Based on the specified complexity n, we generate n jobs. For each job, the start time is sampled from $[0, 2n - 4]$, and the duration (end time minus start time) is sampled from $[1, 4]$. After obtaining the list of intervals, we sort it based on the end time. `last_end` is initialized as -1, and `is_selected` is initialized as an empty list.

Listing 16 **Python program for (c) interval scheduling of sorted jobs**

```python
def interval_scheduling_of_sorted_jobs(jobs: list[tuple[int, int]],
    last_end: int, is_selected: list[bool]) -> list[bool]:
    while jobs:
        start, end = jobs[0]
        jobs = jobs[1:]
        if start >= last_end:
            last_end = end
            is_selected = is_selected + [True]
        else:
            is_selected = is_selected + [False]
        print(jobs, is_selected)
    return is_selected
```

(d) Breadth First Search (Listing 17). This task is to perform a breadth-first search from a specified vertex in a connected unweighted graph. In implementing this task, we considered the following points.

- Instead of representing the graph as a list of edges, we constructed and stored an adjacency list in advance. This eliminates the need for the LM to search for edges at each step.
- The output is defined as the distance from `start_node` to each vertex, which is stored in the form of a dictionary so that correspondence between each vertex and distance is clear. All values of this dict `Distance` are initialized as -1.
- To ensure that the function body consists only of a `while` loop, we initialize the queue outside the function. Specifically, we set `Queue = [start_node]`.

For a given complexity parameter n, the input is an undirected graph with $n+1$ vertices and $2(n+1)$ edges. We first uniformly sample a minimum spanning tree and then add random edges, thereby ensuring that the graph is connected.

This function takes `Graph`, `Distance`, and `Queue` as inputs, fills `Distance`, and returns it. We define a step as the process from discovering one vertex to discovering the next. Since the `start_node` is already discovered, the total number of steps is one less than the number of vertices. This is the reason why, for complexity n, we consider a graph with $n+1$ vertices.

Listing 17 **Python program for (d) breadth first search**

```
def breadth_first_search(
    Graph: dict[int, list[int]],
    Distance: dict[int, int],
    Queue: list[int]
) -> dict[int, int]:
    while Queue:
        current_node = Queue[0]
        Queue = Queue[1:]
        print(Queue, current_node)
        for neighbor in Graph[current_node]:
            if Distance[neighbor] == -1:
                Distance[neighbor] = Distance[current_node] + 1
                Queue = Queue + [neighbor]
            print(Distance, Queue, current_node, neighbor)
    return Distance
```

H.2 EVALUATION

Listing 18 shows the prompts used and an example of inference by Llama3.1-8B-Instruct. These prompts were used for all models except those with ET-CoT and SemCoder (Ding et al., 2024a). For SemCoder, we strictly followed the prompt templates provided at `https://github.com/ARiSE-Lab/SemCoder`.

Listing 18 **Example of the prediction pipeline**

Prompt for predictions:

System prompt:

```
You are an assistant capable of simulating the execution of Python code
    . The user will provide Python code that has print statement(s), and
     your task is to:

1. First, reason through the behavior of the code step by step to
    understand how the internal variables are updated. Enclose this
    reasoning between <reason> and </reason> tags.

2. Then, show the complete simulated output of the code, which consists
     of all printed lines and the returned value. Between <trace> and </
    trace> tags, write the output from each print call in the order they
     are executed, placing each one on its own line. Between <answer>
    and </answer> tags, write the returned value separately, as it would
     appear in the terminal after running the code.

Use the following format:

<reason>

[your reasoning here]</reason><trace>

[print output here]</trace><answer>

[only the returned value here]</answer>
```

User prompt:

```
def add_equal_length_numbers(A: list[int], B: list[int], C: list[int],
    carry: int) -> list[int]:
    while A and B:
        a = A[-1]
        b = B[-1]
        c = a + b + carry
        carry = c // 10
        c = c % 10
        A = A[:-1]
        B = B[:-1]
        C = [c] + C
        print(A, B, C, carry)
    if carry:
        C = [carry] + C
    return C

A = [2, 1, 5]
B = [8, 6, 0]
C = []
carry = 0

f(A, B, C, carry)
```

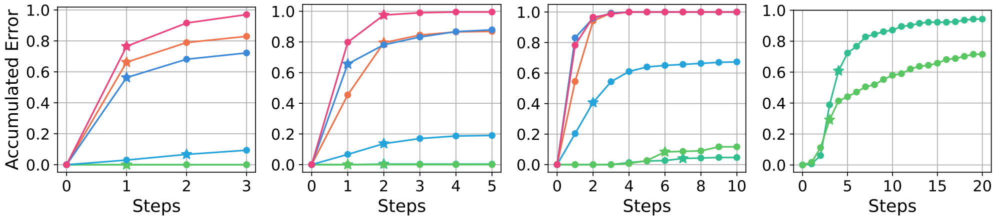
Generated predictions (Llama3.1-8B-Instruct):

```
<reason>

The function `add_equal_length_numbers` performs addition of two
    numbers represented as lists of digits, using a carry variable to
    handle cases where the sum of two digits is greater than 9. Here's a
     step-by-step breakdown of how it works:

1. The function takes four parameters: `A` and `B` which are the lists
    of digits to be added, `C` which is the list to store the result,
    and `carry` which is the carry from the previous addition step.
2. The function enters a while loop that continues as long as both `A`
    and `B` are non-empty.
3. In each iteration of the loop, the last elements of `A` and `B` are
    extracted using `A[-1]` and `B[-1]`. These elements are added
    together along with the current `carry` to get the sum `c`.
4. The carry for the next addition step is calculated as `c // 10`.
    This is done by performing integer division of `c` by 10, which
    gives the quotient and keeps the remainder.
5. The last digit of the sum `c` is calculated as `c % 10`, which gives
     the remainder of dividing `c` by 10. This is the new digit to be
    added to the result list.
6. The last elements of `A` and `B` are removed using `A = A[:-1]` and
    `B = B[:-1]`.
7. The new digit is appended to the result list `C` using `C = [c] + C
    `.
8. If there is a carry after the addition, it is set as the new carry
    for the next step.

The function continues this process until either `A` or `B` is empty.
    If there is a carry left after the last addition, it is used to add
    a new digit to the result.

Finally, the function returns the result list `C` after all additions
    have been performed.

</reason><trace>

[2, 1, 5] [8, 6, 0] [5, 7, 5] 0
[1, 5] [6, 0] [7, 5, 5] 0
[5] [0] [5, 5, 5] 0
```

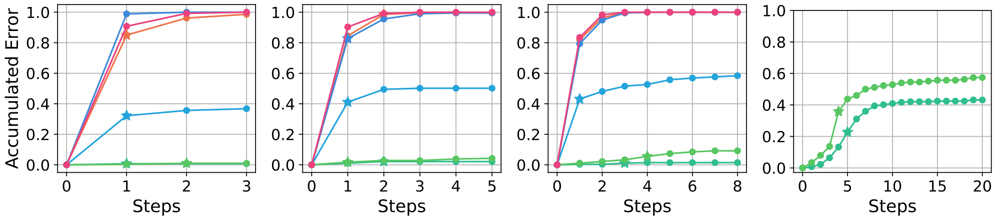
```
[] [] [5, 5, 5] 0
[] [] [5, 5, 5]

</trace><answer>

[5, 5, 5]

</answer>
```

H.3 FULL RESULTS

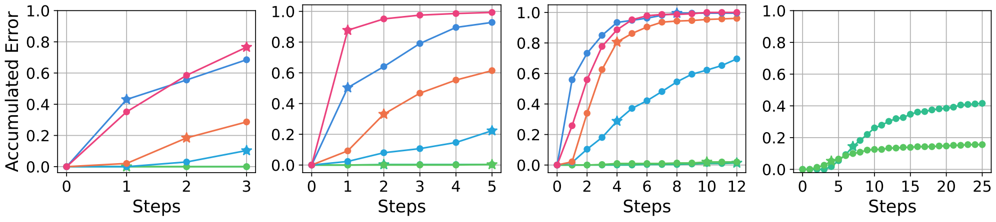
Here we provide the full results for Figures 5 and 7. Figure 8 corresponds to Figure 5, and Figure 9 corresponds to Figure 7, respectively.



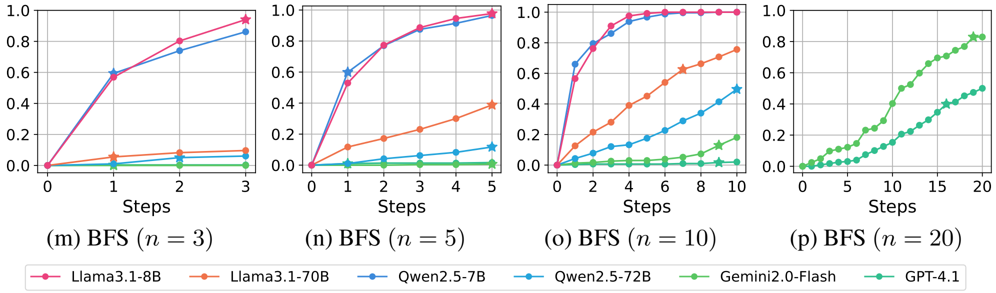
(a) Addition ($n = 3$) (b) Addition ($n = 5$) (c) Addition ($n = 10$) (d) Addition ($n = 20$)



(e) Jump DP ($n = 3$) (f) Jump DP ($n = 5$) (g) Jump DP ($n = 8$) (h) Jump DP ($n = 20$)



(i) Int. sched. ($n = 3$) (j) Int. sched. ($n = 5$) (k) Int. sched. ($n = 12$) (l) Int. sched. ($n = 25$)

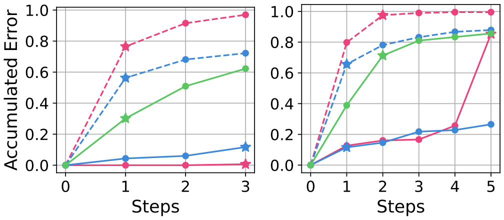


(m) BFS ($n = 3$) (n) BFS ($n = 5$) (o) BFS ($n = 10$) (p) BFS ($n = 20$)

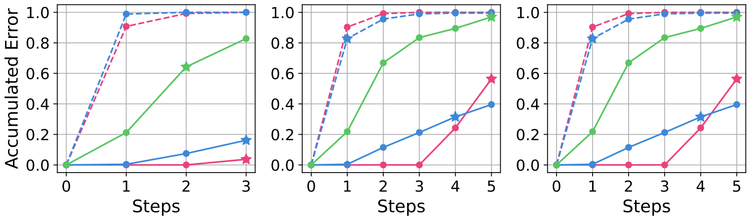
Figure 8: Step-wise cumulative error rate for iterative code simulations (full results).

We now provide an overview of the results in Figure 8. First, Llama3.1-8B and Qwen2.5-7B fail on all tasks for $n = 3, 5$ with high probablity. In contrast, for Llama3.1-70B, Qwen2.5-72B, Gemini2.0-Flash, and GPT-4.1, the following tendency can be observed, especially for $n \geq 5$. In (d) BFS, the cumulative error rate increases gradually, whereas in (a) addition, (b) jump game DP, and (c) interval scheduling, the step-wise failure rate is higher in the early steps. One hypothesis for this task-specific difference is that BFS is graph-based, unlike addition, jump game DP, and interval scheduling. Although we did not complete the experiments with the minimum spanning tree due to eventually adopting BFS, partial results exhibited a similar pattern to BFS, with the error rate increasing gradually. However, this contrast between the two modes is intriguing, and rather than drawing general conclusions from only five algorithms, further dedicated exploration would be more valuable.

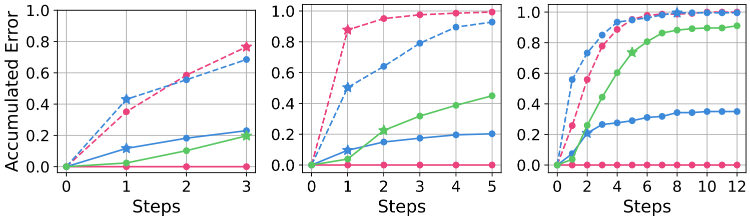
Finally, as for Figure 9, the improvements from ET-CoT over the base models are substantial across all tasks and all complexities for both Llama3.1-8B and Qwen2.5-7B, and they are also more stable



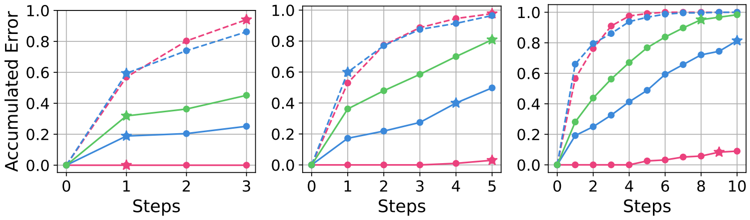
(a) Addition $(n = 3)$ (b) Addition $(n = 5)$



(c) Jump DP $(n = 3)$ (d) Jump DP $(n = 5)$ (e) Jump DP $(n = 8)$



(f) Int. sched. $(n = 3)$ (g) Int. sched. $(n = 5)$ (h) Int. sched. $(n = 12)$



(i) BFS $(n = 3)$ (j) BFS $(n = 5)$ (k) BFS $(n = 10)$

Figure 9: ET-CoT mitigates the initial instability of iterative code simulation (full results).

than SemCoder. Furthermore, in particular, Llama3.1-8B+ET-CoT completely suppresses the initial instability.