Can Large Language Models Reason About Code?

Changshu Liu
Department of Computer Science
University of Illinois Urbana-Champaign
Illinois, USA
cl144@illinois.edu

Shizhuo Zhang
Department of Computer Science
University of Illinois Urbana-Champaign
Illinois, USA
shizhuo2@illinois.edu

Ali Reza Ibrahimzada
Department of Computer Science
University of Illinois Urbana-Champaign
Illinois, USA
alirezai@illinois.edu

Reyhaneh Jabbarvand
Department of Computer Science
University of Illinois Urbana-Champaign
Illinois, USA
reyhaneh@illinois.edu

arXiv:2402.09664v4 [cs.SE] 3 Apr 2024

***Abstract*—Large Language Models (LLMs) have been widely used to automate programming tasks. Their capabilities have been evaluated by assessing code quality through test execution. However, as we will show, success in code synthesis does not imply code reasoning, which is essential to trust LLMs with tasks that involve program analysis, e.g., test generation and debugging. This paper introduces CodeMind, a framework designed to gauge the code reasoning abilities of LLMs through several inductive reasoning tasks. CodeMind currently supports three tasks: *Independent Execution Reasoning (IER)*, *Dependent Execution Reasoning (DER)*, and *Specification Reasoning (SR)*. The first two evaluate models to predict the execution output of an arbitrary code or code the model could correctly synthesize. The third one evaluates LLMs' abilities to implement the specified expected behavior. Our extensive evaluation of ten LLMs across five benchmarks in two different programming languages for two code generation tasks (code synthesis and translation) using CodeMind shows that LLMs, to some degree, can explain the program execution flow, *specifically for simple programs and the ones they can correctly generate*. However, their performance drops for code with higher complexity, non-trivial logical and arithmetic operators, non-primitive types, and API calls. We observe that, while correlated, code generation abilities do not imply code reasoning: ranking LLMs based on test passing can be very different compared to code reasoning[1].**

I. Introduction

Large Language Models (LLMs) have shown emerging abilities in code generation, specifically when instruction-tuned or prompted through Chain- or Tree-of-Thoughts (CoT [1] or ToT [2]) and in-context learning [3], [4]. However, several studies suggest that LLMs struggle to generalize this ability to real-world programs [5], [6] or to tasks that require understanding code logic rather than natural language [7], [8]. This is mainly because LLMs are trained to associate code synthesis with natural language specifications, i.e., combine code constructs similar to thousands to millions of examples they have seen while aligning to the requirements specified in the natural language. As a result, they intuitively have limited code reasoning to generalize to real-world problems or reliably perform broader program analysis tasks. While testing can validate the code LLMs generate, relying solely on test execution without assessing their abilities in code reasoning can be misleading.

To illustrate how code reasoning tasks better evaluate LLMs' abilities for coding, Figure 1-a shows a code synthesized by GPT-3.5 given a natural language specification. The code constructs corresponding to the concepts specified in natural language are highlighted with matching colors. Due to the ambiguity in the natural language, this code returns the smallest number in the list rather than the number at the index equal to the value of the smallest number. For a given input $[2, 5, 4, 3]$, the code returns 2 instead of 4, and the assertion fails.

One way to relieve the inevitable natural language ambiguity and improve the performance of code generation tasks, e.g., code synthesis or translation, is including test data in the prompt [7], [9]–[12]. Figure 1-b shows the new specification and corresponding generated code. Executing the new code with the specified input-output pair (and additional test data) results in a test pass, implying a level of code reasoning, which we refer to as Specification Reasoning (SR). However, it is a weak proxy for code reasoning as it still involves the association of code and natural language. A *stronger* level of code reasoning is following how given inputs to the code evolve into output through execution, which we call Execution Reasoning (ER). This task challenges LLMs more in reasoning about code without natural language cross-reference. Figure 1-c shows the CoT reasoning of GPT-3.5 in response to the ER task [2]. Even though the model previously generated this correct code itself (validated through testing), it cannot correctly reason how the *same inputs* evolve into output through code execution.

To advance the assessment of LLMs for programming tasks, this paper introduces CodeMind to enable code reasoning evaluation. CodeMind formally defines *three* inductive code

[1]The reasoning of LLMs and humans exhibit fundamental differences due to the distinct nature of their cognitive processes. Our conclusions on the extent of code reasoning abilities of LLMs do not imply human-like reasoning

[2]ER prompts are more complex than what is shown in the illustrative example.

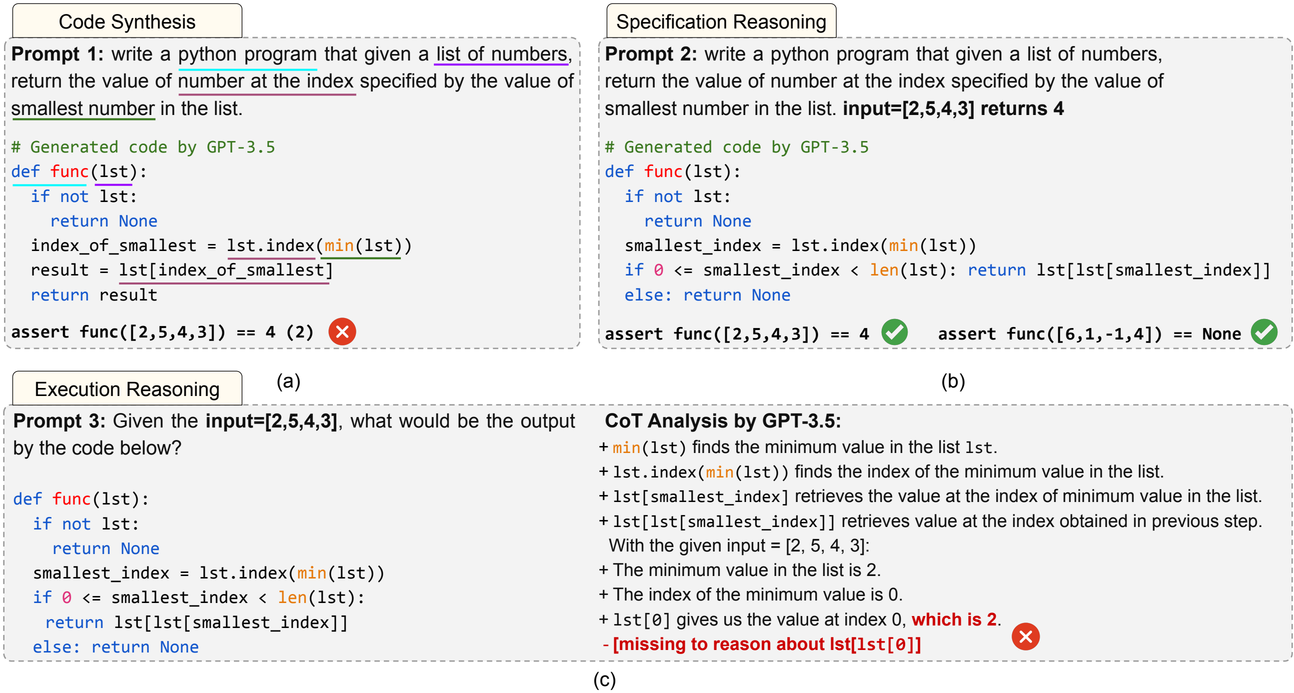

Fig. 1: An example illustrating the importance of evaluating LLMs on code reasoning

reasoning tasks and metrics: **Independent Execution Reasoning (IER)** and **Dependent Execution Reasoning (DER)** assess if LLMs can reason about how given inputs evolve to output for any arbitrary code (IER) or only the code that they correctly generate (DER). **Specification Reasoning (SR)** evaluates the extent to which LLMs can reason and implement the specified behavior. CodeMind supports code reasoning for two most common generative tasks: code synthesis—generating code given natural language specification—and code translation—generating code given another code in a different programming language as a specification.

Using CodeMind, we performed a large-scale study to assess state-of-the-art LLMs for code reasoning. We selected *ten* models, including both general-purpose and Code LLMs, and prompted them for IER, DER, and SR tasks on *5395* programs written in Java and Python. These programs are from *five* programming benchmarks, namely HumanEval [13], MBPP [14], CRUXEval [15] CodeNet [16], and Avatar [17]. We observe that:

(1) LLMs can *explain* the code statement by statement and often follow the execution flow. Yet, they fail to reason about output correctly, and their abilities are limited to simple programs. Open-source LLMs that have achieved comparable effectiveness as GPT models in code synthesis are behind them with a *notable gap* concerning code reasoning (§IV-A). In-depth analysis (§IV-D) suggests the root cause to be factors other than the difference in model size.

(2) LLMs often achieve a higher performance reasoning about the code (with similar or even higher complexity) they can correctly synthesize or translate (§IV-B). This is potentially because synthesis/translation already enforces a level of inductive reasoning, making the execution reasoning less challenging.

(3) LLMs, to a limited extent, can reason about test data in the specification, *even if deceptive*, and bring that into solving code synthesis and translation problems. Including test data helps code synthesis more than translation, likely due to ambiguity in natural language.

(4) Depending on the complexity and specific properties of the programs or programming language, there could be *a (negative) negligible to no correlation* between the ranking of models based on code synthesis/translation—generating a code that passes all tests—and code execution reasoning performance (§IV-B). This necessitates CodeMind tasks and metrics to complement the evaluation of LLMs for code.

(5) Nested code constructs, complex conditional predicates and loop conditions, the non-trivial combination of arithmetic and logic operators, and API invocations can significantly challenge LLMs for code reasoning (§IV-D). Our experiments show that simplifying the code logic, e.g., unrolling the loops to eliminate the necessity of reasoning about loop conditions, can increase the code reasoning performance.

Our contributions include (1) CodeMind framework for code reasoning, which is open-source [18] and is designed so that other researchers can add reasoning tasks to it. (2) a large-scale evaluation of LLMs for code reasoning using CodeMind for two code generation tasks; and (3) a comprehensive, in-depth analysis of results that offers a catalog of root causes negatively impacting the abilities of LLMs for code reasoning [3].

[3]Mechanistic interpretability of LLMs to investigate how underlying layers and properties of models result in such observation is a separate research and out of the scope of this work.

II. CodeMind

Program specification (either in natural language, code, or mathematical expressions) defines the logic that the code should implement. Formally speaking, it defines a function $S : S_I \rightarrow S_O$, where S_I is a set of all possible inputs to the program and S_O is a set of corresponding outputs. A code synthesized based on the implementation is also a function $C : C_I \rightarrow C_O$. We define a program to be *correct with respect to specification* if it satisfies all the following conditions:

$$C_I \subseteq S_I, C_O \subseteq S_O, \forall i \in C_I, C(i) = S(i)$$

This entails the model to reason about how inputs evolve to a given output through implementation (execution reasoning), and implement the code such that it generates correct output for the given inputs (specification reasoning).

A. Execution Reasoning

Considering the formalization mentioned above, we define two execution reasoning tasks as follows.

Definition 1: Independent Execution Reasoning (IER). Given a program $C : C_I \rightarrow C_O$ and set of inputs $\hat{I} = \{i | i \in C_I\}$, LLM L can correctly reason about code execution if $\hat{o} = C(\hat{I})$, where $\hat{o} = L(\hat{I})$ is the predicted output by L. Note that in this task, we do not deal with specification, so we can assess LLMs for any arbitrary code with ground-truth pairs of $\langle \hat{I}, \hat{o} \rangle$.

IER evaluates LLMs for general inductive code reasoning, which requires knowing different code constructs, arithmetic and logic operations, and programming language properties. However, even for human developers, reasoning about their developed code is easier than any arbitrary code. Furthermore, as a self-consistency [8] measurement, LLMs should be able to reason about the code they can correctly synthesize. This demands the following execution reasoning task:

Definition 2: Dependent Execution Reasoning (DER). Given a specification $S : S_I \rightarrow S_O$, a program $C : C_I \rightarrow C_O$ generated by LLM L, and set of inputs $\hat{I} = \{i | i \in C_I, C(i) = S(i)\}$, LLM L can correctly reason about code execution if $\hat{o} = C(\hat{I})$, where $\hat{o} = L(\hat{I})$ is the predicted output by L. The assumption here is that when LLM L generates code C that passes the test $\langle \hat{I}, \hat{o} \rangle$, it should be able to predict $\hat{o}$ correctly.

B. Specification Reasoning

In addition to execution reasoning, a model should understand specifications to synthesize the correct code. This specification can be given through natural language (in code synthesis) or programming language (in code translation). To evaluate if LLMs truly reason about the specification and not generate a probabilistic guess, we measure how including test data in the specification helps the model generate the correct code. Regardless of the type of specification, we define the specification reasoning task as follows.

Definition 3: Specification Reasoning (SR). Given a specification $S : S_I \rightarrow S_O$ in natural language or programming language, an arbitrary test $t = \langle i, o \rangle$, where $i \in S_I, o \in S_O, S(i) = o$, program $C_S : C_{S_I} \rightarrow C_{S_O}$ (generated by LLM L given the specification S), and program $C_{S+t} : C_{S+t_I} \rightarrow C_{S+t_O}$ (generated by LLM L given the specification S and t), the LLM can correctly reason about specification if $C_{S+t}(i) = o \quad \& \quad C_S(i) \neq o$. In other words, LLM L should be able to pass a test with $\langle i, o \rangle$ when they are explicitly specified in the prompt but fail it otherwise. This shows that the model has not just overfitted into the specification but can reason about it.

C. Evaluating Code Reasoning

1) Evaluating Execution Reasoning: We measure the performance of a model L in execution reasoning for a given program C with inputs $\hat{I}$ using the *Execution Reasoning Score (S_{ER})* as below:

$$S_{ER}(L, C, \hat{I}) = \begin{cases} 1, & \text{if } L(\hat{I}) = C(\hat{I}) \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

which is 1 if the model can correctly predict the expected output and 0 otherwise. Given that IER is measured on any arbitrary code, $S_{IER} = S_{ER}$. For DER, however, the reasoning is conditioned to the code correctness (Definition 2). As a result, we define S_{DER} as below:

$$S_{DER}(L, C, \hat{I}) = Pass_{\langle L, C, \hat{I} \rangle} \times S_{ER}(L, C, \hat{I}) \tag{2}$$

where $Pass_{\langle L, C, \hat{I} \rangle}$ denotes test result: 1 if the generated code C by the LLM L passes the tests with inputs $\hat{I}$ and 0, otherwise. We also introduce the *Execution Reasoning Rate (R_{ER})* metric, a collective metric that measures how much a given LLM L can reason about multiple programs in a benchmark. We calculate R_{ER} for a set of m programs in benchmark B as:

$$R_{ER}(L, B) = \frac{\sum_{i=1}^{m} [\![S_{ER}(L, C_i \in B) = 1]\!]}{m} \tag{3}$$

where $[\![]\!]$ denote the Iverson bracket: it returns 1 if the condition in square brackets is satisfied and 0 otherwise. We compute $R_{IER} = R_{ER}$ and R_{DER} as:

$$R_{DER}(L, B) = \frac{\sum_{i=1}^{m} [\![S_{DER}(L, C_i \in B, \hat{I}_i) = 1]\!]}{m} \tag{4}$$

2) Evaluating Specification Reasoning: We measure the performance of a model L in specification reasoning using S_{SR} as below:

$$S_{SR}(L, S, t) = (1 - Pass_{\langle L, C_S, t \rangle}) \times Pass_{\langle L, C_{S+t}, t \rangle} \tag{5}$$

S_{SR} is a conservative metric that rules out cases where LLMs only generate a correct code based on the natural language or programming language specification. However, given the possibility of data contamination, we need such a conservative metric as a lower bound for the reasoning abilities of LLMs. Similar to execution reasoning, we calculate the collective R_{SR} values for a set of m programs in benchmark B as:

$$R_{SR}(L, B) = \frac{\sum_{i=1}^{m} [\![S_{SR}(L, S_i \in B, t_i) = 1]\!]}{m} \tag{6}$$

TABLE I: Performance of subject LLMs (R_{IER}) on IER task through CoT prompting. We highlight the top three best-performing models with red (1st), green (2nd), and blue (3rd).

Dataset	Programming Language	# Subjects	General LLMs				Code LLMs					
			GPT-4	GPT-3.5	Llama 2	Mistral	CodeLlama	DeepSeekCoder	Magicoder	StarCoder	StarCoder2	WizardCoder
MBPP	Python	408	80.88%	71.32%	45.59%	31.37%	42.40%	57.84%	59.80%	43.63%	57.84%	46.08%
HumanEval	Python	162	79.01%	64.20%	30.86%	32.72%	45.06%	41.98%	52.47%	38.89%	46.30%	40.12%
CRUXEval	Python	800	80.50%	65.13%	25.38%	34.13%	37.75%	44.38%	46.50%	35.50%	52.00%	35.88%
CodeNet	Python	1914	70.43%	49.06%	18.97%	17.35%	27.95%	26.65%	33.28%	26.28%	43.22%	24.87%
	Java	1939	71.17%	51.93%	23.99%	18.15%	28.52%	32.13%	36.46%	29.34%	32.50%	29.35%
Avatar	Python	86	52.33%	39.53%	24.42%	16.28%	23.26%	18.60%	24.42%	19.77%	32.56%	24.42%
	Java	86	48.84%	34.88%	23.26%	11.63%	27.91%	23.26%	24.42%	13.95%	27.91%	13.95%
Total	Java and Python	5395	72.60%	54.24%	24.26%	21.54%	30.40%	33.85%	38.68%	30.14%	40.95%	29.99%

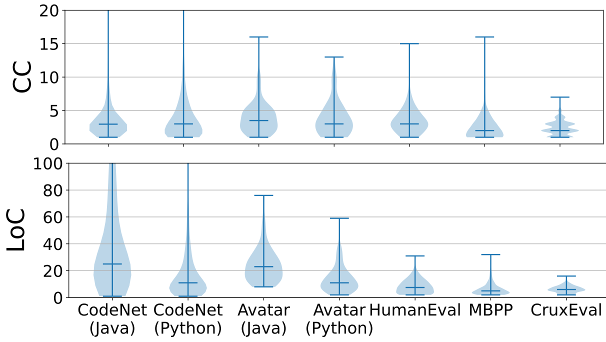

Fig. 2: Complexity distribution of the subject programs in terms of Cyclomatic Complexity (CC) and Line of Code (LoC)

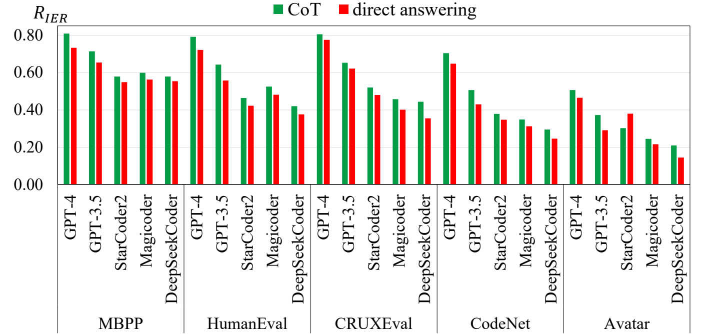

Fig. 3: Comparison in IER performance through CoT and direct answering prompting styles for five best-performing LLMs

III. Experimental Setup

Our study includes ten LLMs and 5395 programs in Java and Python from five programming datasets. We explain the details of LLMs and program selection below.

Subject LLMs. We chose ten pre-trained or instruction-tuned models, covering both general-purpose and Code LLMs. Our choice was limited to computing resources, so we selected models with less than 20B parameters that outperform the rest for programming tasks. Our subject LLMs are GPT-4 [19], GPT-3.5 [20], Llama 2 (13B) [21], Mistral [22], CodeLlama (13B, instruction-tuned) [23], StarCoder (15.5B) [24], StarCoder 2 (15B), WizardCoder (15B, instruction-tuned) [25], Magicoder (7B) [26] (instruction-tuned), DeepSeek-Coder (6.7B) [27]. We followed the best practices and customized the prompt templates per each model (all prompts are publicly available for further investigation [18]). Except for the GPT models, we set the temperature to zero to ensure the reproducibility of the results. Our code is open-source to users for using CodeMind for other models and temperatures.

Subject Programs. Our criteria for selecting subject programs were the existence of test data (inputs and corresponding expected output) and implementations of the same program in multiple programming languages (to investigate its impact on code reasoning). From several existing benchmarks [5], [6], [13]–[17], [28]–[32], we chose the programs in HumanEval [13], MBPP [14], CodeNet [16], Avatar [17], and CRUXEval [15]. We chose Java and Python versions of the programs as they are more prominently used programming languages. HumanEval and MBPP are well-known benchmarks for code synthesis. CodeNet and Avatar are code translation benchmarks. CRUXEval is a benchmark of relatively simple Python programs generated by CodeLlama (34B) to evaluate input prediction and output prediction of LLMs. Since the code in CodeNet and Avatar takes inputs interactively from the console, we only consider those with single-line input to eliminate the difficulty of understanding the input for LLMs and its impact on the code reasoning.

Figure 2 shows the complexity distribution of the programs in terms of *Cyclomatic Complexity (CC)* [33], and *Lines of Code (LoC)*. CC measures the independent execution paths in the program control flow graph (CFG). We calculate CC as $CC = E - N + 2P$, where E and N are the number of edges and nodes in the CFG, respectively, and P is the number of methods when we measure CC in for class. For code reasoning, the higher the number of independent paths, the more challenging it is for the model to succeed by chance.

IV. Results and Analysis

In this section, we evaluate LLMs' performance in IER (RQ1), DER (RQ2), and SR (RQ3), and investigate possible factors affecting their reasoning on code execution (RQ4).

A. RQ1. LLMs' Performance in IER

To evaluate the performance of LLMs on the IER task, we promoted them under two settings: direct answering and CoT. For direct answering, we prompted each model to predict the output for the given input(s). Under the CoT setup, we first instruct the model to *simulate* the execution step by step and reason about the output value after the execution of each statement. We then ask it to predict the output for the given input at the end of execution. In both settings, the prompt

contains *one* in-context example for two purposes: introducing the IER task and instructing the response formatting. Details about CoT and direct answering prompts can be found in our artifact website [18]. We set the temperature to 0 for the sake of reproducibility of our results.

Given that IER only requires an arbitrary code and corresponding ground-truth pair of $\langle \hat{I}, \hat{o} \rangle$ (§II-A), we prompted the LLMs using all 5395 subject programs in this experiment. Table I and Figure 3 [4] show the result of this experiment through CoT and direct-answering prompting. GPT models outperform others on the IER task, with large margins of 31.65% (GPT-4) and 13.29% (GPT-3.5) from the best open-source model, StarCoder 2. Detailed observations include:

- On CodeNet and Avatar with equivalent logic implemented in two programming languages, we observe no persistent gain or drop in R_{IER} from one language to another across studied LLMs. One-tail t-tests over the E_{IER} values show no statically significant dominance for any programming language (p-value=0.3 for Python dominance hypothesis and p-value=0.7 for Java dominance hypothesis with α=0.05).
- Moving down in the table, the R_{IER} values drop, i.e., execution reasoning on CodeNet and Avatar programs is harder, compared to MBPP, HumanEval, and CRUXEval. One potential reason is the varying complexity between the datasets and a *strong negative correlation* (measured by Spearman's Rank Order Correlation (ROC) [34]) between CC and R_{IER} values (Figure 4). At the same time, some models achieve a lower performance on CRUXEval compared to HumanEval, whose programs are more complex regarding both LoC and CC. Also, despite having similar CC distribution, models struggle more on the Avatar compared to CodeNet. This entails a better understanding of what factors other than CC impact the R_{IER} performance of the models (§IV-D). In other words, the notion of program complexity for LLMs could be different and, thereby, should be measured differently than classic complexity metrics.
- Compared to direct answering, CoT-style prompting improves the performance of LLMs on IER by 5.07% on average across all the benchmarks and LLMs. GPT models benefit more from CoT, likely due to their better and more rigorous alignment in thinking step by step. They also benefit from the natural language explanation they produce as feedback for the next step [19].

B. RQ2. LLMs' Performance in DER

We seek to address the critical question of how effectively the model can correctly reason about the *correct programs* it has generated through code synthesis or translation. This requires us to align code synthesis/translation and reasoning tasks. DER evaluation consists of three steps: (1) following the best practices, CodeMind prompts LLMs for code synthesis or translation; (2) it executes the generated program against existing tests; and (3) for the programs with tests pass, it prompts the model using CoT style for code execution reasoning using one of the test inputs.

We performed code synthesis on MBPP and HumanEval, as the other three datasets are not designed for code synthesis and lack proper natural language specifications. For code translation, we used CodeNet and Avatar, as they have ground-truth translations in both Java and Python programming languages. Furthermore, their test data can be reused to validate the code generated in both languages without inaccuracies potentially involved in test translation [7]. We excluded Llama 2 from this experiment as we could not reproduce their code synthesis results. We also removed the comments from the LLM-generated codes to ensure they do not impact the code reasoning. Similar to IER, we set the temperature to zero to account for the non-determinism and reproducibility of the results. Consequently, our synthesis and translation results might be different from existing leaderboards or published research[5].

Tables II-III present the results of DER on code synthesis and translation. In Table II, *Synthesis* rows show the percentage of programs in each dataset that models correctly synthesized, i.e., generated code passed *all* existing tests. In Table III, *Translation* rows show the percentage of programs in each dataset for a programming language pair that models were about to correctly translate, i.e., generated code passed *all* existing tests. The *Reasoning* rows in these two tables demonstrate the R_{DER} values (Equation 4). GPT models still outperform open-source models on the DER task, with a margin of 17.03% (GPT-4) and 12.50% (GPT-3.5) in code synthesis and 27.37% (GPT-4) and 13.55% (GPT-3.5) in code translation, from the best open-source model.

DER depends on successful code generation, causing different values for the denominator of Equation 4 across models. To ensure this does not threaten the validity of our conclusions, we also computed R'_{DER} on the overlap between correctly generated code across all LLMs. In code translation, we excluded StarCoder and WizardCoder due to their poor performance. While R'_{DER} values are higher than R_{DER} due to a smaller denominator, the ranking between the models in code reasoning remains almost the same.

Models achieve higher R_{DER} compared to R_{IER} (from Table I) in both synthesis (7.51% higher) and translation (10.28% higher). We performed another controlled experiment and calculated R'_{IER}, accumulated on S_{IER} of the ground-truth code for the programs that LLMs achieved $S_{DER} = 1$. New experiments show that, except in a few cases, the R_{DER} values are still higher than R'_{IER}, but with a lower improvement of 1.47% for synthesis and 6.32% for translation.

Before concluding that the models are more competent in execution reasoning when evaluated on the programs they correctly generate, we investigated whether LLM-generated programs are similar to original ground-truth codes in IER. Figure 5 shows the CC distribution of the programs in MBPP, HumanEval, CodeNet, and Avatar, compared to that generated

[4]The comparison between CoT and direct-answering prompting styles for all the models is available on the artifact website [18].

[5]For translation task, our studied subject programs (based on inclusion criteria mentioned in §III) is different from [7], which also contributes to the potential difference.

TABLE II: Performance of subject LLMs on DER task through CoT prompting for code synthesis. We highlight the top three best-performing models with red (1st), green (2nd), and blue (3rd).

Dataset	# Subjects	Task	General LLMs			Code LLMs					
			GPT-4	GPT-3.5	Mistral	CodeLlama	DeepSeekCoder	Magicoder	StarCoder	StarCoder 2	WizardCoder
MBPP	408	Synthesis	86.52%	80.39%	43.36%	56.86%	72.30%	70.34%	44.85%	61.12%	61.03%
		Reasoning	82.62%	79.20%	43.50%	43.53%	69.39%	69.34%	56.83%	63.49%	48.19%
	93	R'_{DER}	90.23%	84.95%	46.24%	49.46%	75.27%	78.49%	60.22%	68.81%	65.59%
$R_{DER} - R_{IER}$			1.74	7.88	11.89	1.13	5.15	9.54	13.20	5.65	2.11
$R_{DER} - R'_{IER}$			1.94	$-0.05\downarrow$	3.41	$-3.01\downarrow$	1.08	6.17	3.31	7.23	$-6.68\downarrow$
HumanEval	162	Synthesis	79.63%	69.75%	29.94%	43.11%	72.46%	70.37%	41.98%	45.06%	51.50%
		Reasoning	81.40%	74.63%	34.12%	35.09%	54.55%	57.58%	58.97%	45.45%	59.50%
	47	R'_{DER}	85.10%	80.84%	29.79%	40.42%	57.45%	55.32%	48.45%	61.29%	59.57%
$R_{DER} - R_{IER}$			0.64	10.43	1.4	$9.97\downarrow$	12.57	5.09	9.56	0.85	19.38
$R_{DER} - R'_{IER}$			0.98	2.20	$-2.09\downarrow$	$-7.71\downarrow$	9.87	0.65	2.85	3.32	2.39

TABLE III: Performance of subject LLMs on DER task through CoT prompting for code translation. We highlight the top three best-performing models with red (1st), green (2nd), and blue (3rd).

Dataset	# Subjects	Task	General LLMs			Code LLMs					
			GPT-4	GPT-3.5	Mistral	CodeLlama	DeepSeekCoder	Magicoder	StarCoder	StarCoder 2	WizardCoder
Avatar	86	Translation(Python $\rightarrow$ Java)	55.81%	51.16%	22.09%	36.05%	52.33%	50.00%	23.26%	60.47%	39.53%
		Reasoning	72.92%	59.09%	21.05%	29.03%	35.56%	37.21%	20.59%	37.50%	20.59%
	12	R'_{DER}	75%	66.67%	33.33%	16.67%	41.67%	41.67%	-	50%	-
$R_{DER} - R_{IER}$			24.08	24.21	9.42	1.12	12.30	12.79	6.64	9.59	6.64
$R_{DER} - R'_{IER}$			22.92	18.47	13.05	$-18.54\downarrow$	1.00	9.79	4.68	2.42	0.59
Avatar	86	Translation(Java $\rightarrow$ Python)	56.98%	68.60%	30.23%	58.14%	55.81%	65.12%	39.53%	18.60%	2.33%
		Reasoning	59.18%	49.15%	15.38%	22.00%	25.00%	28.57%	30.00%	40.38%	0%
	6	R'_{DER}	83.33%	50%	50%	50%	33.33%	66.67%	-	66.67%	-
$R_{DER} - R_{IER}$			6.85	9.62	3.75	1.26	6.40	4.15	10.23	7.82	24.42
$R_{DER} - R'_{IER}$			$-0.14\downarrow$	1.01	20.23	0.96	$-1.78\downarrow$	5.98	7.27	15.38	0
CodeNet	691	Translation(Python $\rightarrow$ Java)	74.85%	70.32%	35.13%	45.18%	69.59%	69.44%	47.51%	26.02%	55.41%
		Reasoning	82.03%	64.24%	41.73%	45.81%	30.72%	42.11%	37.23%	38.76%	41.42%
	60	R'_{DER}	96.67%	68.33%	35.00%	55.00%	16.17%	43.33%	-	46.67%	-
$R_{DER} - R_{IER}$			10.86	15.18	23.58	17.29	1.41	5.65	7.89	6.26	12.07
$R_{DER} - R'_{IER}$			4.30	5.61	8.66	5.68	$-3.31\downarrow$	11.26	$-0.62\downarrow$	$-1.13\downarrow$	6.06
CodeNet	691	Translation(Java $\rightarrow$ Python)	51.66%	52.53%	24.46%	37.05%	44.28%	45.73%	6.95%	38.78%	2.03%
		Reasoning	73.67%	60.06%	26.63%	40.56%	35.62 %	47.46%	40.82%	54.65%	21.43%
	58	R'_{DER}	91.38%	74.17%	27.59%	43.10%	41.38%	56.90%	-	62.07%	–
$R_{DER} - R_{IER}$			3.24	11.00	9.28	12.61	8.97	14.18	14.54	11.43	3.44
$R_{DER} - R'_{IER}$			$-0.28\downarrow$	3.31	$-3.55\downarrow$	3.59	$-6.54\downarrow$	3.16	3.32	4.28	$-21.52\downarrow$

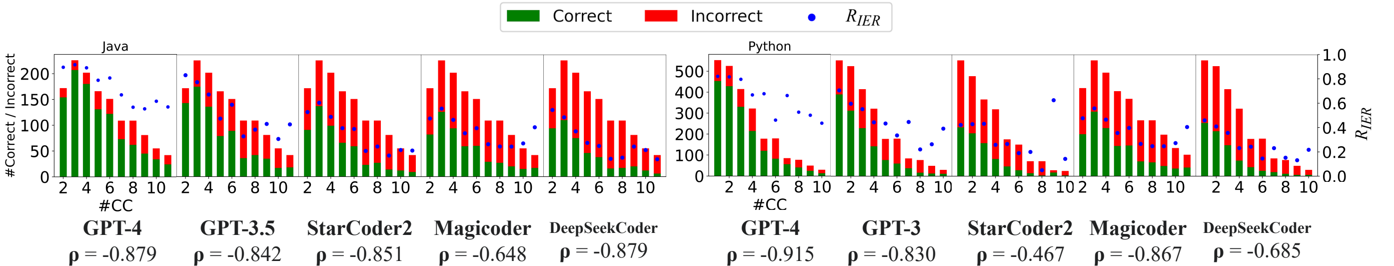

Fig. 4: Impact of CC on the IER performance of LLMs. The ρ symbol denotes the Person's ROC coefficient

by subject LLMs. We can observe that the synthesized/translated code, if not more complex, is no less than the ground-truth code in these datasets. CC is only a proxy, and other factors may impact the reasoning of models. However, a consistent and intuitive negative correlation between reasoning abilities and CC across LLMs increases its impact size. Consequently, we confirm that models reason better on code they correctly generate. However, there is still a considerable gap between the LLM's code synthesis and reasoning abilities, specifically on open-source models.

Given that code generation and reasoning are unified in DER, we first computed the Spearman's ROC between the rank of models based on their performance in synthesis/translation and reasoning for each dataset. The results show a *strong positive correlation* on MBPP ($\rho = 0.79$), a *negligible negative correlation* on HumanEval ($\rho = -0.11$), *strong correlation* on CodeNet (Java to Python $\rho = 0.54$), *weak negative correlation* on CodeNet (Python to Java $\rho = -0.25$), *negligible negative correlation* on Avatar (Java to Python $\rho = -0.18$), and *strong correlation* on Avatar (Python to Java $\rho = 0.59$).

These results communicate a strong message: the ranking of LLMs based on their code generation abilities and test passes could be different from that based on code reasoning. Models may overfit widely used datasets such as HumanEval, giving a false promise on the abilities of LLMs for programming. This necessitates a framework such as CodeMind to promote other evaluation aspects of Code LLMs as well as the need for more diverse and contamination-free evaluation datasets.

TABLE IV: Performance of LLMs on SR task for code synthesis. ↑ indicates the improvement from *No Test* to *With Test*.

Dataset	Setting	General LLMs			Code LLMs					
		GPT-4	**GPT-3.5**	**Mistral**	**CodeLlama**	**DeepSeekCoder**	**Magicoder**	**StarCoder**	**StarCoder 2**	**WizardCoder**
MBPP	With Test	90.69% ↑	85.05% ↑	50.74% ↑	63.73% ↑	78.68% ↑	75.25% ↑	51.47% ↑	66.98% ↑	67.89% ↑
	No Test	72.13%	78.87%	48.28%	53.68%	67.65%	69.61%	41.67%	54.80%	52.21%
	Misleading Test	14.22%	12.99%	8.58%	10.54%	12.01%	9.30%	10.78%	11.52%	12.25%
HumanEval	With Test	91.98% ↑	74.07% ↑	57.41% ↑	70.37% ↑	87.04% ↑	81.48% ↑	56.17% ↑	58.64% ↑	76.54%
	No Test	88.27%	70.37%	54.32%	65.43%	82.10%	80.86%	38.89%	41.46%	76.54 %
	Misleading Test	17.28%	16.67%	9.58%	11.98%	12.96%	17.90%	15.43%	12.96%	17.28%

TABLE V: Performance of LLMs on SR task for code translation. ↑ indicates the improvement from *No Test* to *With Test*.

Dataset	Task	Setting	General LLMs			Code LLMs					
			GPT-4	**GPT-3**	**Mistral**	**CodeLlama**	**DeepSeekCoder**	**Magicoder**	**StarCoder**	**StarCoder2**	**WizardCoder**
Avatar	Python → Java	With Test	78.75% ↑	75.00%↑	35.00%	45.00%↑	78.75%↑	70.00%↑	53.75%↑	43.75%↑	55.00%
		No Test	68.60%	62.79%	29.07%	43.02%	65.12%	63.95%	51.16%	23.26%	46.51%
		Misleading Test	4.65%	3.49%	2.33%	2.33%	3.49%	3.49%	3.49%	1.16%	3.49%
	Java → Python	With Test	82.72%↑	88.89%↑	49.38% ↑	64.20%	77.78%↑	86.42%↑	64.20%↑	74.07%↑	29.63%↑
		No Test	62.79%	74.42%	34.88%	66.28%	61.63%	72.09%	25.58%	66.28%	3.49%
		Misleading Test	4.65%	1.16%	1.16%	2.33%	2.33%	3.49%	2.33%	2.33%	0%
CodeNet	Python → Java	With Test	72.70%	74.71%↑	41.67% ↑	45.91%↑	72.51%↑	61.11%	48.39%↑	47.81%↑	56.14% ↑
		No Test	74.85%	70.32%	37.13%	45.18%	69.59%	69.44%	47.51%	26.02%	55.41%
		Misleading Test	4.63%	3.62%	2.32%	2.02%	3.18%	3.04%	2.75%	2.32%	5.20%
	Java → Python	With Test	82.05%↑	74.38%↑	34.44% ↑	37.19%↑	60.20%↑	62.52%↑	36.47%↑	53.98%↑	10.13%↑
		No Test	51.66%	52.53%	24.46%	37.05%	44.28%	45.73%	6.95%	38.78%	2.03%
		Misleading Test	4.49%	4.20%	1.45%	2.17%	3.47%	4.05%	0%	3.62%	0.14%

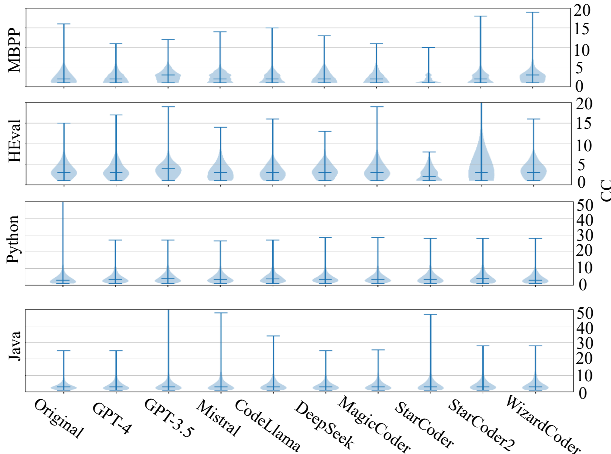

Fig. 5: CC distribution of the programs generated by LLMs (synthesized or translated) compared to the original programs in the HumanEval, MBPP, CodeNet, and Avatar (Java and Python samples from CodeNet and Avatar are aggregated)

C. RQ3. LLMs' Performance in SR

Specification Reasoning (SR) offers a novel perspective in understanding the code generation process of LLMs, particularly about how they leverage input-output pairs in the specifications for code generation. To evaluate the abilities of LLMs for SR, we prompted LLMs for code synthesis and translation under the following three settings:

(1) Natural language specification with one ground-truth input-output (With Test). Under this setting, we randomly select and add one of the existing tests to the specification. We validate the synthesized code using only this test.

(2) Natural language specification with no input-output (No Test). We remove the test added to the specification in the previous setting and re-prompt LLMs for code generation. We validate the synthesized code using only the test from the previous setting. Intuitively, if including test data helps LLMs in code synthesis, we observe a drop in LLMs' performance.

(3) Natural language specification with misleading input-output (Misleading Test). We mutate the expected output of the test from the first setting and add it to the specification. We validate the synthesized code using `the mutated test`. The mutation changes the expected output to a value that does not align with the specification. For example, if the expected output is `True`, mutation changes it to `False`. Similarly, if the expected output is a positive integer, we mutate it to a negative one with a significant difference. Compared to the *With Test* example, passing on the misleading test is stronger evidence that models can reason about the test data in the specification and incorporate it into code generation.

Similar to §IV-B, we used MBPP/HumanEval[6] for code synthesis and Avatar/CodeNet for code translation. The results in Table IV show that the performance of LLMs with test data in their specification is higher in both code synthesis (8.17%) and translation (11.24%). The R_{SR} (Equation 6 values for code synthesis (0.37) and translation (0.14) suggest that among the problems that LLMs could not generate a correct code given the specification with no test, 37% and 14% can pass after adding the tests in the specification. Introducing deceptive test information surprisingly leads to test passes (13.01% in code synthesis and 2.79% in code translation, on average). Code synthesis is seemingly more attentive to misleading tests than code translation. We speculate this to be due to ambiguity in natural language compared to code, making LLMs generate a different code in the presence of misleading tests. These results show that LLMs, although to a limited extent, consider the test data in the specification during code generation.

D. RQ4. Factors Impacting LLMs' Code Execution Reasoning

We further analyzed the IER results, which evaluate the general ability of LLMs in code reasoning. In the first step,

[6] HumanEval specification contains some test data by default. We have used a pre-processed version of this dataset in all our experiments.

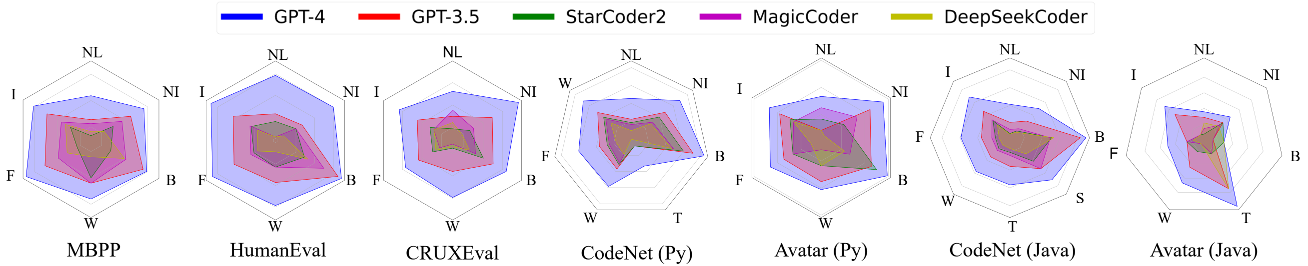

Fig. 6: IER performance of top five best-performing LLMs on programs clustered per specific code constructs across all datasets. We abbreviate the tags with B (Basic), F (For), I (If), NI (Nested If), NL (Nested Loop), S (Switch), T (Try), and W (While)

TABLE VI: Impact of unrolled loops on the performance of LLMs on IER task.

Language	#Subjects	Setting	GPT-4	GPT-3	Llama2	Mistral	CodeLlama	DeepSeekCoder	Magicoder	StarCoder	StarCoder2	WizardCoder
Python	1366	Loops	74.01%	51.45%	21.13%	18.59%	28.23%	31.41%	36.58%	25.90%	28.70%	28.26%
		Unrolled Loops	78.77%	57.10%	22.92%	21.89%	32.65%	35.29%	41.43%	30.23%	31.48%	30.64%
Java	867	Loops	66.21%	44.75%	13.26%	16.38%	26.18%	29.87%	35.18%	25.84%	29.76%	25.49%
		Unrolled Loops	72.55%	49.25%	16.15%	17.19%	28.71%	34.14%	38.87%	28.84%	34.71%	28.95%

we wanted to see if LLMs know how different code constructs work. Without knowing the logic of each code construct, reasoning about code execution is impossible. To that end, we tagged each of 5395 programs based on code constructs used in their implementation with the following labels: **For**, **While**, **If**, **Try**, **Switch**, **Nested Loop**, **Nested If**, and **Basic**. A program tagged with a Basic label has no particular code construct. Next, we clustered the programs per tag and aggregated S_{IER} values of LLMs for programs in each cluster. Figure 6 shows the results of this analysis for the top five best-performing LLMs (results for all models are available at [18]). We can observe that models handle conditional statements better than recursion, except for Try-Catch/Except statements. Furthermore, the R_{IER} values (aggregated S_{IER} values for programs) notably drop when it comes to nested constructs.

1) Impact of Loop Properties: Given that models struggle the most with recurring constructs, we focused the programs with For, While, and Nested Loop tags at the next step. We manually investigated 2862 programs with loops and identified the following common factors impacting the abilities of LLMs to reason about loops.

- *Loop Control Condition.* Without knowing the precise number of iterations, it is impossible to predict the execution output correctly. In the majority of failed IER cases, the loop control conditions are either complex expressions, e.g., `c<=(F-100*A*a-100*B*b)/(C)` or `val%(long)Math.pow(div,count)==0`, or if simple, e.g., `i<ff.length`, require reasoning about variable values before the loop.
  To confirm the importance of loop control conditions, we *unrolled* loops using the actual execution information obtained from Java and Python debugger. Unrolling loops involves repeating the loop body, i.e., statements inside the loop, for the number of loop iterations, depending on the test input. After unrolling the loops, we repeated the IER experiment to calculate R_{IER} values. We excluded 3162 programs from RQ1, as loop unrolling caused their lengths to exceed the context window of subject LLMs. The results of this experiment (Table VI) on the remaining 2233 programs show unrolling loops improve the IER performance of LLMs by 4.23% on average.
- *Loop Length.* In many cases, LLMs struggled to keep track of data flow after a few loop iterations and predicted an incorrect output for the entire execution. The *moderate to strong negative correlation* between R_{IER} values and the loop length demonstrated in Figure 7 further confirms the inverse impact of loop length on the code execution reasoning, i.e., R_{IER} decreases as the loop length increases.
- *Loop Body.* The complexity of the code inside the loop is also important for correct output prediction. We observed several instances where LLMs correctly realized the number of iterations for a loop with a short length but could not predict the output due to a mistake in predicting variable values local to the loop.
  In the example below (p02994 from CodeNet-Java), the loop condition is simple. CodeLlama can correctly reason about the number of loop iterations but fails to calculate the values in the `diff` array, resulting in a wrong output. Magicoder can also correctly reason about loop length and correctly *explain* what the code does. However, it cannot predict the output correctly (predicted output is 10), preventing us from judging if it *understands* the code as well as it *explains* it.

```
Input:5 2, Expected Output:18
int n = scanner.nextInt();
int l = scanner.nextInt();
List<Integer> diff = new ArrayList<Integer>();
for(int i = 1; i < n+1; i++) {
    int value = l+i-1;
    if(value < 0) {value = value * -1;}
    diff.add(value);
}
removeK = diff.indexOf(Collections.min(diff)) + 1;
println(n*l + n*(n+1)/2 - n - (l+ removeK -1));

CodeLlama, Predicted Output:24
Analysis: In the given inputs n is 5 and l is 2. The
sequence of numbers from 1 to 5 is [1,2,3,4,5]. The
```

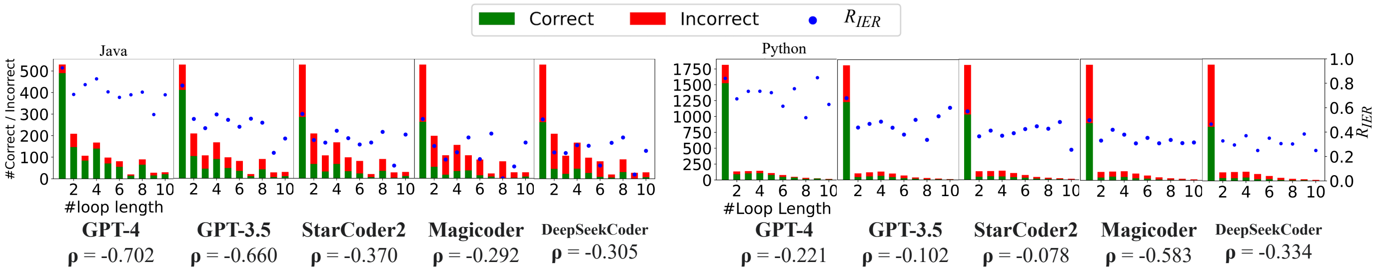

Fig. 7: Impact of loop length in Java and Python programs (aggregated for all five benchmarks) on LLMs' performances

```
differences between 1 and each number in the sequence are
[1,0,1,2,3]. The smallest difference is 0 at index 1. So,
the index 1 is removed from the sequence. The new
sequence becomes [1,3,4,5]. The sum of the original
sequence [1,2,3,4,5] and the new sequence [1,3,4,5] is
calculated as (5*2+5*(5+1)/2-5-(2+1-1))=15+15-5-2=24.
```

- *Recursion Logic.* While this happens rarely, we observed cases of LLMs hallucinating and struggling to follow the basic logic of the loop, e.g., terminate the loop once the control condition is met. In the code snippet below (p03059 from CodeNet-Java), the loop condition depends on a constant variable `c` and variable `time`, whose value changes inside the loop. All models failed in IER on this relatively simple program.

```
Input:3 5 7, Expected Output:10
int a, b, c = sc.nextInt();
int amount = 0, time = a;
for(int i = 2; time <= c + 0.5; i++){
    amount += b;
    time = a * i;
}
println(amount);
```

Looking at the CoT analysis, GPT-3.5 knows the loop condition and can reason about all the variable values in the first and second iterations. However, it stays in the loop for one more iteration. For other models, the reasoning about this code is more challenging.

```
GPT-3.5, Predicted Output:15
Analysis: Enters for loop and iterates from 2 onwards
    until
the value of time exceeds c + 0.5. In first iteration,
amount becomes 5 and time becomes 6. In second iteration,
amount becomes 10 and time becomes 9. In third iteration,
amount becomes 15 and time becomes 12. 12 is greater than
7+0.5, loop exits.
```

2) Impact of if-else Properties: We identify the following three factors impacting the IER performance in code with conditional constructs:

- *Conditional Predicates.* LLMs often struggle to realize the conditional predicates correctly, take the wrong branch, and mispredict the output. In the example below (P01741 from CodeNet-Java), StarCoder accurately reads the input and assigns it to variable `d`. However, it incorrectly concludes the value of `d` is less than one and takes an incorrect branch.

```
Input:1.00, Expected Output:2
d = sc.nextDouble();
if(d < 1)out.println(d * Math.sqrt(2));
else if(d < Math.sqrt(2))out.println("2");
else if(d < 2)out.println(d * Math.sqrt(2));
else if(d < Math.sqrt(2) * 1.5)out.println("3");
else out.println(d * Math.sqrt(2));

StarCoder, Predicted Output:1.414
Analysis: The code reads a double value from the input
and assigns it to d. Then it checks the value of d using
a series of if-else statements. The value of d is 1.0,
which is less than 1. The code executes the first if
statement and calculates d*Math.sqrt(2), which is 1.414.
```

- *Nested Constructs. Nested conditional statements* make it even harder for LLMs to follow the program's flow. In the following example (P03865 from CodeNet-Java), GPT-3.5 can correctly reason about the conditional predicates, i.e., it takes the first outer branch and the first branch inside it. However, it hallucinates that since two conditions are `True`, then the boolean values inside the inner branch should also be `True`. As a result, it takes the true branch of the last conditional statements and returns an incorrect answer.

```
Input:aba, Expected Output:Second
String s = sc.next();
int a = s.length();
boolean flag = true;
if(s.charAt(0) == s.charAt(a-1)){
    if(a%2 == 1){flag = false;}
    else{flag = true;}
} else{
   if(a%2 == 1){flag = true;}
   else{flag = false;}
}
if(flag){println("First");}
else{println("Second");}
GPT-3.5, Predicted Output:First
Analysis: The variable s is "aba". The length of s is 3.
The variable flag is true. The code checks if the first
character of s is equal to the last character of s. Both
are a, so the condition is true. Then it checks if the
length of s is odd or even. Since a is 3 and 3%2 is 1,
    the
condition is true. Since both conditions are true, the
value of flag remains true. The code then prints "First".
```

- *Control Flow Logic.* We observed a few cases where an LLM struggles recognizing conditional statements. For example, regardless of which branch to take, it always goes into both *if* and *else* branches to reason about execution.

3) Impact of Math and Logic Operators: Knowing how the math and logic operators work is essential to reasoning how inputs evolve into outputs through execution. The observations from previous analyses also underscore their importance to reasoning about the program control flow (loop control condition and conditional predicates). Arithmetic and logic operators are prevalent in all studied programs, with the former dominating the number of the latter. LLMs struggled more to

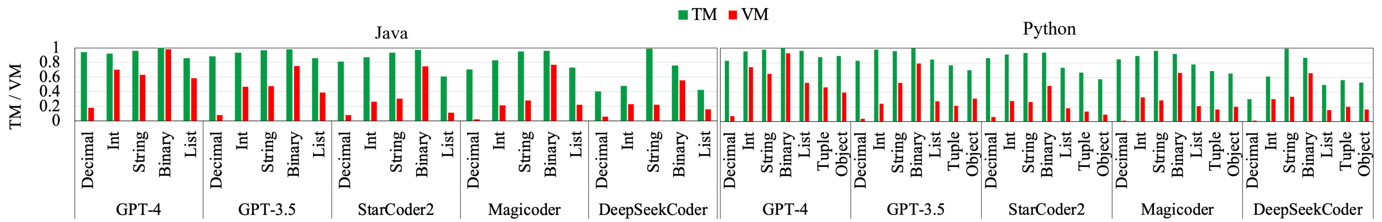

Fig. 8: Performance of LLMs on predicting different types of outputs

reason about bitwise operators (e.g., `>>>`, `∧`, `>>`, `<<`, `|`, and `&`) compared to arithmetic and comparison operators.

The code snippet below (MBPP-task 311) contains several arithmetic and logic operators. StarCoder correctly knows how most operators work. It can reason that the bitwise AND of `n` and `n+1` is used to determine whether integer `n` is a power of 2. It also knows that the loop condition checks if the least significant bit of `n` is 0. However, it seemingly cannot understand the meaning of the return statement (setting the least significant bit to 1), and fails to predict the correct output.

```
Input:10, Expected Output:14
def set_left_most_unset_bit(n):
  if not (n & (n + 1)): return n
  pos, temp, count = 0, n, 0
  while temp:
    if not (temp & 1): pos = count
    count += 1
    temp>>=1
  return (n | (1 << (pos)))

StarCoder, Predicted Output:10
Analysis: The variable n is initialized to 10. The code
checks if n is a power of 2. If it is, the code returns
n. If it is not, the code calculates the position of
the leftmost unset bit. The code then returns the
result of n | (1 << (pos)).
```

4) Impact of API Calls: APIs are inseparable parts of programs and LLMs should also reason about the output of API calls in the implementation: either know the API's logic, likely from the training, or make an informed guess based on the API name. By investigating the failed IER cases, we observed LLMs struggle to even reason about the output of commonly used APIs. In the following program (p02836 from CodeNet-Java), GPT-3.5 struggles with `charAt()` API. The return value of `s.charAt(6)` and `s.charAt(5)` should be `'e'` and `'d'`. However, GPT-3.5 concludes a reverse order for the return values, `'d'` and `'e'`.

```
Input:redcoder, Expected Output:1
Scanner sc = new Scanner(System.in);
String s = sc.next();
int length = s.length();
int count =0;
for(int i=0;i<length/2;i++){
    char a = s.charAt(i);
    char b = s.charAt(length - 1 -i );
    if(a != b){count++;}
}
System.out.println(count)

GPT-3.5, Predicted Output:3
Analysis: The string is "redcoder" and the length is 8. The
characters at index 0 and index 7 are r and r respectively,
which are equal. The characters at index 1 and index 6 are
e and d  respectively, which are not equal. The characters
at index 2 and index 5 are 'd' and 'e' respectively, which
are not equal. The characters at index 3 and index 4 are
'c' and 'o' respectively, which are not equal. So, the
count is incremented 3 times. The output will be 3.
```

5) Impact of Output Types: To better understand the impact of types on IER results, we categorized programs based on the output types and checked (1) if LLMs were able to predict the `type` of output correctly (Type Match) and (2) if they could correctly reason about the *values* of specific output types (Value Match). We identified seven types, namely `Int` (e.g., 2), `Decimal` (e.g., 2.34), `String` (e.g., "CodeMind"), `Binary` (e.g., True or False), `List` (e.g., [1,3,4,7]), `Tuple` (Python-specific, e.g., (2,7)) and `Object` (Python-specific, e.g., "age":10, "gender":"female").

Figure 8 shows the results. We can observe that LLMs achieve a high TM ($> 80\%$) for both Java and Python. However, they struggle to reason about correct value. Considering type categories, LLMs are more successful at TM and VM for primitive types while struggling to predict the type and values of outputs with more complex types such as Tuples, Lists, and Decimals. This is because complex types consist of multiple items with primitive or non-primitive types, recursively challenging LLMs for type and value match.

V. Related Work

A large body of work has assessed LLMs for reasoning tasks of different modalities [8], [35]–[43], including natural language, visual data, math, logic, and code. CodeMind is more closely related to the very recent studies focusing on code reasoning [15], [44], [45].

A closely related work proposes CRUXEval benchmark to assess the code reasoning abilities of LLMs. The dataset consists of simple programs generated by CodeLlama (34B) with test cases [15]. They evaluated a series of LLMs on CRUXEval for input and output prediction tasks. Compared to CRUXEval, CodeMind proposes more inductive code reasoning tasks, includes more programs with a variety of levels of complexity, and controls between code synthesis and reasoning tasks by evaluating LLMs using the same program. CodeMind is also equipped with a static analysis pipeline to enable in-depth examination and drawing informed conclusions.

Another related work [44] evaluates LLMs to predict variable values at each statement. Our experiments are larger compared to them: more programs with a diverse distribution of complexity and different programming languages, and more

studied LLMs. We also offer more code reasoning tasks and present a cross-analysis of code synthesis/translation and reasoning abilities.

Zhang et al. [45] investigate transformers' ability to infer the recursive patterns from input and output pairs. They conclude that due to the inherent limitations of transformers, they may fail to learn recursion and instead find shortcut algorithms to reason about how outputs are related to inputs. Compared to this work, we evaluate LLMs regardless of architecture and training data but from the program perspective. We show LLMs can follow recursion but usually lose track of data flow due to the inability to reason about loop conditions correctly.

VI. Concluding Remarks

In this paper, we discussed the necessity of code reasoning tasks as an alternative to evaluate LLMs for programming tasks. We introduced CodeMind, a framework that supports several code reasoning tasks, and used CodeMind in a large-scale grounded theory study to evaluate state-of-the-art LLMs for code reasoning. Our results demonstrate that LLMs, in general, know how code constructs work and achieve some levels of reasoning about program specifications. They may also follow how inputs evolve to output through execution. However, their ability is limited as the code becomes more complex, i.e., has more complex control- or data flow, contains non-primitive types and invokes API calls. We also observe that specification reasoning, which is essential to generate a code from a given program specification (in natural language or code), does not mean models can also reason about code execution.

References

[1] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[2] 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," *arXiv preprint arXiv:2305.10601*, 2023.

[3] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler *et al.*, "Emergent abilities of large language models," *arXiv preprint arXiv:2206.07682*, 2022.

[4] S. Garg, D. Tsipras, P. S. Liang, and G. Valiant, "What can transformers learn in-context? a case study of simple function classes," *Advances in Neural Information Processing Systems*, vol. 35, pp. 30 583–30 598, 2022.

[5] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.

[6] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[7] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Understanding the effectiveness of large language models in code translation," *arXiv preprint arXiv:2308.03109*, 2023.

[8] M. J. Min, Y. Ding, L. Buratti, S. Pujar, G. Kaiser, S. Jana, and B. Ray, "Beyond accuracy: Evaluating self-consistency of code large language models with identitychain," *arXiv preprint arXiv:2310.14053*, 2023.

[9] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[10] M. Zhong, G. Liu, H. Li, J. Kuang, J. Zeng, and M. Wang, "Codegen-test: An automatic code generation model integrating program test information," *arXiv preprint arXiv:2202.07612*, 2022.

[11] F. Shi, D. Fried, M. Ghazvininejad, L. Zettlemoyer, and S. I. Wang, "Natural language to code translation with execution," *arXiv preprint arXiv:2204.11454*, 2022.

[12] K. Zhang, D. Wang, J. Xia, W. Y. Wang, and L. Li, "Algo: Synthesizing algorithmic programs with generated oracle verifiers," *arXiv preprint arXiv:2305.14591*, 2023.

[13] 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.

[14] A. Odena, C. Sutton, D. M. Dohan, E. Jiang, H. Michalewski, J. Austin, M. P. Bosma, M. Nye, M. Terry, and Q. V. Le, "Program synthesis with large language models," in *n/a*, n/a, 2021, p. n/a, n/a.

[15] 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.

[16] R. Puri, D. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, L. Buratti, S. Pujar, and U. Finkler, "Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks," 2021.

[17] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, "Avatar: A parallel corpus for java-python program translation," *arXiv preprint arXiv:2108.11590*, 2021.

[18] CodeMind, "Artifact website," https://github.com/Intelligent-CAT-Lab/CodeMind, 2024.

[19] OpenAI, "Gpt-4 technical report," *https://arxiv.org/abs/2303.08774*, 2023.

[20] ——, "Chatgpt: Optimizing language models for dialogue," *https://openai.com/blog/chatgpt*, 2023.

[21] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[22] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[23] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[24] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023.

[25] C. Xu, Q. Sun, K. Zheng, X. Geng, P. Zhao, J. Feng, C. Tao, and D. Jiang, "Wizardlm: Empowering large language models to follow complex instructions," *arXiv preprint arXiv:2304.12244*, 2023.

[26] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Source code is all you need," *arXiv preprint arXiv:2312.02120*, 2023.

[27] X. Bi, D. Chen, G. Chen, S. Chen, D. Dai, C. Deng, H. Ding, K. Dong, Q. Du, Z. Fu *et al.*, "Deepseek llm: Scaling open-source language models with longtermism," *arXiv preprint arXiv:2401.02954*, 2024.

[28] S. Wang, Z. Li, H. Qian, C. Yang, Z. Wang, M. Shang, V. Kumar, S. Tan, B. Ray, P. Bhatia *et al.*, "Recode: Robustness evaluation of code generation models," *arXiv preprint arXiv:2212.10264*, 2022.

[29] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang *et al.*, "Multi-lingual evaluation of code generation models," *arXiv preprint arXiv:2210.14868*, 2022.

[30] 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," *arXiv preprint arXiv:2305.01210*, 2023.

[31] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li *et al.*, "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5673–5684.

[32] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman *et al.*, "Multipl-e: A scalable and extensible approach to benchmarking neural code generation," *arXiv preprint arXiv:2208.08227*, 2022.

[33] G. K. Gill and C. F. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE transactions on software engineering*, vol. 17, no. 12, pp. 1284–1288, 1991.

[34] C. Spearman, "The proof and measurement of association between two things." 1961.

[35] R. Deshpande, J. Chen, and I. Lee, "Rect: A recursive transformer architecture for generalizable mathematical reasoning." in *NeSy*, 2021, pp. 165–175.

[36] Z. Wu, L. Qiu, A. Ross, E. Akyürek, B. Chen, B. Wang, N. Kim, J. Andreas, and Y. Kim, "Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks," *arXiv preprint arXiv:2307.02477*, 2023.

[37] A. V. Miceli-Barone, F. Barez, I. Konstas, and S. B. Cohen, "The larger they are, the harder they fail: Language models do not recognize identifier swaps in python," *arXiv preprint arXiv:2305.15507*, 2023.

[38] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.

[39] K. Wang, H. Ren, A. Zhou, Z. Lu, S. Luo, W. Shi, R. Zhang, L. Song, M. Zhan, and H. Li, "Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning," *arXiv preprint arXiv:2310.03731*, 2023.

[40] S. Imani, L. Du, and H. Shrivastava, "Mathprompter: Mathematical reasoning using large language models," *arXiv preprint arXiv:2303.05398*, 2023.

[41] H. Luo, Q. Sun, C. Xu, P. Zhao, J. Lou, C. Tao, X. Geng, Q. Lin, S. Chen, and D. Zhang, "Wizardmath: Empowering mathematical reasoning for large language models via reinforced evol-instruct," *arXiv preprint arXiv:2308.09583*, 2023.

[42] K.-H. Huang, M. Zhou, H. P. Chan, Y. R. Fung, Z. Wang, L. Zhang, S.-F. Chang, and H. Ji, "Do lvlms understand charts? analyzing and correcting factual errors in chart captioning," *arXiv preprint arXiv:2312.10160*, 2023.

[43] K. Valmeekam, A. Olmo, S. Sreedharan, and S. Kambhampati, "Large language models still can't plan (a benchmark for llms on planning and reasoning about change)," *arXiv preprint arXiv:2206.10498*, 2022.

[44] E. La Malfa, C. Weinhuber, O. Torre, F. Lin, A. Cohn, N. Shadbolt, and M. Wooldridge, "Code simulation challenges for large language models," *arXiv preprint arXiv:2401.09074*, 2024.

[45] D. Zhang, C. Tigges, Z. Zhang, S. Biderman, M. Raginsky, and T. Ringer, "Transformer-based models are not yet perfect at learning to emulate structural recursion," *arXiv preprint arXiv:2401.12947*, 2024.