Evaluating Intermediate Reasoning of Code-Assisted Large Language Models for Mathematics

Zena Al-Khalili **Nick Howell** **Dietrich Klakow**
Saarland Informatics Campus, Saarland University, Germany
{zakhalili,nhowell,dietrich.klakow}@lsv.uni-saarland.de

Abstract

Assisting LLMs with code generation improved their performance on mathematical reasoning tasks. However, the evaluation of code-assisted LLMs is generally restricted to execution correctness, lacking a rigorous evaluation of their generated programs. In this work, we bridge this gap by conducting an in-depth analysis of code-assisted LLMs generated programs in response to math reasoning tasks, with a focus on evaluating the soundness of the underlying reasoning processes. For this purpose, we assess the generations of five LLMs, on several math datasets, both manually and automatically, and propose a taxonomy of generated programs based on their logical soundness. Our findings show that the capabilities of models significantly impact the logic implemented to solve the problem. Closed-source LLMs ground their programs in mathematical concepts, whereas open-source models often resort to unsound reasoning, relying on memorized information and exhaustive searches. Furthermore, increasing the difficulty of problems decreases sound generations for all models, revealing a critical shortcoming of LLMs on complex mathematics, contrary to what accuracy metrics suggest. Our work highlights the need for more holistic evaluations of code-assisted LLMs beyond execution accuracy metrics, toward a better understanding of LLMs' limits in the math domain.

1 Introduction

Large Language Models (LLMs) have recently achieved outstanding performance on complex reasoning tasks such as mathematical reasoning, powered by scale and multi-step reasoning approaches. Particularly, the Chain-of-Thought (CoT) (Wei et al., 2022) requires an LLM to generate the explicit reasoning steps, before generating the final answer. Despite its success, investigating CoT reasoning steps revealed critical flows of LLMs, such as committing calculation errors (Gao et al., 2023) and generating false positive chains (Lyu et al., 2023), *i.e:* containing reasoning errors yet generating correct final answers. Code-assisted reasoning approaches (Gao et al., 2023; Chen et al., 2022; Lyu et al., 2023; Gou et al., 2023; Yue et al., 2023; Das et al., 2024) proposed to solve these problems by instructing LLMs to generate programmatic reasoning steps instead, *e.g:* Python programs, and delegate their execution to an external interpreter, which ensures precise calculations and faithfulness. Such approaches have been found to further improve LLMs' performance on math tasks.

However, performance improvement is predominantly measured by the correctness of the execution outcome (Gao et al., 2023; Chen et al., 2022; Gou et al., 2023), rather than the quality of the generated programs and the underlying reasoning process.

This is problematic, as the generated programs can rely on exhaustive searches or memorized information to produce correct answers, leading to untrusted and more difficult-to-verify programs. Figure 1 shows programs generated by several LLMs using these hacks when solving math problems.

The goal addressed in this work is to evaluate the reasoning processes of code-assisted LLMs when solving mathematical tasks by analyzing their generated programs. In our evaluation, we focus on assessing the soundness of the logic governing LLMs' solutions and its impact on end performance. We also assess other aspects of the generated programs, such as API calls, complexity, and the most common errors, for a more comprehensive evaluation.

Our assessment begins by manually analyzing a subset of the generated programs, produced by GPT4o-mini, GPT4, Qwen2.5, Llama3, and StarCoder2, as they solve math problems from the ASDiv and MATH500 datasets (3.4). Given the observations from the manual analysis, we design a taxonomy reflecting the different logic types used by evaluated models (3.5). To extend the analysis to the complete set of generated programs, we employ

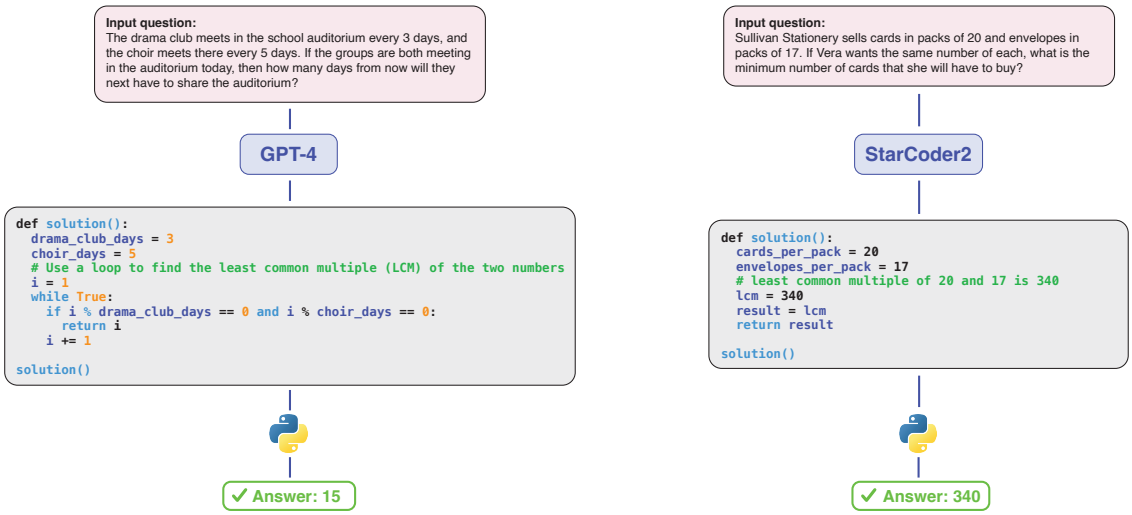


Figure 1: Program generated by GPT-4 (left) that uses a brute force search to find the answer to the input question, and StarCoder2 (right) that depends on memorized information rather than generating solution steps to solve the input question. Both input questions are from ASDiv dataset.

two automated evaluation methods: an LLM-Judge using the latest o3-mini model, and our proposed method, Code-Structure Judge, that trains a Decision Tree classifier with features extracted from programs' Abstract Syntax Tree (3.6). We find Code-Structure judge to outperform LLM-judge, with 81% accuracy against 73% (5.1); therefore, we employ it for the large-scale evaluation of all generated programs.

To the best of our knowledge, this is the first work to analyze generated programs of code-assisted LLMs on math reasoning tasks. We summarize our **findings** from Section (5.2) below:

- LLMs' capabilities influence the type of reasoning they implement to tackle a math task. GPT models and Qwen frequently generate sound programs that are grounded in math concepts, while open-source LLMs resort to unsound reasoning, exploiting memorized information or brute-force searches to find final answers.
- Difficult mathematical problems significantly decrease the distribution of sound generations, even for capable LLMs.
- Code-assisted LLMs are not consistent in the type of logic they employ to approach problems within the same math subdomain.
- Code-assisted LLMs can achieve comparable performance regardless of the type of logic they employ, hindering the trustworthiness of their generated programs.

Our in-depth evaluation highlights the need for more holistic assessment of LLMs' generations, beyond accuracy metrics, that fail to reflect models' actual capabilities and limits in the math domain.

2 Related Work

Programs as Intermediate Steps for Math Reasoning. Code-assisted reasoning approaches such as (Chen et al., 2022; Gao et al., 2023; Imani et al., 2023; Lyu et al., 2023; Das et al., 2024) prompt LLMs to generate programs instead of intermediate steps in natural language, which have been found to improve the performance of LLMs on math reasoning tasks. Beyond in-context learning approaches, other work (Gou et al., 2023; Yue et al., 2023) fine-tuned medium-scale language models such as Code-Llama (Roziere et al., 2023) on code reasoning paths and achieved comparable performance to closed-source models on math reasoning tasks. However, both in-context and fine-tuned approaches are predominantly evaluated by the correctness of the final answer, overlooking the intermediate programs and how they implement the reasoning process. This work focuses on exploring these programs to evaluate the problem-solving abilities of code-assisted LLMs accurately.

Evaluation of Intermediate Steps. This work also relates to several studies that evaluate the intermediate reasoning steps of LLMs in natural language (Golovneva et al., 2022; Hao et al., 2024; Jie et al., 2024; Li et al., 2024) targeting a multitude of evaluation dimensions, such as robustness and

faithfulness, or error identification and correction in several reasoning tasks. These works either use a human-written reference chain to compare against the evaluated chain or employ a capable LLM to localize errors and judge the quality of the reasoning chains. Code intermediate steps are unexplored; therefore, we aim to analyze the quality of these and how they affect LLMs' end performance.

Evaluation of code generated by LLMs. To assess code generated by LLMs, previous work focuses on test-based evaluation and functional correctness, such as (Chen et al., 2021; Liu et al., 2024), others (Ren et al., 2020; Eghbali and Pradel, 2022; Zhou et al., 2023b) proposed metrics to measure how similar a generation is to a reference human-written code, which is expensive to get and doesn't usually account for generation diversity. (Tong and Zhang, 2024) employed a capable LLM as a judge to localize errors in generated programs on code generation tasks. We draw inspiration from their method to evaluate generated programs, on math reasoning tasks, using an LLM-Judge. Finally, (Dou et al., 2024) analyzed characteristics of LLMs generated programs, on code generation tasks, in terms of code complexity, number of API calls, and types of errors, *i.e:* bugs, which cause programs to fail. Although insightful, the analysis concluded with a general type of error, commonly shared across different LLMs, namely *logic error*. In this work, we delve deeper into understanding the logic errors that LLMs commit when writing code to solve a math problem.

3 Evaluation of Intermediate Programs

The primary focus of this evaluation is to assess the soundness of underlying reasoning processes, *i.e:* the logic implemented in the generated programs of code-assisted LLMs. We consider a program to be logically sound if it grounds the implementation in a math concept, where a math concept refers to the principle used to solve a specific mathematical problem, *e.g:* using the Euclidean Algorithm to find the Greatest Common Divisor of two numbers. Sound programs align more with human reasoning and can be easily verified and trusted. In contrast, unsound programs are harder to verify and can't guarantee finding a solution to the given problem. Additionally, we examine the characteristics of generated programs in terms of cyclomatic complexity, types of errors, and API calls to provide a more comprehensive evaluation.

3.1 Evaluation Set-up

Given a math problem in natural language, we prompt an LLM to generate a Python program that solves the problem. In the initial analysis, we report the characteristics of the generated programs. Then, we discard the programs that fail to parse [1] to evaluate logical soundness.

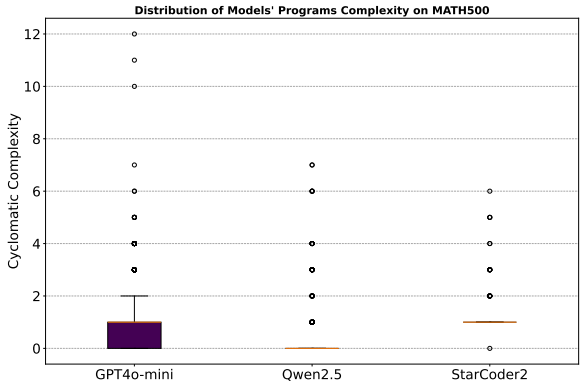
3.2 Evaluation Dataset

We evaluate LLMs on two popular math datasets, namely ASDiv (Miao et al., 2020) and MATH500 (Hendrycks et al., 2021; Lightman et al., 2023). These two sets include diverse problems ranging from grade-school to high-school competition-level questions. We exclude simple problems from the ASDiv data and focus only on more complex skills such as solving a system of equations, finding the greatest common divisor, or the least common multiple. The MATH500 set, on the other hand, has been reported to be more challenging for many LLMs (Qwen et al., 2025; Hendrycks et al., 2021), Therefore, we utilize it to investigate how LLMs modify the logic in their generated programs to tackle more complex math problems.

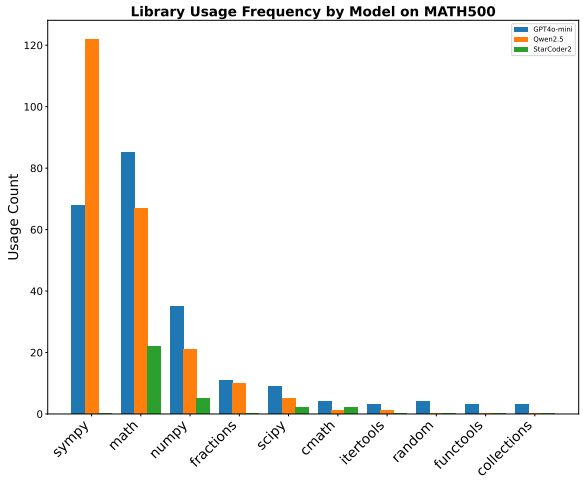
3.3 Initial Analysis: Programs Characteristics

We analyze the generated programs in terms of their API calls to relevant math libraries, cyclomatic complexity, and the most common errors they produce. Figure 2 demonstrates API calls and cyclomatic complexity of programs generated by the evaluated models on the MATH500 problems. We observe that some models exploit symbolic computations through the use of SYMPY, while others prefer numerical approaches through the MATH library. In contrast, one LLM relies much less on external dependencies, with extremely low usage counts across the board. On the other hand, the distribution of cyclomatic complexity, in Figure 2a, shows that generated programs by some models have high complexity values, indicating higher branching code that might be due to the complex conditional logic these LLMs are implementing to solve the problem. Finally, we present a list of the most common errors of programs generated on the MATH500 dataset in Appendix D.1.

[1] We resolve import errors and global misindentation (where the entire body is misindented). See Appendix A.2 for more details.



(a) Cyclomatic Complexity



(b) API calls

Figure 2: Cyclomatic complexity and API Calls of programs generated by evaluated models in response to MATH500.

3.4 Manual Analysis: Programs Logical Soundness

We conduct a detailed human-manual analysis of the generated programs to investigate the type of logic code-assisted LLMs employ when implementing their reasoning in a Python program. This analysis considers only a subset of the evaluation dataset, sized 300 programs randomly sampled from the entire set, and consists of two main steps:

Sectioning the programs We section each program into three blocks: (1) Transcription: This part of the program transcribes the information from the question into variables. (2) Processing: In this section, variables are manipulated via operations and function calls to arrive at the answer. (3) Results collection: Return the final answer of the processing section. Some sections can be combined into a single line of code that represents, for example, processing and returning results simultaneously.

Analyzing Processing Sections We inductively analyze processing code lines, considering the following aspects: logic, implementation style, coherence, verification effort, and trustworthiness. We verify that the code lines are organized into a coherent sequence of steps resembling an implementation of a mathematical concept, that a human can verify against existing implementations of the concept. Additionally, we verify that all steps of a solution are explicitly generated, rather than inferred by the model. Inferred steps might involve imprecise computations, due to LLMs' proven shortcomings in arithmetic calculations (Cobbe et al., 2021; Lewkowycz et al., 2022; Gao et al., 2023), which makes verifying and trusting these solutions harder. Finally, we check how math concepts are implemented in the generated programs. The grounded implementations can occur in several styles, including those that utilize only primitive operations for straightforward math problems, from-scratch implementations, or by calling functions from related math libraries that represent a more abstracted form of the solution. Notably, we observe other trends in some generated programs, such as relying on brute-force loops that search the space of all possible answers one by one, or lacking a processing section altogether, directly returning an answer.

3.5 Proposed Taxonomy of Programs

Given the observations from the manual analysis, we propose to categorize LLMs' generated programs into six mutually exclusive classes, three of which represent programs with logically sound and grounded reasoning, but vary in implementation style, while the other three represent ungrounded programs, with unsound reasoning, that mainly rely on memorized information or exhaustive searches to find the final answer.

1. **Conceptual** programs through library calls. Reference a math concept through calls to relevant math libraries, standard, or external.

2. **Primitive** programs are expressed in terms of the primitive operations only due to problem simplicity, where no library functionality can be called or implemented.

3. **From-scratch Implementation** of a library functionality. Instead of a call to a library function, the model implements the same functionality from scratch. The model either inlines this implementation in the generated

code or writes it as a custom function to be called when required.

4. **Brute-Force** programs that search through all possible values to find the answer without guiding the search with any math knowledge.

5. **Disorganized** programs consist of incoherent steps that seem to be a mix of the previous classes. Usually includes variables defined but not used, or the opposite.

6. **No Logic** programs skip the processing section altogether, merely returning a result without explicitly generating the steps to arrive at it. (Generating the logic as comments in natural language is also considered No Logic.)

3.6 Automated Evaluation

Extending human manual analysis to the entire evaluation dataset requires a significant amount of time and effort. Therefore, we investigate automating the evaluation by training classifiers that label generated programs using a single class from our proposed taxonomy. For this purpose, we first annotate a training set using classes from the taxonomy, then we train a decision tree classifier using features extracted from the programs. We compare the trained classifier to an LLM-Judge and evaluate both on a held-out annotated set. We pick the more accurate judge for the large-scale evaluation.

3.6.1 Training Set Annotation

To train the automated evaluation methods, two authors annotate a randomly sampled subset of the generated programs using labels from the taxonomy. The annotated set is 300 programs, 210 examples were used for training, while the rest are held out for measuring the performance of the automated judges. The annotation guidelines were developed based on observations from the manual analysis. After experimentation with different annotation schemes, we found that human assignment of per-program labels produced low inter-annotator agreement (IAA), while per-line labeling produced very high agreement. We thus selected a per-line annotation scheme, along with a simple algorithm for assigning a program-level label based on the per-line labels. More on the annotation guidelines, process, and annotators' background can be found in Appendix B.

3.6.2 Automated Evaluation Methods

Code-Structure Judge: Decision Tree Model We note that each class of the taxonomy tends to rely on specific characteristics of the code structure. Hence, we propose using features from the code structure to train a decision tree classifier that will be used to evaluate generated programs. The features to train the classifier are extracted from the Abstract Syntax Tree (AST) of each program and are the following:

- Number of function calls, including calls to functions that model writes itself.
- Number of import statements.
- Number of built-in operations. e.g: +, < , ...
- Number of control flow statements: e.g, If, for, break, ..
- Number of variables defined but not used, and number of variables used but not defined.

The max depth used for the decision tree model is 5. We experimented with other models, such as SVM and Random Forest, but found no further gains. A neural classifier, on the other hand, would require much more training data and, consequently, more annotation and human effort.

LLM-Judge Following other related work that utilizes LLMs as a judge (Tong and Zhang, 2024), we employ an LLM to assess a given program and assign a taxonomy class to that program. For this purpose, we provide a detailed description of the taxonomy classes in the prompt and ask the LLM-judge to analyze the input program carefully. Finally, we ask it to provide a single class number that best fits the program. The prompt includes no reference programs for taxonomy classes. The full prompt is provided in Appendix A.4.

We experimented with two models from OpenAI most advanced models, namely GPT4o and O3-mini, and found that O3-mini achieved slightly better results as a judge. We specify the reasoning effort of the model to be high and allow for $20,000$ max output tokens, including reasoning tokens.

4 Experimental Setup

4.1 Evaluated Models

We analyze code generations of several LLMs, including open-source models such as: StarCoder2 15B (Lozhkov et al., 2024), Llama 3.1 8B (Dubey

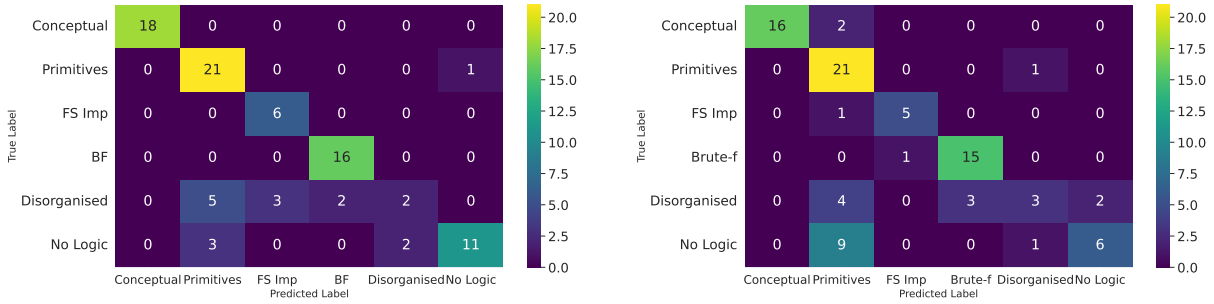


Figure 3: Confusion matrices, showing counts, of Code-Structure judge (left) and LLM-judge (right) on the held-out set. Overall accuracies are 81%, with a standard deviation of 0.005 over four different seeds, and 73% for Code-Structure judge (ours) and LLM-judge (OpenAI o3-mini), respectively. 'FS Imp': From-scratch implementation, 'BF': Brute Force.

et al., 2024), and Qwen2.5 7B (Qwen et al., 2025). We use the instruction-tuned version of these models. Furthermore, we evaluate GPT-4 (Achiam et al., 2023) and GPT4o-mini from OpenAI. All models were evaluated with greedy decoding. Implementation details are in Appendix A.1

4.2 Prompts

We use the prompt from (Gao et al., 2023) to evaluate LLMs on the ASDiv dataset, with only three demonstrations rather than eight, as no further gain was observed with the full set. For MATH500, we prompt LLMs using demonstrations from the training split of the dataset, adapted from (Gou et al., 2023). We evaluate GPT4o-mini and Qwen in zero-shot settings, since these models generated more solutions in natural language otherwise. The two full prompts can be found in Appendix A.3.

5 Results and Discussion

5.1 Comparison of Automated Evaluation Methods

To compare automated evaluation methods, we measure the agreement with human judgment using accuracy on the annotated held-out set.

Code-Structure Judge achieved a mean accuracy of 81%. Figure 3 (left) shows the confusion matrix of the Code-Structure judge on the held-out. We notice that Code-Structure judge achieves both high precision and recall on most of the classes, except for the Disorganized class with a low recall of 0.16. The Disorganized programs can be a mix of other classes, and the distinction between these can be semantic rather than structural. Appendix C.2 includes some incorrectly classified programs by the Code-Structure judge, and the table of precision, recall, and F1-score. LLM-judge, on the other hand, achieved lower accuracy with only 73%. The Primitive class has a low precision of 0.56. In contrast, the Disorganized and No Logic classes have much lower recall of 0.25 and 0.37 respectively, decreasing the overall accuracy of this judge by 8% in comparison to the Code-Structure judge. Figure 3 (right) shows the confusion matrix of LLM-judge in comparison to ground truth labels. We conducted a qualitative error analysis of some incorrectly classified instances of the No Logic class and found that the LLM-judge tends to mistake the step of transcribing information from the question to be part of the logic, and consequently classifies the instance as Primitive instead.

5.2 Evauation of Generated Programs of Code-assisted LLMs

We employ Code-Structure judge for automatically evaluating the entire set of generated programs in response to various math problems, and provide the findings below:

LLMs' capabilities significantly impact the type of implemented reasoning. The majority of programs generated by GPT models and Qwen implement logically sound reasoning to solve input problems from the ASDiv dataset. These LLMs ground their programs in mathematical concepts, utilizing numerous API calls to math libraries. Additionally, for many questions, they employ only primitive operations, given the simplicity of some problems in the ASDiv dataset. Figure 4 illustrates this, where for the above-mentioned models, the Conceptual and Primitive classes are dominant in the distribution of all programs. On the other hand, the open-source models, StarCoder2 and Llama3.1, rely much more on Primitve programs but also on ungrounded, unsound reasoning hacks to imple-

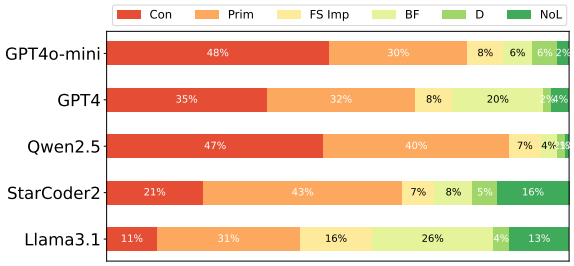


Figure 4: Distribution of programs logic for all evaluated models in percentages on ASDiv problems. Classes are: 'Con': Conceptual, 'Prim': Primitives, 'FS Imp': From scratch Implementation, 'BF': Brute-Force, 'D': Disorganized, and 'NoL': No Logic.

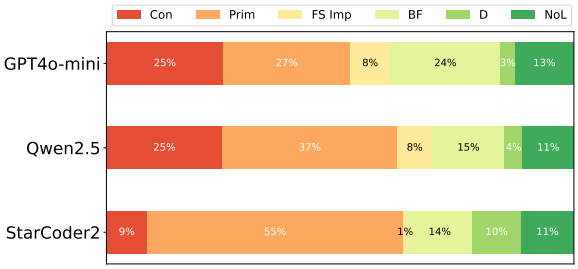


Figure 5: Distribution of programs logic for all evaluated models in percentages on the MATH500 dataset. Classes are: 'Con': Conceptual, 'Prim': Primitives, 'FS Imp': From scratch Implementation, 'BF': Brute-Force, 'D': Disorganized, and 'NoL': No Logic.

ment the solution. For instance, Llama3.1 employs math libraries in only 11% of its programs, while resorting to Brute-Force searches and memorized information, demonstrated in No Logic class, in almost 40% of its generated programs.

Complex math problems increase the generation of unsound reasoning for all evaluated LLMs. The difficulty imposed by the MATH500 dataset completely alters the type of reasoning implemented in the generated programs. This dataset is reported to be challenging for LLMs (Hendrycks et al., 2021), as it probs for skills such as Calculus, Geometry, Algebra, Probability, among others. GPT4o-mini and Qwen now generate 25% more programs with unsound, ungrounded reasoning, specifically, with exhaustive searches and programs lacking any logic, at the expense of Conceptual programs that utilize math libraries. Figure 5 demonstrates this phenomenon in the distribution of all generated programs. Upon qualitatively analyzing some programs generated by GPT-4o-mini in the No Logic class, we found that the drastic increase in the number of programs with this class can be attributed to the increase in programs with reasoning in the comments rather than code lines. We classify these instances as No Logic, because they resemble reasoning in natural language, and the code is not efficiently helping in any way to find the correct answer. The preference of textual reasoning over code might be due to problem complexity, as empirically investigated and discussed in (Chen et al., 2025), demonstrating that some GPT models prefer textual reasoning over code reasoning depending on the complexity of the task.

5.3 Further Analyses

In this section, we study the impact of sound reasoning, or its absence, on LLMs' end performance. Furthermore, we demonstrate how different math subdomains impact LLMs' preferred logic for solving the problem.

Impact of type of implemented reasoning on LLMs End-performace. To investigate the impact of the reasoning implemented in generated programs on LLMs' end-performance on math tasks, we execute the generated programs and match the execution outcome to ground truth answers, then calculate execution accuracy over programs in each logic class. Table 1 presents execution accuracy per class on all datasets. On ASDiv, we observe that sound programs, from Conceptual, Primitive, and From-Scratch Implementation classes, score slightly higher accuracy than unsound programs. However, StarCoder2 and Llama3.1 fail to follow the same trend. Qualitative analysis of their generated programs indicates that while these two LLMs employ API calls or from-scratch implementation to solve the problems, they call or implement the wrong API functionalities, causing the observed low accuracy. On the challenging dataset MATH500, Table 1 illustrates that execution accuracy of programs is on par for all logic types, sound and unsound. This is problematic, as programs with logically unsound reasoning, *i.e:*, false positives, can't be trusted or easily verified; instead, LLMs seem to hack their way to finding final answers. Finally, the reported high accuracy on the Disorganized programs is mainly due to some false positive predictions from the Code-Structure judge.

Impact of problem domain on type of implemented reasoning. Given that MATH500 ques-

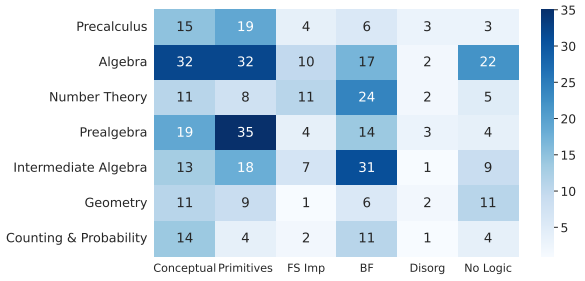
ASDiv						
Model	**Conceptual**	**Primitive**	**FS Imp**	**Brute-Force**	**Disorganized**	**No Logic**
GPT4o-mini	90%	96%	86%	77%	93%	85%
GPT4	86%	86%	100%	68%	60%	63%
Qwen2.5	91%	78%	89%	100%	60%	100%
StarCoder2	48%	44%	76%	47%	30%	61%
Llama3.1	35%	48%	37%	53%	33%	57%
MATH500						
GPT4o-mini	54%	57%	41%	46%	64%	51%
Qwen2.5	37%	59%	51%	52%	23%	56%
StarCoder2	0%	15%	0%	25%	0%	38%

Table 1: Execution (macro) accuracy in percentages per logic class for evaluated models on ASDiv (top) and MATH500 (bottom). Despite the unsound reasoning implemented in programs from Brute-Force and No Logic, high execution accuracy can still be achieved. High accuracy on Disorgnized programs is mainly due to false positive predictions from the Code-Structure judge. 'FS Imp' is From-Scratch Implementation.

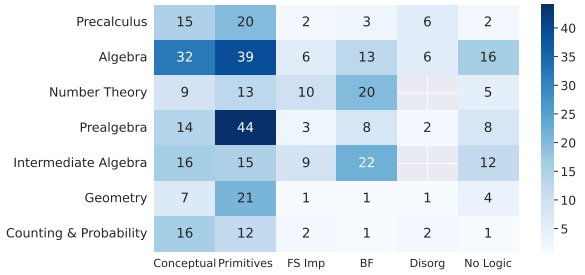
tions are annotated with the math subdomain they test for, such as Calculus, Algebra, Probability, etc, we investigate whether the evaluated LLMs consistently approach problems within a domain using the same logic. We observe that GPT4o-mini and Qwen2.5 tend to use more Primitive solutions for easier problems, such as Prealgebra. Additionally, they consistently employ Conceptual programs for Algebra problems. However, both models appear to be less consistent with the type of reasoning they employ across the rest of the subdomains, indicating higher uncertainty about the best logic to tackle the problems. Figure 6 illustrates the distribution of programs' logic per MATH500 subdomains. StarCoder2, on the other hand, heavily relies on Primitive solutions for all subdomains, demonstrating a lack of diversity in the logic it employs for different types of math problems.

6 Conclusion

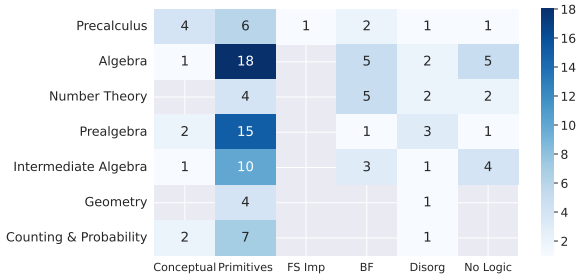
In this work, we conducted an in-depth analysis of code-assisted LLMs generated programs in response to math problems. Our assessment focuses on evaluating the logical soundness of underlying reasoning processes implemented in the generated programs. We categorized the generated programs into six different categories, which make up our proposed taxonomy of logical soundness. Three of which represent sound reasoning that ground the programs in verifiable math concepts. In contrast, the other three exploit memorized information or exhaustive searches to find final answers. We an-



(a) GPT4o-mini



(b) Qwen2.5



(c) StarCoder2

Figure 6: Distribution of programs logic, showing counts, per math subdomain in MATH500. Classes: 'FS Imp': From-Scratch Implementation, 'BF': Brute-Force, 'Disorg': Disorganized.

notated a subset of programs using classes from the taxonomy and trained a decision-tree classifier, which we call the Code-Structure Judge. The Code-Structure judge outperformed an LLM-judge baseline and was employed for a large-scale evaluation of more generated programs. Our findings show that the capabilities of LLMs and the difficulty of the problem impact the type of reasoning implemented, yet regardless, LLMs can exploit unsound reasoning to achieve comparable accuracy. Our work underscores the importance of a comprehensive evaluation of code-assisted LLMs. We hope our findings inspire future work to further study why LLMs employ ungrounded solutions and how to mitigate this phenomenon.

Limitations

We note there are limitations to our work. First: Code-structure Judge is still not accurate on the Disorganized class, causing many false positives. Second, we depend on one prompting technique and don't compare performance when utilizing prompts to improve generated programs with further refinement, such as Self-Refine (Madaan et al., 2024) or Code-based Self-Verification (Zhou et al., 2023a).

Acknowledgments

The authors would like to thank Vagrant Gautam for their mentorship and useful feedback. We are also grateful to Marius Mosbach and Ellie Pavlick for feedback during the early stages of this work, to Miaoran Zhang for the valuable discussions, and to Tural Mammadov for help with the experimental infrastructure. We also thank anonymous reviewers for feedback. The authors received funding from the DFG (German Research Foundation) under project 232722074, SFB 1102.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*.

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

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*.

Yongchao Chen, Harsh Jhamtani, Srinagesh Sharma, Chuchu Fan, and Chi Wang. 2025. Steering large language models between code execution and textual reasoning. In *The Thirteenth International Conference on Learning Representations*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *ArXiv*, abs/2110.14168.

Debrup Das, Debopriyo Banerjee, Somak Aditya, and Ashish Kulkarni. 2024. Mathsensei: A tool-augmented large language model for mathematical reasoning. *arXiv preprint arXiv:2402.17231*.

Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, Yan Liu, Enyu Zhou, Ming Zhang, Yuhao Zhou, Yueming Wu, Rui Zheng, Ming Wen, Rongxiang Weng, Jingang Wang, Xunliang Cai, Tao Gui, Xipeng Qiu, Qi Zhang, and Xuanjing Huang. 2024. What's wrong with your code generated by large language models? an extensive study. *Preprint*, arXiv:2407.06153.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*. License: Llama 3 Community License Agreement.

Aryaz Eghbali and Michael Pradel. 2022. Crystalbleu: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Olga Golovneva, Moya Chen, Spencer Poff, Martin Corredor, Luke Zettlemoyer, Maryam Fazel-Zarandi, and Asli Celikyilmaz. 2022. Roscoe: A suite of metrics for scoring step-by-step reasoning. *arXiv preprint arXiv:2212.07919*.

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

Shibo Hao, Yi Gu, Haotian Luo, Tianyang Liu, Xiyan Shao, Xinyuan Wang, Shuhua Xie, Haodi Ma, Adithya Samavedhi, Qiyue Gao, et al. 2024. Llm reasoners: New evaluation, library, and analysis of step-by-step reasoning with large language models. *arXiv preprint arXiv:2404.05221*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. *arXiv preprint arXiv:2103.03874*.

Shima Imani, Liang Du, and Harsh Shrivastava. 2023. Mathprompter: Mathematical reasoning using large language models. *arXiv preprint arXiv:2303.05398*.

Yeo Wei Jie, Ranjan Satapathy, Goh Siow Mong, Erik Cambria, et al. 2024. How interpretable are reasoning explanations from prompting large language models? *arXiv preprint arXiv:2402.11863*.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. 2022. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35:3843–3857.

Xiaoyuan Li, Wenjie Wang, Moxin Li, Junrong Guo, Yang Zhang, and Fuli Feng. 2024. Evaluating mathematical reasoning of large language models: A focus on error identification and correction. *arXiv preprint arXiv:2406.00755*.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *Preprint*, arXiv:2305.20050.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

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

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. *arXiv preprint arXiv:2301.13379*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Shen-yun Miao, Chao-Chun Liang, and Keh-Yih Su. 2020. A diverse corpus for evaluating and developing English math word problem solvers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 975–984, Online. Association for Computational Linguistics. License: CC-BY-NC 4.0.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*. License: Llama 2 Community License Agreement.

Weixi Tong and Tianyi Zhang. 2024. CodeJudge: Evaluating code generation with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 20032–20051, Miami, Florida, USA. Association for Computational Linguistics.

Lyz `lyz@riseup.net`. autoimport. License: GPL-3.0.

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

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. 2023. Mammoth: Building math generalist models through hybrid instruction tuning. *arXiv preprint arXiv:2309.05653*.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, et al. 2023a. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. *arXiv preprint arXiv:2308.07921*.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023b. Codebertscore: Evaluating code generation with pretrained models of code. *arXiv preprint arXiv:2302.05527*.

A Experimental Details

A.1 Implementation Details

We use `starcoder2-15b-instruct`, `Meta-Llama-3-8B-Instruct`, `Qwen2.5-7B-Instruct` from HuggingFace (Wolf et al., 2020). For OpenAI models we use `gpt-4-32K` and `gpt-4o-mini` We use int8bit model quantization for all models except OpenAI models as we observe no major differences in the execution accuracy per model. Finally, we use NVIDIA A100 GPUs 40GB, with batch size = 32 for evaluating Llama3 and StarCoder2 and Qwen2.5 on the ASDiv, which took around one hour. While evaluating these models on the MATH subset took about 3 hours with batch size=16.

A.2 Resolving some execution errors

To resolve import errors we used `autoimport` (`lyz@riseup.net`). For indentation errors we used the `inspect` module from the Python standard library as a post processing step.

A.3 PAL Prompt and MATH Prompt

In our experiments, we employ PAL prompt from (Gao et al., 2023) to evaluate LLMs on the ASDiv data, however we only include three demonstrations out of eight, as we observe no further gains from including the entire set. The full prompt is in Figure 7.

Th prompt employed for evaluating LLMs on the MATH data is found in Figure8

A.4 LLM-Judge Prompt

Figure 9 present the prompt used to prompt o3-mini as an LLM-Judge

B Training Set Annotation Guidelines and Annotators Background

Double independent annotation was performed on 50 of the 300 samples, with 93% line-granularity and 100% sample-granularity agreement. Disagreements were then adjudicated, and guidelines were updated to resolve the ambiguities.

B.1 Annotation format

Each line of program code is prefixed with five characters of the form `TPIR` plus space.

Columns:

`T`. transcription

`P1-5`. processing (blank implies "no logic")

`I`. inference-time computation

`R`. result collection

B.2 Transcription

Purely transcriptive statements that transcribe either data (e.g. numbers) or relations between data (e.g. equations) from the question.

Purely transcriptive statements can still contain operations that are explicitly described in the problem statement; these operations do not count as processing.

```
      """ One number is twice as
      large as another. The smaller
      number is 3. Find the other
      number """
      def solution():
T       x = 3
T  R    y = 2*x
   R    return y
```

Even if the model uses the question to compute at inference time, without pure transcription of data or relations no `T` is marked:

```
     """
     One number is twice as large
     as another. The sum of the
     numbers is 12. Find the gcd of
     the two numbers
     """
     def solution():
  I    x = 4
  I    y = 8
 1 R   z = math.gcd(x, y)
   R   return z
```

If the model has performed partial computations (collapsing e.g. a pure transcription and true processing) then it receives no `T`; it can possibly receive other labels.

```
     """
     Two boards have total length
     10; the long board is 2 longer
     than the short board. Find the
     lengths of the two boards
     """
     def solution():
T      total_length = 10
T      difference = 2
 2IR   short_length = (total_length
         difference) / 2
T  R   long_length = short_length +
         difference
   R   return short_length, long_length
```

B.3 Processing types

Comments are never marked as anything!!

1. conceptual lib
   - calls a library function later
   - mark the entire function; but not the comments
2. primitive
   - uses primitives in a way that is correct, and cannot be simplified by lib
3. from scratch implementation
   - plausibly correct, looks like inlined implementation of a lib function
   - writes a function for itself, then calls (mark the call as implementation as well.)
4. brute-force
   - search through all space
   - mark the entire loop
5. disorganised
   - probably incorrect (more or less random?, disorganised)

Empty 'processing type' field (i.e., a space " ") indicates that no processing occurs.

B.4 Inference-time computation

If the model skips steps (either entirely, or in part (in which case there will still be a processing label)) Then the line gets the "inference-time computation" flag

In the case of the model entirely precomputing, the processing type will probably be empty and the `I` flag will appear after.

```
     def solution():
  I    x = 6 # lcm of 3 and 2
       return x
```

If the model has performed symbolic manipulation to avoid the use of symbolic equation libraries, it receives `I` as processing/transcription labels.
–

B.5 Result collection

Lines that return the final answer, or that store the variable holding the final answer, are marked `R` in the final column.

```
     def solution():
  IR   x = 6 # lcm of 3 and 2
   R   return x

     """ One number is twice as
     large as another. The sum of
     the numbers is 12.  Find the
     gcd of the two numbers"""
     def solution():
  I    x = 4
  I    y = 8
 1 R   z = math.gcd(x, y)
   R   return z
```

B.6 Annotators Background

The annotation process of the generated programs is time-consuming and isn't feasible to do at a larger scale because it requires expertise with code understanding. Both annotators are experts in computer science and have done prior reading on programming languages and related topics, that make them a better fit for the annotation process.

Furthermore, programmatic steps and code structure patterns are less subjective than natural language, carrying less nuance, which can lead to disagreement or bias. For example, it is hard to mistake a library call from a brute-force program.

C More Results on Code-Structure Judge and LLM-Judge Performance

C.1 Per-class F_1 scores

Table 2 provides precision, recall, and F1 scores of both code judges.

C.2 Examples of incorrectly classified programs

We provide a few examples of programs where Code-Structure judge misclassifies Disorganized programs and their true label in Figure 10.

D Further Analysis of Generated Programs

D.1 Most common bugs in the generated programs on MATH500 dataset

StarCoder2 generated 60 programs with bugs: 23 of them with undefined symbols and 15 that were calling undefined functionality from libraries. e.g, calling 'log3' from the math library. Qwen2.5 generated 40 programs with bugs as follows: 8 were with undefined symbols, 7 with undefined attributes of objects e.g, a float object has no attribute denominator. GPT4o-mini generated only 5 programs with bugs, two of which have undefined symbols.

	Precision	**Recall**	$\mathbf{F_1}$
Conceptual	1.	1.	1.
Primitive	0.72	0.95	0.82
From-scratch Implementation	0.66	1.	0.8
Brute-Force	0.88	1.	0.94
Disorganized	0.5	0.16	0.25
No Logic	0.91	0.68	0.78
	Precision	**Recall**	$\mathbf{F_1}$
Conceptual	1.	0.88	0.94
Primitive	0.56	0.95	0.71
From-scratch Implementation	0.83	0.83	0.83
Brute-Force	0.83	0.93	0.88
Disorganized	0.6	0.25	0.35
No Logic	0.75	0.37	0.5

Table 2: Precision, recall, and F_1 of the Code-Structure Judge (top) and LLM-Judge (bottom)

Figure 7: PAL prompt used to prompt evaluated models with three demonstrations on the ASDiv dataset.

System: Write python program to solve math problems.

User: # Q: Michael had 58 golf balls. On tuesday, he lost 23 golf balls. On wednesday, he lost 2 more. How many golf balls did he have at the end of wednesday?

Assistant:

```
# solution in Python:
def solution():
    """Michael had 58 golf balls. On tuesday, he lost 23 golf balls.
    On wednesday, he lost 2 more. How many golf balls did he have at
    the end of wednesday?"""
    golf_balls_initial = 58
    golf_balls_lost_tuesday = 23
    golf_balls_lost_wednesday = 2
    golf_balls_left = golf_balls_initial \
            - golf_balls_lost_tuesday \
            - golf_balls_lost_wednesday
    result = golf_balls_left
    return result
```

User: # Q: Olivia has $23. She bought five bagels for $3 each. How much money does she have left?

Assistant:

```
# solution in Python:
def solution():
    """Olivia has $23. She bought five bagels for $3 each. How much
    money does she have left?"""
    money_initial = 23
    bagels = 5
    bagel_cost = 3
    money_spent = bagels * bagel_cost
    money_left = money_initial - money_spent
    result = money_left
    return result
```

User: # Q: There were nine computers in the server room. Five more computers were installed each day, from monday to thursday. How many computers are now in the server room?

Assistant:

```
# solution in Python:
def solution():
    """There were nine computers in the server room. Five more
    computers were installed each day, from monday to thursday. How
    many computers are now in the server room?"""
    computers_initial = 9
    computers_per_day = 5
    num_days = 4  # 4 days between monday and thursday
    computers_added = computers_per_day * num_days
    computers_total = computers_initial + computers_added
    result = computers_total
    return result
```

Figure 8: Prompt structure for evaluating on MATH dataset.

System: `Write python programs to solve math problems`

User: `# Q: Find the coefficient of` x^3 `when` $3(x^2 - x^3 + x) + 3(x + 2x^3 - 3x^2 + 3x^5 + x^3) - 5(1 + x - 4x^3 - x^2)$ `is simplifie.`

Assistant:

```
# solution in Python:
from sympy import symbols, simplify
def solution():
    x = symbols('x')
    expr = 3*(x**2 - x**3 + x) + 3*(x + 2*x**3 - 3*x**2 + 3*x**5 + x**3) -
      5*(1 + x - 4*x**3 - x**2)
    simplified_expr = simplify(expr)
    x3_coefficient = simplified_expr.as_coefficients_dict()[x**3]
    result = x3_coefficient
    return result
```

User: `# Q: The surface area of a sphere with radius r is` $4\pi r^2$`. Including the area of its circular base, what is the total surface area of a hemisphere with radius 6 cm? Express your answer in terms of pi`

Assistant:

```
# solution in Python:
import math
def solution():
    radius = 6
    # Surface area of the hemisphere
    hemisphere_area = 2 * math.pi * radius**2
    # Area of the circular base
    base_area = math.pi * radius**2
    # Total surface area
    total_surface_area = hemisphere_area + base_area
    result = total_surface_area / math.pi
    return result
```

User: `# Q: Monica tosses a fair 6-sided die. If the roll is a prime number, then she wins that amount of dollars (so that, for example, if she rolls 3, then she wins 3 dollars). If the roll is composite, she wins nothing. Otherwise, she loses 3 dollars. What is the expected value of her winnings on one die toss? Express your answer as a dollar value to the nearest cent.`

Assistant:

```
# solution in Python:
def solution():
    # Probabilities of each outcome
    prime_prob = 1 / 6
    composite_prob = 1 / 3
    otherwise_prob = 1 / 6
    # Expected value of each outcome
    prime_expected_value = (2 * prime_prob) + (3 * prime_prob) + (5 * prime_prob)
    composite_expected_value = 0 * composite_prob
    otherwise_expected_value = -3 * otherwise_prob
    # Total expected value
    total_expected_value = prime_expected_value + composite_expected_value +
      otherwise_expected_value
    # Dollar value to the nearest cent
    result = "{:.2f}".format(total_expected_value)
    return result
```

Figure 9: The prompt for the LLM-Judge task with o3-mini

User:

```
You are an expert code judge that analyze code based on the
following classes:
 1- Conceptual through library calls. Reference a math concept
 through calls to relevant math libraries, standard or external.
 2- primitive solution: programs are expressed in terms of the
 primitive operations only due to problem simplicity, where no
 library functionality can be called or implemented.
 3- From-scratch Implementation of a library functionality.
 Implements a library functionality from scratch. implementation
 is inlined in the generated code, or can be a custom function
 to be called when required.
 4- Brute-Force. The program search through all possible values
 to find the answer without guiding the search with some math
 knowledge.
 5- Disorganized: the program consists of incoherent steps that
 seem to be a mix of the previous classes. Usually include
 variables used but not defined or the opposite.
 6- No Logic: These programs merely return a result without
 explicitly generating the steps to arrive at it, transcribing
 information from the question only without further processing
 the information is also No logic. generating the logic as
 comments doesn't count either.


Instructions:
- given an input program your task is to analyze it and then
provide a class number from the list above.
- don't fix the code.
- Put your final answer in \boxed{}.
```

```
def find_other_number():
    difference = 100
    one_number = 91
    other_number = one_number + difference
    return other_number
result = find_other_number()
```

True label: "Primitive"

```
def solution():
    cards_per_pack = 20
    envelopes_per_pack = 17
    # least common multiple of 20 and 17 is 340
    lcm = 340
    result = lcm
    return result
solution()
```

True label: "No Logic"

Figure 10: Instances that were labeled "Disorganized" by the Code-Structure Judge, and their true label.