CoCoNUT: Structural Code Understanding does not fall out of a tree

Claas Beger
Department of Computer Science
Cornell University
Ithaca, New York
cbb89@cornell.edu

Saikat Dutta
Department of Computer Science
Cornell University
Ithaca, New York
saikatd@cornell.edu

***Abstract*—Large Language Models (LLMs) have shown impressive performance across a wide array of tasks involving both structured and unstructured textual data. More recently, LLMs have demonstrated remarkable abilities to generate code across different programming languages. Recent results on various benchmarks for code generation, repair, or completion suggest that certain models have programming abilities comparable to or even surpass humans. In this work, we demonstrate that the high performance on such benchmarks does not correlate to humans' innate ability to understand the structural control flow of code.**

For this purpose, we extract code solutions from the HumanEval benchmark, which the relevant models perform very strongly on, and trace their execution path using function calls sampled from the respective test set. Using this dataset, we investigate the ability of 7 state-of-the-art LLMs to match the execution trace and find that, despite the model's abilities to generate semantically identical code, they possess only limited ability to trace the execution path, especially for longer traces and specific control structures. We find that even the top-performing model, Gemini 1.5 Pro can only fully correctly generate the trace of 47% of HumanEval tasks.

In addition, we introduce a subset for three key structures not, or only contained to a limited extent in HumanEval: Recursion, Parallel Processing, and Object Oriented Programming principles, including concepts like Inheritance and Polymorphism. Besides OOP, we show that none of the investigated models achieve an average accuracy of over 5% on the relevant traces. Aggregating these specialized parts with the ubiquitous HumanEval tasks, we present the Benchmark CoCoNUT: Code Control Flow for Navigation Understanding and Testing, which measures a models ability to trace the execution of code upon relevant calls, including advanced structural components. We conclude that the current generation LLMs still need to significantly improve to enhance their code reasoning abilities. We hope our dataset can help researchers bridge this gap in the near future.

***Index Terms*—Code Understanding, Large Language Models, Code Execution, Benchmarks**

I. Introduction

Large Language Models (LLMs), such as GPT-4 and Llama, have demonstrated remarkable progress across diverse tasks and are now widely used in applications like code generation (e.g., GitHub Copilot [1]), healthcare [2], and finance [3].

In programming, LLMs show great promise by automating tasks such as generating code, authoring tests, detecting bugs, and repairing programs. However, achieving true success in these tasks requires LLMs to not only generate code but also *reason* about its behavior. To develop such capabilities, robust benchmarks that evaluate code reasoning are essential.

Current benchmarks like HumanEval [4] and MBPP [5] primarily assess code generation for simple tasks and are already saturating, with GPT4 achieving over 90% accuracy [6]. While newer benchmarks include execution reasoning tasks (e.g. CruxEval [7], LiveCodeBench [8]), they do not test the understanding of control flows. Thus, a more structurally focused code reasoning benchmark is needed.

Our Work. To address this, we introduce CoCoNUT– a novel benchmark for evaluating LLMs' ability to trace complex control flows. Each task in CoCoNUT includes a program and test input, requiring the model to generate a trace of line numbers executed by the program. This challenging task necessitates handling long-term dependencies, control structures, and nested expressions, as shown in Figure 1. We argue that tracing execution is a natural test of a model's understanding of code.

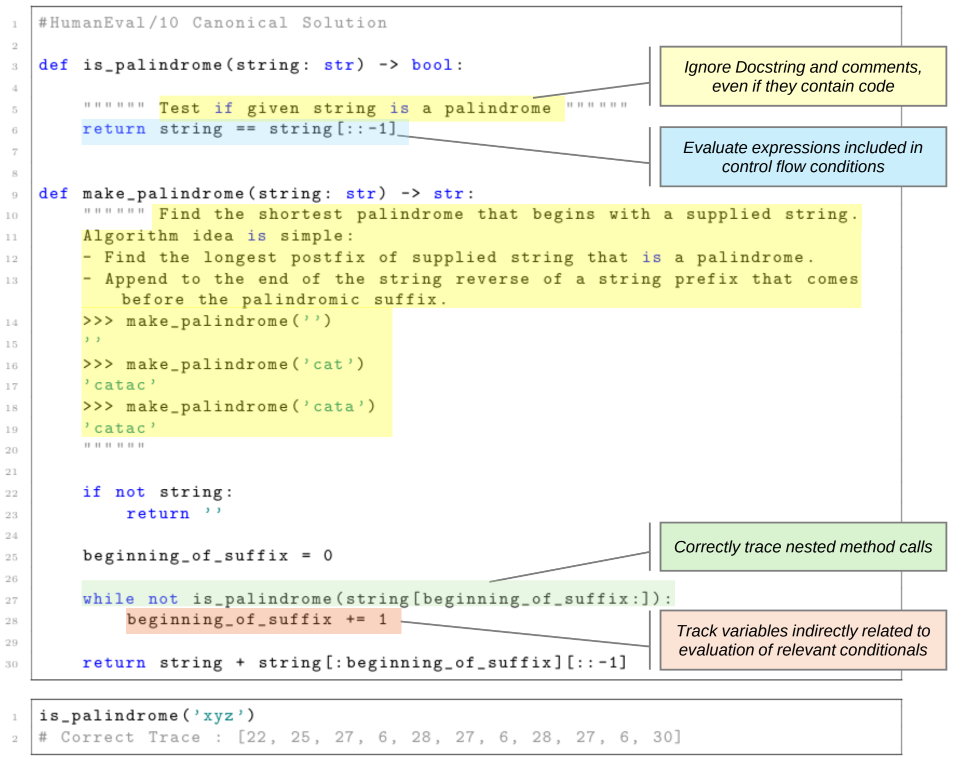
Currently, CoCoNUT includes 161 tasks and 1083 traces derived from HumanEval (*HumanEval-Trace*), and 124 tasks with 620 traces focusing on advanced programming structures such as recursion, parallel processing, and object-oriented principles (*Advanced-Trace*) – totaling 285 tasks and 1703 traces.

Results. We evaluate 7 state-of-the-art LLMs, including proprietary models (GPT-4o, Gemini 1.5 Pro, and Claude 3.5 Sonnet) and open-source models (e.g., LLama3.1, Qwen2.5Coder, Mistral Codestral 22B, and CodeLLama 7B). While LLMs excel at tracing simple programs, their performance on advanced control flows lags far behind their code generation abilities. For instance, the top-performing model, Gemini 1.5 Pro, traces only 47% of HumanEval tasks, and accuracy sharply declines for traces longer than 25 lines. Surprisingly, advanced prompts like Chain-of-Thought offer limited benefits and sometimes degrade the performance on complex tasks. We provide a detailed discussion of results in § IV. Our code and dataset are available here.

II. CoCoNUT Framework

A. Tasks

We utilize the HumanEval dataset proposed by [9], which consists of 164 Python programming tasks and


Ignore Docstring and comments, even if they contain code
Evaluate expressions included in control flow conditions
Correctly trace nested method calls
Track variables indirectly related to evaluation of relevant conditionals

```
1  #HumanEval/10 Canonical Solution
2
3  def is_palindrome(string: str) -> bool:
4
5      """ Test if given string is a palindrome """
6      return string == string[::-1]
7
8
9  def make_palindrome(string: str) -> str:
10     """ Find the shortest palindrome that begins with a supplied string.
11     Algorithm idea is simple:
12     - Find the longest postfix of supplied string that is a palindrome.
13     - Append to the end of the string reverse of a string prefix that comes
             before the palindromic suffix.
14     >>> make_palindrome('')
15     ''
16     >>> make_palindrome('cat')
17     'catac'
18     >>> make_palindrome('cata')
19     'catac'
20     """
21
22     if not string:
23         return ''
24
25     beginning_of_suffix = 0
26
27     while not is_palindrome(string[beginning_of_suffix:]):
28         beginning_of_suffix += 1
29
30     return string + string[:beginning_of_suffix][::-1]
```

```
1  is_palindrome('xyz')
2  # Correct Trace : [22, 25, 27, 6, 28, 27, 6, 28, 27, 6, 30]
```

Fig. 1. Tracing the execution flow requires a diverse understanding of code structures, even for shorter sequences like HumanEval Tasks. We list a couple of examples for Task ID 10.

serves as a suitable benchmark for structural understanding. These tasks are frequently used to evaluate LLMs, ensuring a correlation between control flow understanding and task-solving performance.

To adapt the dataset, we merge the function signature (including docstring examples) with the canonical solution. Using an abstract syntax tree, we parse the testing code to identify all calls to the candidate function and extract their arguments. HumanEval tasks are concise, with function bodies ranging from 1 to 28 lines and an average of 7.7 tests per function, resulting in approximately 1250 potential execution traces. We employ the Python tracer provided by the sys library to analyze execution for each argument set provided by a test call, filtering out sets requiring more than 1024 consecutive tokens, yielding 1083 argument sets across 161 tasks.

Additionally, Liu et al. introduced EvalPlus [10], which expands the number of tests for HumanEval tasks, revealing that model performance was previously overestimated. While we generate an additional dataset based on EvalPlus (over 120,000 traces without trace length filtering), we limit our scope to a preliminary evaluation on the base HumanEval tests.

B. Evaluation Metrics

We employ two main evaluation metrics: *Exact Match* and *Similarity*. Exact Match provides a straightforward measure of whether a model can fully trace the relevant execution. To avoid bias from tasks with varying numbers of tests, we compute task-level accuracy first, then aggregate it as an overall *Accuracy Mean*. Note that this metric does not apply when evaluating traces of different lengths, as trace lengths can vary significantly and are not always tied to code line numbers.

While accuracy is intuitive, it does not capture the diversity of errors. A trace that deviates by only one line is better than a completely incorrect prediction. To address this, we use a *Similarity* metric based on the Gestalt-pattern matching algorithm [11]. This metric focuses on matching contiguous subsequences, emphasizing longer matches over distributed ones. The similarity between two sequences A and B is computed as:

$$S(A, B) = \frac{2 \cdot M}{|A| + |B|} \tag{1}$$

where M is the total number of matching characters in contiguous blocks, and $|A|$ and $|B|$ are the lengths of the sequences. The metric ranges from 0 to 1 and complements the accuracy by quantifying partial matches. For incorrectly generated traces, we also report *False Similarity*. Further, we distinguish the hard accuracy (**Acc Hard**), as the fraction of tasks that the models solve completely, meaning they are able to reproduce the trace of all given tests.

For concurrency, traditional metrics are insufficient due to variations in execution order across threads. We isolate concurrent segments, compare overlapping indices, and sort the data before applying the similarity metric. While lenient, this approach requires models to correctly identify the start and end of concurrent execution as a prerequisite, which we consider a valid trade-off.

III. Methodology

A. Research Questions

In our work, we are particularly interested in the following research questions:

- **RQ1:** How does the understanding of execution semantics relate to code generation abilities?
- **RQ2:** What is the impact of different prompting techniques on the overall performance?
- **RQ3:** How well can current state-of-the-art language models understand the execution semantics of programs? How does this ability change with the increasing complexity of programs, as measured by length or by introducing more advanced programming concepts?

B. Dataset Curation

As described earlier, we collect programs from multiple sources, including HumanEval. For the advanced topics, we extract LeetCode programs based on tags from multiple open-source repositories ([12], [13]). Further, we complement this data by collecting programs from the multi-language programming platform RosettaCode [14], which also provides relevant topic tags. We extract the actual programs from a related open-source repository [15]. Unfortunately, these programs often do not have corresponding tests or relevant execution logic. We employ GPT4o to generate simple base test cases which we expand manually. Generally, we create five tests for every sample and try to employ increasing difficulty (regarding number of calls, trace length, complex control flows and similar aspects), which is necessary to invoke the topical functionality. Since the advanced programs generally feature multiple components (methods, classes, etc.), we create a main code block for the models to trace, rather than a single method call. For the topic of concurrency, as briefly mentioned in the previous section, we need to make further adjustments to enable valid tracing. In our prompt, we describe how the model may use parentheses to mark concurrent execution, We further demonstrate it in the given prompt and tolerate smaller deviations like multiple parentheses.

We find that models initially struggle to find the correct format, so we employ one-shot prompting, giving the model a very simple code call with the corresponding execution trace to demonstrate the correct format. We explicitly ask the model to refrain from producing other output in order to produce a meaningful comparison with Chain-of-Thought prompting. We observe that especially the smaller LLama model, but also some of the larger models deviate from this instruction sometimes, which we score as a zero in both metrics. In the given prompt, we merge the function signature and docstring with the canonical solution and annotate the lines with an index number to alleviate potential ambiguity through newlines or comments. Separate from the given code, we name the called function and the argument list. If necessary, we provide superficial cleaning of the predictions (removing whitespaces, markdown, or Python annotations). If not specified otherwise, we use greedy decoding and a maximum token length of 1024 for direct prompting, as well as 4096 for CoT. We elect to increase the token count in this manner to avoid cutting off the generation before the final trace is reached.

IV. Results

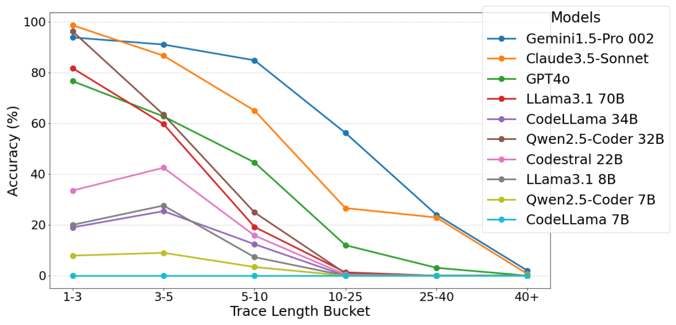


Fig. 2. Model Accuracy by Buckets of Trace Length using Direct Prompting.

A. Performance of SOTA LLMs on CoCoNUT

We query three popular large-scale language models that perform at the top of the HumanEval and EvalPlus benchmarks respectively. Namely, we use Claude 3.5 - Sonnet, Gemini Pro 1.5 002, and GPT4o. Since there is relevant work describing that Chain-of-Thought improves execution tracing [16], [17] (although different from our definition of trace), we specifically look at the performance of direct and Chain-of-Thought prompting. We present the aggregated results of our experiments on HumanEval in Table I. Across all tables, we refer to Similarity as **Sim** and Accuracy as **Acc**.

The results show that none of the models are able to match their performance on HumanEval. Further, we observe two interesting points in that the performance comparison between the models does not reflect the ranking on HumanEval, on which GPT4o significantly outranks the other two large models. Upon closer investigation, GPT4o strongly struggles with hallucinations, where function signatures are called directly (which is specified to be incorrect in the prompt), and code lines in the docstrings or unrelated code indices are listed. Further, there is a large difference between the average accuracy over all traces and the number of execution cases that the models are able to fully solve.

B. Correlation on Generative vs Execution Tracing Tasks

We conduct a small experiment to assess the correlation between the performance in solving the described HumanEval coding problem, as measured by the given test suite, and the generation of execution traces for the provided sample solution. For this experiment, we utilize LLama 3.1 70B, which demonstrated strong performance on HumanEval tasks.

TABLE I
PERFORMANCE OVERVIEW ON THE HUMANEVAL-TRACE

Task ID	**CoT**				**Direct**			
	Acc Hard (%)	**Acc Mean (%)**	**Sim**	**False Sim**	**Acc Hard (%)**	**Acc Mean (%)**	**Sim**	**False Sim**
Gemini1.5-Pro 002	**47.2**	**66.2**	**0.88**	0.37	**47.0**	**65.7**	**0.89**	0.37
Claude3.5-Sonnet	41.0	61.6	0.87	0.43	41.0	58.7	0.88	0.44
GPT4o	16.8	39.4	0.75	0.50	21.2	38.8	0.75	0.50
LLama3.1 70B	16.2	38.1	0.76	0.52	25.5	36.0	0.71	0.42
CodeLLama 34B	1.2	7.6	0.46	0.43	2.5	10.0	0.57	0.52
Qwen2.5-Coder 32B	26.1	44.3	0.81	0.50	32.7	42.4	0.78	0.44
Codestral 22B	9.3	25.0	0.71	**0.57**	3.1	17.8	0.66	**0.59**
LLama3.1 8B	1.9	12.6	0.56	0.51	0.6	10.4	0.53	0.48
Qwen2.5-Coder 7B	1.9	11.0	0.61	0.56	0.0	4.1	0.56	0.55
CodeLLama 7B	0.0	0.1	0.28	0.28	0.0	0.0	0.41	0.41

We intentionally refrain from using a code-specific model, as its training, potentially influenced by HumanEval, might lead to an overestimation of performance on the code generation task. Applying this model on HumanEval results in a pass rate of 80.4% using base tests and 74.8% using evalplus. While we include evalplus tests for completeness, we emphasize that the comparison using base tests is more significant, as these are the tests with which we reproduce the code execution.

To generate a meaningful difficulty ranking, we order the generative results by the number of tests failed by each solution. Similarly, we rank the execution tracing results based on the average solution similarity per task. We compute Spearman's rank correlation for these orderings, yielding values of -0.05/-0.09 (base and evalplus) for the Chain-of-Thought approach and 0.06/0.01 for the direct approach. Both pairs of figures suggest no significant correlation between the difficulty of code generation and execution tracing. Additionally, we compute the relative overlap of failed tasks between the two approaches. Approximately 20% of the tasks that failed in execution tracing also failed during code generation for Chain-of-Thought prompting, and 17% for direct prompting. These findings support our previous assumption that the difficulty in execution tracing primarily arises from the trace length, but also from the inclusion of specific code constructs that are challenging for the model to execute correctly. The latter aspect will be further discussed in subsection V-A. However, it does not appear that there is a significant overlap with the difficulty sources of Code Generation.

Answer to RQ1*: Thus, we conclude that there is only a very weak intermediate connection between the model performance on Code Generation and Execution Tracing, which does not apply to task-specific difficulty.*

C. *Direct vs CoT Prompting over Trace Lengths*

Besides the performance metrics, we set out to compare direct and CoT prompting. Our findings suggest that the benefits of CoT may not be as significant for larger models. While Codestral and and the smaller versions of LLama and QwenCoder experience a 3x increase in hard accuracy, Gemini and Claude stay consistent and GPT4o, as well as larger

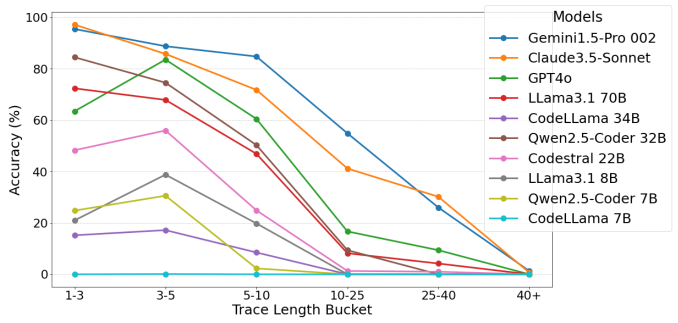


Fig. 3. Model Accuracy by Buckets of Trace Length using Chain-of-Thought Prompting.

versions of the two previously named models even decreases. Besides the general performance, we were interested in finding how the complexity of the execution, as measured by the length of the code or trace, impacts the performance. Although we find that the length of the execution trace and the code length have a similar effect on the task performance, we generally focus on trace length for evaluation purposes. We distribute the trace lengths in 6 buckets and aggregate performance within them. Our results are shown in Figure 2 and Figure 3.

The stronger models generally start to see a large performance drop from a trace length of 25 onwards. Similarly, medium sized models (70-22B) have a cutoff of around 10 and the smaller models already struggle with lengths beyond 5. This supports the hypothesis that the ability to trace long code executions emerges at scale. Further, after a trace length of 40, models are generally unable to generate a meaningful trace in the vast majority of cases. For larger traces, some of the models also lost the correct execution path and went into endless loops, which were truncated according to the token limit. We would further like to address the performance of the different code-specific models we included. While some models, like the larger version of QwenCoder seem to have improved capabilities even with smaller parameter counts, others like CodeLLama or CodeStral just perform in line with their parameter count. We hypothesize that this is due to the fact that different models had varying degree of access to relevant execution data during their training period, despite the focus on code. In addition, we observed some errors with instruction-following in some of them.

We also consider the performance using CoT prompting and again observe the most significant impact for smaller models, even though larger models also improve slightly. Notably, Chain-Of-Thought mostly improves the existing capabilities, and models generally tend to lose functionality at the same threshold as direct prompting. This is somewhat surprising since existing works on investigating traces with a focus on variable states tend to claim that CoT naturally combats long-dependence issues through verbose self-documentation [18], similar to alternative approaches like Tree-of-Thought [19] or employment of other means for the model to document intermediate steps [20].

Answer to RQ2: *Using our current results, we cannot clearly identify a significantly positive effect of CoT on execution tracing. With regard to our second research question, we conclude that there is some importance to the chosen prompting technique, but the effect varies according to model size and the type of program that is supposed to be traced.*

D. Advanced Structural Topics

Besides HumanEval, CoCoNUT employs evaluation on different advanced aspects. We show an overview of the corresponding performance in Table II. Apart from a few models on the task of Object-Oriented Programming Principles, none of the models can provide performance beyond 5% task accuracy. Further, it is apparent that Chain-Of-Thought only improves performance slightly on Recursion, but not on the other topics. We note that the corresponding programs are generally longer in code, as well as execution traces. However, this is still a very controlled setting since all relevant methods are in a single file and none of the code samples are longer than 200 lines of code.

We believe that this effectively shows how even SOTA models struggle with fully reproducing the execution calls involving the application of advanced code concepts. More explicitly, we observe that models cannot handle resolving recursion after a certain depth and regularly produce infinite loops. For Concurrency, the models struggle with correctly identifying the start and end of parallel execution, as well as the nested worker calls. For OOP, models generally struggle with navigating the longer code segments needed to set up the necessary class structure and correct call resolution.

Answer to RQ3: *We find that state-of-the-art LLMs, despite strong performance on generative tasks, struggle with structural tracing, especially on code featuring diverse sources of complexity. In general, larger models appear to feature a basic understanding of simple control flow structures but traces beyond the length of 40, as well as advanced structural code concepts, remain a significant challenge for all models to reason about.*

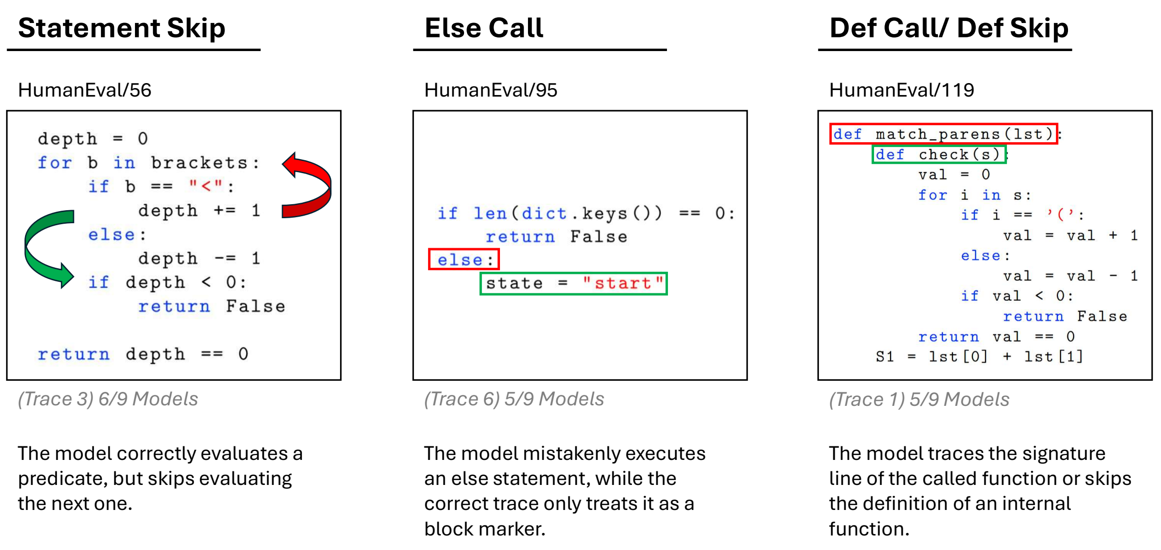
TABLE II
PERFORMANCE ON A SUBSET OF DIFFERENT ADVANCED PROGRAMMING CONCEPTS SOURCED FROM OPEN SOURCE PROJECTS AND COMPETITIONS. EVERY PROGRAM WAS TRACED WITH 5 SAMPLE BLOCKS OF INCREASING DIFFICULTY.

Task ID	**CoT**		**Direct**	
	Acc Mean (%)	**Sim**	**Acc Mean (%)**	**Sim**
Object-Oriented (40 Programs 200 Traces)				
Gemini1.5-Pro 002	14.0	0.79	**20.0**	**0.81**
Claude3.5-Sonnet	0.0	0.77	1.0	0.69
GPT4o	4.5	**0.82**	4.0	0.73
LLama3.1 70B	**15.0**	0.74	10.0	0.75
CodeLLama 34B	0.0	0.37	0.0	0.37
Qwen2.5-Coder 32B	14.5	0.78	4.0	0.73
Codestral 22B	1.5	0.62	1.5	0.6
LLama3.1 8B	0.5	0.58	1.0	0.48
Qwen2.5-Coder 7B	0.0	0.58	0.0	0.56
CodeLLama 7B	0.0	0.30	0.0	0.40
Recursion (66 Programs 330 Traces)				
Gemini1.5-Pro 002	2.7	0.47	0.9	**0.41**
Claude3.5-Sonnet	0.3	0.42	1.2	**0.41**
GPT4o	2.7	**0.49**	**1.8**	0.38
LLama3.1 70B	1.2	0.36	0.6	0.27
CodeLLama 34B	0.0	0.29	0.0	0.27
Qwen2.5-Coder 32B	**3.0**	0.35	**1.8**	0.30
Codestral 22B	1.0	0.29	0.0	0.29
LLama3.1 8B	0.3	0.21	0.0	0.35
Qwen2.5-Coder 7B	0.0	0.15	0.0	0.16
CodeLLama 7B	0.0	0.23	0.0	0.26
Concurrency (20 Programs 100 Traces)				
Gemini1.5-Pro 002	**1.0**	**0.41**	**1.0**	0.39
Claude3.5-Sonnet	0.0	0.4	0.0	**0.42**
GPT4o	0.0	0.39	0.0	0.4
LLama3.1 70B	**1.0**	0.34	**1.0**	0.33
CodeLLama 34B	0.0	0.24	0.0	0.27
Qwen2.5-Coder 32B	0.0	0.36	0.0	0.37
Codestral 22B	0.0	0.29	0.0	0.38
LLama3.1 8B	0.0	0.28	0.0	0.26
Qwen2.5-Coder 7B	0.0	0.21	0.0	0.18
CodeLLama 7B	0.0	0.23	0.0	0.28

V. DISCUSSION

A. Qualitative Error Analysis

In order to better understand the types of errors produced by models when evaluating program calls, we conducted a qualitative study by manually reviewing a set of ten randomly selected HumanEval traces for each of the nine models. Among these, 23% of the traces were identified correctly. We observed several common error patterns across all models, which we outline in Figure 4. Generally, models struggled with trace-specific issues, often arising from particular statements, such as ‘else‘ statements, predefined functions, and similar constructs that serve as block markers but are not executed by the program. These cases accounted for 20% of the 90 analyzed traces. Notably, errors were distributed equally across different model sizes, with the exception of the definition call, which was explicitly described in the prompt. Simpler mistakes, such as skipping statements or conditions, accounted for 16% and 8% of the cases, respectively. Statement skips,



```
depth = 0
for b in brackets:
    if b == "<":
        depth += 1
    else:
        depth -= 1
    if depth < 0:
        return False

return depth == 0
```

The model correctly evaluates a predicate, but skips evaluating the next one.

```
if len(dict.keys()) == 0:
    return False
else:
    state = "start"
```

The model mistakenly executes an else statement, while the correct trace only treats it as a block marker.

```
def match_parens(lst):
    def check(s):
        val = 0
        for i in s:
            if i == '(':
                val = val + 1
            else:
                val = val - 1
            if val < 0:
                return False
        return val == 0
    S1 = lst[0] + lst[1]
```

The model traces the signature line of the called function or skips the definition of an internal function.

Fig. 4. Examples of three of the most common error types encountered in HumanEval tracing. The subtitle denotes the number of models that exhibited this error (as the first deviation) in a given Trace.

as demonstrated in Figure 4, were particularly frequent in the given HumanEval problem, during which models failed to execute the depth check before advancing to the next loop iteration. This issue occurred predominantly when the depth was greater than or equal to zero, possibly influencing the model to skip the check. Condition skips, where the body of a conditional statement was executed without evaluating the predicate, seem to result from similar model behavior. A related error type, which we classified as ”loop skip,” accounted for 4% of the cases. This error occurs when a model exits a loop prematurely.

We also identified a group of error types more common in smaller, less proficient models. These included the “nonsense” error (12%), which refers to accessing lines that are either empty or contain only comments, the “empty” error (9%), where the model failed to produce a valid list, and the “predicate wrong” error (6%), where an incorrectly evaluated predicate led to an incorrect control flow jump. Lastly, the “no exit” error occurred when the model produced an unclosed list. Some models, such as CodeLLama, struggled with format expectations, often producing empty or non-sensical outputs.

We further evaluated five random traces for each of the advanced topics using the best-performing model, Gemini 1.5-Pro. Surprisingly, the most frequent errors encountered were not specific to the corresponding topic. A majority of the traces involved definition call and skip errors, which were more prominent in this context compared to HumanEval, likely due to the multiple functions present in the programs. One trace, which focused on object-oriented principles, exhibited an error in resolving a nested expression, where the inner object initialization was simply skipped. Similarly, a concurrency trace displayed an error related to the marking of concurrency. We note that such errors likely also occur in the other traces, but they were not the first ones, which we limited our analysis to.

In general, the advanced nature of concept-based programs makes it more challenging for models to accurately represent program execution. This is due to the increased length and complexity of the traces, as well as the nested structure of the programs, which include multiple functions or class definitions. When considering realistic software suites, which contain exponentially more lines of code and significantly more complex control flows between different files and packages, it is clear that current state-of-the-art models are still far from being able to fully capture the functionalities of realistic software systems.

B. Alternative Prompts and Settings

To validate our current elicitation approach, we experiment with alternative prompt techniques and line annotations. We apply these techniques on a small subset of one test case for every task ID, resulting in 163 samples. The results are shown in Table III. We show CoT as our baseline and first experiment with static analysis. A lot of the programs contain output or intermediate variables that do not have an influence on the control flow. We ask the model to seek out the variables that do have an influence and disregard the other ones. In addition, we ask the model to try and determine the number of loop iterations before entering and simply reproducing the steps the extracted number of times rather than actually executing it. We found that the model only applies limited analysis, sometimes even ignoring the instruction. Correspondingly, we observe a small performance decrease. We also experiment with increasing the number of samples given in the prompt. In particular, we add one code sample detailing a recursive program execution, and one more that contains nested method calls. We observe no significant performance increase, showing that the model generally seems to possess knowledge of the demonstrated execution concepts, only requiring one prompt to learn the correct output format. Apart from the prompt, we test alternatives to simply annotating the line numbers as comments. Since LLMs are known to struggle with numerical representations, we use a combination of singular letters and symbols, as well as the word **Line**, prior to the index. Both of these approaches reduced the performance, indicating that line indices are a sufficient annotation choice.

As denoted in earlier sections, we generally employ Greedy-decoding, as we assume it to be more applicable to a ground-truth task such as execution tracing. However, to validate these assumptions, we regenerate the samples of the best model, Gemini, using an increased temperature of 1.0. Throughout the CoT and Direct prompting, we observe a performance decrease of about 3 percent average accuracy for Chain-of-Thought and a 10 percent decrease for direct prompting.

TABLE III
COMPARISON OF DIFFERENT PROMPTING TECHNIQUES USING GEMINI1.5-PRO 002 ON A HUMANEVAL SUBSET

Prompting Technique	Average Acc	Average Sim
CoT	**0.6522**	0.8914
Static Analysis	0.6149	0.8843
3-Shot	0.6522	**0.8967**
Symbols	0.0311	0.7787
Line	0.5776	0.8780

VI. Related Work

Coding Benchmarks for LLMs. Numerous coding benchmarks have been proposed for LLMs in recent years. We summarize and compare a few popular ones here. Open AI HumanEval and MBPP [5] are the most popular code generation benchmarks. Several other benchmarks such as CruxEval [7], LiveCodeBench [8], and Codemind [21] include execution reasoning tasks (in addition to code generation) such as predicting the output of a code snippet given some input. CodeXGlue [22] is another prior dataset consisting of 10 different coding tasks. However, it does not include any execution-related tasks.

Runtime Reasoning REval [23] is the most recent work that proposed four execution-related tasks: coverage prediction, program state prediction, execution path prediction (prediction next statement), and output prediction. However, these tasks do not require the LLMs to reason about control flow or different programming structures like the tasks in CoCoNUT.

Ma et al. [24] evaluate LLMs on many different tasks, including generating Abstract Syntax Trees, Control Flow, and Call Graphs. However, they limit their investigation of the dynamic behavior of the execution to Equivalent Mutant Detection and Flaky Test Reasoning, which does not directly concern structural understanding abilities.

Hooda et al. [25] show that LLMs are vulnerable to different mutations related to control and data flow, as well as type and identifier assignment. A strong understanding of the full execution trace would help build resilience against such approaches. To the best of our knowledge, there is no prior work on evaluating the execution tracing abilities of LLMs, which we investigate in this work.

Training LLMs for better execution reasoning. Few recent approaches focused on improving the execution reasoning of LLMs. For instance, Ding et al. developed a new coding dataset augmented with tests and execution traces and trained an LLM, called SemCoder [26]. They showed that such a training strategy elicits better code generation and execution reasoning from the LLM. We also tested SemCoder on CoCoNUT, however, the model was not able to correctly follow the natural language instructions, resulting in no meaningful results. Ni et al. [16] showed that fine-tuning LLMs on Chain-of-Thought reasoning over execution trace improved the performance of PaLM on the HumanEval and the MBPP benchmarks [5]. However, their approach to tracing mostly consists of variable states for straight-line code, which they insert into the source code as comments instead of control flow reasoning. While they demonstrate that this approach also works without inserting the trace, they note that the model exhibits hallucination issues while adapting the trace into natural language reasoning steps. This naturally motivates enhancing language models' abilities to directly extract execution representations.

VII. Conclusion and Future Work

In this paper, we have introduced CoCoNUT, a benchmark for evaluating a model's capability to trace the control flow of a program given relevant input. Our evaluation of state-of-the-art models reveals significant gaps in their performance compared to generative tasks. We observe a strong correlation between trace length and performance, with models generally struggling to trace executions longer than 10 lines—a capability that appears to emerge only at larger scales. Notably, specific training on code does not consistently outperform increases in parameter size. Models uniformly fail on advanced structural programs with longer base traces, highlighting their limitations in predicting the full execution of complex code despite their widespread use in software engineering.

We also investigated the effects of prompting techniques. Chain-of-Thought (CoT) prompting generally improves performance, particularly for smaller models, but offers only marginal benefits for larger ones and sometimes propagates errors, leading models to incorrectly pursue alternative traces. While we experimented with more sophisticated prompt engineering on a subset of HumanEval, we did not find significant improvements over CoT.

To explore the connection between Code Generation and Execution Tracing, we used HumanEval as a proxy. Despite strong performance in code generation, tracing HumanEval solutions yielded significantly lower results, with performance only slightly higher than half the corresponding pass rate on generation. Additionally, we found no significant correlation between task and trace difficulty for the two approaches. This indicates that models lack the human-like ability to trace code they generate, raising concerns about their role in software development.

Next Steps and Future Work. An interesting next step, which we leave to future work, would be to conduct fine-tuning on our dataset, which could improve performance on adjacent tasks such as error localization or code summarization. This approach may help models differentiate between the functionality of code components and their natural language token equivalents, thereby enhancing their ability to reason about code behavior more effectively. We intentionally adapted code snippets from sources that provide equivalent code in multiple programming languages, as we aim to expand our benchmark across them. We believe that structural code understanding is the most significant if it is achieved across several different languages since this implies nuanced understanding rather than pure knowledge of when to use certain tools or commands in one specific language. We also publish our relevant repository to enable the testing of a wider array of models, as well as potential training or fine-tuning on the execution trace data.

References

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. W. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, I. Babuschkin, S. A. Balaji, S. Jain, A. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *ArXiv*, vol. abs/2107.03374, 2021.

[2] J. Clusmann, F. R. Kolbinger, H. S. Muti, Z. I. Carrero, J.-N. Eckardt, N. G. Laleh, C. M. L. Löffler, S.-C. Schwarzkopf, M. Unger, G. P. Veldhuizen *et al.*, "The future landscape of large language models in medicine," *Communications medicine*, vol. 3, no. 1, p. 141, 2023.

[3] "Introducing bloomberggpt, bloomberg's 50-billion parameter large language model, purpose-built from scratch for finance," https://www.bloomberg.com/company/press/bloomberggpt-50-billion-parameter-llm-tuned-finance.

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

[5] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, "Program synthesis with large language models," 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[6] "EvalPlus Leaderboard — evalplus.github.io," https://evalplus.github.io/leaderboard.html, [Accessed 19-11-2024].

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

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

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[10] J. Liu, C. S. Xia, Y. Wang, and L. ZHANG, "Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=1qvx610Cu7

[11] J. W. Ratcliff, D. E. Metzener *et al.*, "Pattern matching: The gestalt approach," *Dr. Dobb's Journal*, vol. 13, no. 7, p. 46, 1988.

[12] P.-Y. Chen, "LeetCode Solutions — walkccc.me," https://walkccc.me/LeetCode/, [Accessed 19-11-2024].

[13] "GitHub - neetcode-gh/leetcode: Leetcode solutions — github.com," https://github.com/neetcode-gh/leetcode, [Accessed 12-11-2024].

[14] "Rosetta Code — rosettacode.org," https://rosettacode.org/wiki/Rosetta_Code, 2007, [Accessed 26-01-2025].

[15] "RosettaCodeData/Task at main · acmeism/RosettaCodeData — github.com," https://github.com/acmeism/RosettaCodeData/tree/main/Task, [Accessed 12-11-2024].

[16] A. Ni, M. Allamanis, A. Cohan, Y. Deng, K. Shi, C. Sutton, and P. Yin, "Next: Teaching large language models to reason about code execution," 2024. [Online]. Available: https://arxiv.org/abs/2404.14662

[17] C. Lyu, L. Yan, R. Xing, W. Li, Y. Samih, T. Ji, and L. Wang, "Large language models as code executors: An exploratory study," 2024. [Online]. Available: https://arxiv.org/abs/2410.06667

[18] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023. [Online]. Available: https://arxiv.org/abs/2201.11903

[19] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," 2023. [Online]. Available: https://arxiv.org/abs/2305.10601

[20] M. Nye, A. J. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan, C. Sutton, and A. Odena, "Show your work: Scratchpads for intermediate computation with language models," 2021. [Online]. Available: https://arxiv.org/abs/2112.00114

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

[22] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[23] J. Chen, Z. Pan, X. Hu, Z. Li, G. Li, and X. Xia, "Reasoning runtime behavior of a program with llm: How far are we?" in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 140–152.

[24] W. Ma, S. Liu, Z. Lin, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, L. Li, and Y. Liu, "Lms: Understanding code syntax and semantics for code analysis," 2024. [Online]. Available: https://arxiv.org/abs/2305.12138

[25] A. Hooda, M. Christodorescu, M. Allamanis, A. Wilson, K. Fawaz, and S. Jha, "Do large code models understand programming concepts? a black-box approach," 2024. [Online]. Available: https://arxiv.org/abs/2402.05980

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

Appendix

A. Trace Length Buckets

In the following, we display the measured accuracies by trace-lengths per the outlined buckets.

TABLE IV
Model Accuracy by Buckets of Trace Length using Direct Prompting.

Model	**1-3**	**3-5**	**5-10**	**10-25**	**25-40**	**40+**
Gemini1.5-Pro 002	93.8%	**91.0%**	**84.8%**	**56.2%**	**24.0%**	**2.0%**
Claude3.5-Sonnet	**98.6%**	86.6%	65.0%	26.6%	22.9%	0.7%
GPT4o	76.6%	62.7%	44.6%	12.0%	3.1%	0.0%
LLama3.1 70B	81.7%	59.7%	19.2%	1.3%	0.0%	0.0%
CodeLLama 34B	19.0%	25.4%	12.4%	0.0%	0.0%	0.0%
Qwen2.5-Coder 32B	96.2%	63.4%	24.9%	0.9%	0.0%	0.0%
Codestral 22B	33.5%	42.5%	15.8%	0.4%	0.0%	0.0%
LLama3.1 8B	20.0%	27.6%	7.3%	0.0%	0.0%	0.0%
Qwen2.5-Coder 7B	7.9%	9.0%	3.4%	0.0%	0.0%	0.0%
CodeLLama 7B	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

TABLE V
Model Accuracy Across Trace Length Buckets using Chain-of-Thought prompting.

Model	**1-3**	**3-5**	**5-10**	**10-25**	**25-40**	**40+**
Gemini1.5-Pro 002	95.5%	**88.8%**	**84.8%**	**54.9%**	**26.0%**	**1.3%**
Claude3.5-Sonnet	**97.2%**	85.8%	71.8%	41.2%	30.2%	0.7%
GPT4o	63.5%	83.6%	60.5%	16.7%	9.4%	0.0%
LLama3.1 70B	72.4%	67.9%	46.9%	8.2%	4.2%	0.0%
CodeLLama 34B	15.2%	17.2%	8.5%	0.0%	0.0%	0.0%
Qwen2.5-Coder 32B	84.5%	74.6%	50.3%	9.4%	0.0%	0.0%
Codestral 22B	48.3%	56.0%	24.9%	1.3%	1.0%	0.0%
LLama3.1 8B	21.0%	38.8%	19.8%	0.0%	0.0%	0.0%
Qwen2.5-Coder 7B	24.8%	30.6%	2.3%	0.0%	0.0%	0.0%
CodeLLama 7B	0.0%	0.1%	0.0%	0.0%	0.0%	0.0%

B. HumanEval Solution Line Length

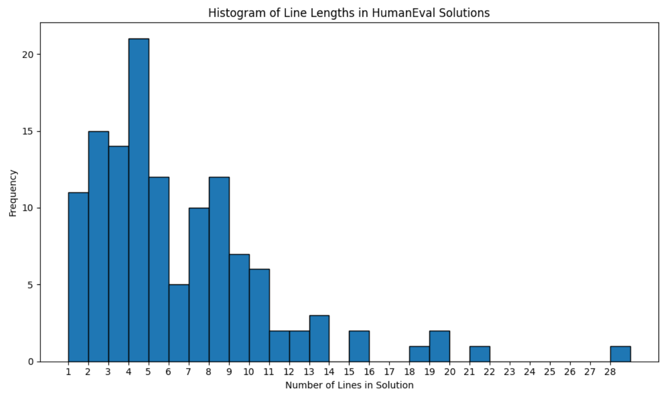


Fig. 5. Length distribution across HumanEval solutions.

C. Prompts

We include a basic prompt for the CoT approach on the HumanEval Trace task in the following (note that the contained code usually contains appropriate newlines and indentation):

This task will evaluate your ability to appreciate the control flow of code with a given input. In the following, I will give you the source code of a program written in Python. The program may feature different functions, which may call each other. To make the task more accessible to you, I have annotated the lines with their index as comments (those begin with a #). The following is very important! Please note that the function signatures are generally not called, instead you should start with the first line of the function. This does not apply to the function call, of course. In addition to the function, I will give you an initial input and the called function. It is your task to return the called lines, in order, as a list. I will give you an example:

```python
def simple_loop(x):  #1
    for i in range(3):  #2
        print(i + x)  #3
    return i  #4
```

Input: (5)
Correct solution: [2,3,2,3,2,3,2,4]
Now I will give you your task. Here is the source code: [code]
Here is the called function: [function]
Here is the input to the function [input]
Please produce the python list containing the executed line numbers in order now. Remember not to include the function signature lines. Think about the solution step-by-step, going through execution steps one at a time. Finally, print the solution as a list of executed steps.

The prompts for the advanced topics are adjusted to account for the difference in format. The prompts for concurrency also feature an adapted code example.