SymCode: A Neurosymbolic Approach to Mathematical Reasoning via Verifiable Code Generation

Sina Bagheri Nezhad[1,2]**, Yao Li**[1]**, Ameeta Agrawal**[1]
[1]Portland State University, Portland, USA
[2]ElastixAI, Seattle, USA
{sina5, liyao, ameeta}@pdx.edu

Abstract

Large Language Models (LLMs) often struggle with complex mathematical reasoning, where prose-based generation leads to unverified and arithmetically unsound solutions. Current prompting strategies like Chain of Thought still operate within this unreliable medium, lacking a mechanism for deterministic verification. To address these limitations, we introduce SymCode, a neurosymbolic framework that reframes mathematical problem-solving as a task of verifiable code generation using the SymPy library. We evaluate SymCode on challenging benchmarks, including MATH-500 and OlympiadBench, demonstrating significant accuracy improvements of up to 13.6 percentage points over baselines. Our analysis shows that SymCode is not only more token-efficient but also fundamentally shifts model failures from opaque logical fallacies towards transparent, programmatic errors. By grounding LLM reasoning in a deterministic symbolic engine, SymCode represents a key step towards more accurate and trustworthy AI in formal domains.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in natural language, yet their proficiency in domains requiring rigorous, multi-step formal reasoning, such as advanced mathematics, remains a significant challenge (Wei et al., 2022; Ahn et al., 2024). When prompted to solve complex math problems, LLMs that reason in prose often generate solutions that are unreliable, containing subtle arithmetic errors, logical fallacies, or hallucinated intermediate steps. Furthermore, these natural language rationales are often opaque; their convoluted structure can obscure the reasoning path, making it difficult for even a domain expert to verify their correctness. This lack of a clear, deterministic failure signal also makes it challenging to create an automated feedback loop to iteratively refine an incorrect answer.

Current approaches to mathematical reasoning broadly fall into two categories: inference-time prompting and model fine-tuning. Inference-time methods like Chain of Thought (CoT) (Wei et al., 2022) and Tree of Thoughts (ToT) (Yao et al., 2023) have improved performance by encouraging models to articulate their reasoning. These methods, while accessible, inherit the weaknesses of natural languages. Training-based methods, on the other hand, can improve a model's intrinsic capabilities but require significant computational resources and large, high-quality datasets, and may not generalize well to novel problems.

To overcome these critical shortcomings, we introduce **SymCode**, a neurosymbolic framework that reframes mathematical problem-solving for any class of problems that can be formalized programmatically. Instead of prompting an LLM to describe its reasoning in prose, SymCode instructs it to construct a verifiable, executable *Python script* where the code serves as the reasoning trace. Unlike prior work like Program-Aided Language Models (PAL) (Gao et al., 2023), which uses code as an external calculator for intermediate steps, SymCode treats the *entire program* as the final, self-contained reasoning artifact. This elevates the LLM's role from a simple calculator to an expert translator, converting a natural language problem into a formal, verifiable script.

The SymCode framework orchestrates the strengths of three components to address the challenges of prose-based reasoning. As illustrated in Figure 1, the process begins with instructing an LLM to interpret the problem and generate a Python script that leverages SymPy, a computer algebra system (CAS) that manipulates mathematical expressions in their exact symbolic form, thereby eliminating arithmetic errors. Next, the generated script is executed in a sandboxed Python interpreter, which provides a deterministic pass/fail signal for programmatic verification. Finally, a self-
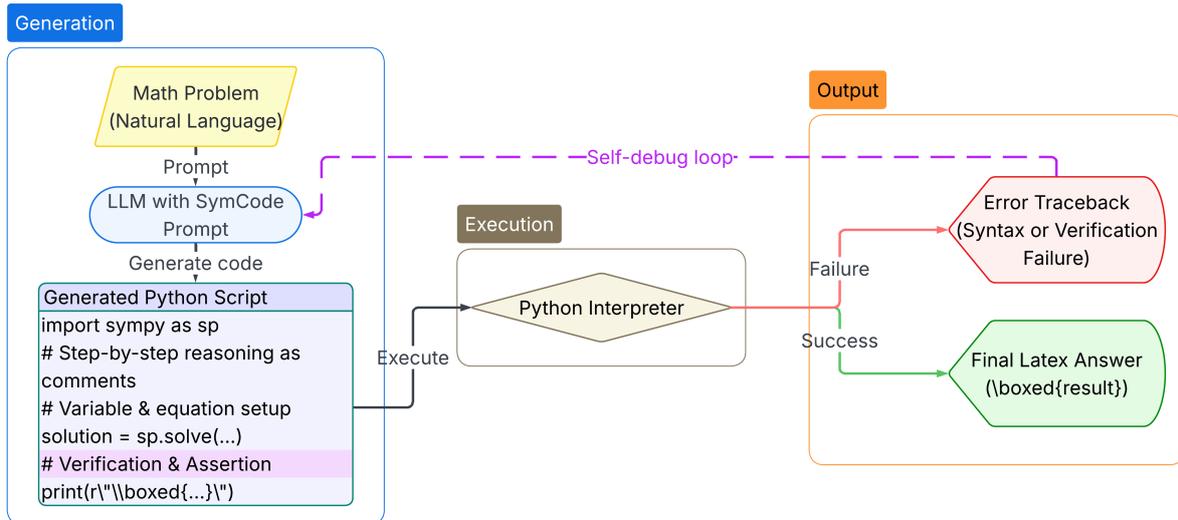
Figure 1: Overview of the SymCode framework. A natural language problem is translated into Python code by the LLM, executed, and iteratively refined through error feedback until successful execution or a retry limit is reached.

debugging loop feeds any execution error back to the LLM, enabling it to iteratively correct its own code. This entire process creates a transparent, auditable, and self-correcting reasoning trace that is both human-readable and machine-executable.

Our work is guided by two main research questions:

1. To what extent does reframing mathematical reasoning as verifiable code generation improve a system's accuracy on complex mathematical problems compared to established prose-based prompting techniques?

2. Beyond accuracy, how does this neurosymbolic approach alter the characteristics of the LLM's reasoning process, specifically concerning token efficiency and the fundamental nature of its failure modes?

Our main contributions are as follows:

- We introduce and formalize **SymCode**, a prompt-based framework for advanced mathematical reasoning that transforms an LLM into a neurosymbolic reasoner generating self-contained, verifiable Python scripts.

- Framing mathematical reasoning as code generation enables us to apply an iterative self-debugging mechanism where LLM uses deterministic interpreter feedback to correct its own errors, a robust verification process not available to prose-based reasoners.

- Through extensive experiments on three challenging mathematical benchmarks—MATH-500, OlympiadBench, and AIME—we show that SymCode improves accuracy by up to 13.6 percentage points (and up to 16.8 with SymCode+) over traditional prompting baselines, with the performance gap increasing as problem difficulty rises.

- We provide a detailed analysis showing that reasoning-as-code is substantially more token-efficient than prose-based methods.

2 Related Work

The use of code for mathematical reasoning is well-established, from the computer-assisted proof of the four-color theorem (Appel and Haken, 1989) to modern formal proof assistants like Lean (Moura and Ullrich, 2021). Historically, however, these powerful symbolic systems required expert human effort to manually translate natural language problems into formal specifications. The recent challenge of automating this translation and equipping LLMs with robust mathematical abilities has spurred a variety of research directions, which can be broadly categorized into three main areas: training-based methods that modify model weights, training-free strategies that operate at inference-time, and methods that augment LLMs with external tools.

A significant line of research focuses on enhancing reasoning by fine-tuning models on specialized data or with reinforcement learning. For instance,

Method	Reasoning Modality	External Tools	Self-Correction	Verification
Training-Based Methods (Model-Tuning)				
LeDex 2025	Code	Interpreter Feedback	Yes (Learned)	Execution-based
rStar-Math 2025	Hybrid (Code+)	MCTS+Code	Yes (Self-Evolution)	Process Reward Model
Training-Free Methods (Inference-Time)				
CoT 2022	Natural Language	None	No	No
ToT 2023	Natural Language	None	Branching	None (NL-based vote)
PAL 2023	Code	Python Interpreter	No	Result Only
MATHSENSEI 2024	Hybrid	Search/Prog/Solver	No (has code refiner)	Tool-based
NSAR 2025	Hybrid	Python/Symbolic	No	Result Only
SymCode+ (Ours)	**Code**	**SymPy (CAS)**	**Yes (Self-Debug Loop)**	**Constraints and Result**

Table 1: Comparison of SymCode+ with key LLM-based mathematical reasoning methods, grouped into training-free inference-time approaches and training-based fine-tuning approaches. SymCode is a training-free method focused on verifiable `SymPy` code generation with a self-debugging loop. Abbrev.: CAS = Computer Algebra System, MCTS = Monte Carlo Tree Search.

`rStar-Math` employs Monte Carlo Tree Search (MCTS) guided by a process reward model to create a "deep thinking" process, fine-tuning smaller models to achieve strong performance (Guan et al., 2025). These training-based methods represent a powerful but distinct paradigm focused on improving the model's internal capabilities.

Another category of methods aims to improve reasoning without altering the model's parameters, focusing instead on structuring the generation process at inference-time. Early efforts in this area focused on eliciting more structured thought processes. The seminal Chain of Thought (CoT) prompting method demonstrated that instructing a model to "think step-by-step" significantly improves performance (Wei et al., 2022). This concept was further generalized by approaches like Tree of Thoughts (ToT), which allows models to explore multiple reasoning paths concurrently (Yao et al., 2023), and decomposition techniques that break complex problems into simpler sub-problems (Khot et al., 2023). More recently, this paradigm has shifted towards scaling test-time compute, a strategy popularized by OpenAI's o1 model (OpenAI et al., 2024). This has led to methods like *budget forcing*, where a model's thinking process is deliberately extended to encourage deeper exploration (Muennighoff et al., 2025). While effective, these methods still largely operate within the domain of natural language, making their reasoning chains prone to arithmetic errors and logical inconsistencies without a formal method for verification (Ahn et al., 2024).

To overcome the unreliability of prose-based computation, a third line of work has focused on augmenting LLMs with external tools, particularly code interpreters. Program-Aided Language Models (PAL) (Gao et al., 2023) pioneered this approach by prompting an LLM to generate an executable program, offloading computation to a reliable interpreter. This has been extended with models like MATHSENSEI, which integrates multiple tools including web search and symbolic solvers (Das et al., 2024). Our work represents a fundamental shift from this paradigm. Rather than using code for mere computation, we use it to change the reasoning modality itself. SymCode instructs the LLM to translate a problem into a formal, *symbolic representation*, which is then manipulated by a Computer Algebra System (CAS). This elevates the task from executing a sequence of arithmetic steps to solving a system of symbolic equations. In essence, PAL uses code for calculation, whereas SymCode uses code for formal mathematical reasoning. This directly operationalizes a true neurosymbolic approach by bridging neural language interpretation with the rigorous logic of symbolic systems (Kautz, 2022; Fang et al., 2024; Nezhad and Agrawal, 2025).

The use of code also enables robust self-correction mechanisms. Several methods train models to debug their own code, either through fine-tuning on datasets of errors and corrections (Jiang et al., 2025) or by using reinforcement learning to refine outputs based on execution feedback (Kumar et al., 2024). While some studies suggest that Reinforcement Learning with Verifiable Rewards (RLVR) (Lambert et al., 2025) primarily amplifies existing capabilities rather than creating new ones (Yue et al., 2025), the iterative self-debugging

loop in `SymCode` provides a concrete, inference-time mechanism for refinement.

Our approach also differs from general-purpose code generation, where models produce applications from specifications (e.g., text-to-SQL) (Zan et al., 2023). In those tasks, the code is the final product. In contrast, we employ code as the *reasoning modality itself*—a transparent, intermediate representation of logic.

To highlight the distinctions and contributions of our approach, Table 1 provides a comparative overview of key related methods in mathematical reasoning.

3 The SymCode Framework

The SymCode framework adapts an LLM from a probabilistic text generator into a structured, neurosymbolic reasoner. While prior work has used code as an external computational tool, the fundamental principle of SymCode is to treat the entire reasoning process as the act of generating a verifiable program where the code *is* the reasoning trace. The motivation for this shift is to overcome the inherent limitations of prose-based reasoning, which is often ambiguous, prone to subtle logical and arithmetic errors, and lacks a mechanism for automated verification. Instead of asking the LLM to explain its thinking, we instruct it to write a program that *enacts* that thinking. This methodology leverages the respective strengths of neural and symbolic systems: the LLM excels at interpreting the nuances and context of the problem statement, while the Python interpreter, coupled with the `SymPy` library (Meurer et al., 2017), provides rigor for the formal mathematical manipulations. First, we use a specialized prompt to guide the LLM in structuring its reasoning as a self-contained Python script that leverages the `SymPy` library for deterministic computation (Section 3.1). Second, to enhance accuracy, we introduce an iterative self-debugging loop that enables the model to correct its own programmatic errors based on interpreter feedback (Section 3.2). To provide a concrete illustration of the framework in action, from problem statement to the final generated script, see the full example in Appendix A.

3.1 SymCode

The complete SymCode prompt template is shown below.

```
You are an expert mathematical
    reasoner. Your output must be ONLY
    a single Python code block fenced
    as ```python ... ``` with no
    prose before or after.
Inside that single Python script:
1. Import SymPy with `import sympy as
    sp`
2. Add explicit step-by-step reasoning
    as comments throughout your code
3. Document the problem setup:
   - Clearly identify variables,
   constraints, and goals in comments
   - Define symbols with appropriate
   assumptions (e.g., sp.symbols('x',
   positive=True, integer=True))
4. Include intermediate reasoning
   steps:
   - Each step should have a comment
   explaining the mathematical
   reasoning
   - Use meaningful variable names
   that reflect their purpose
   - Show the algebraic manipulations
   clearly
5. For verification:
   - Substitute solutions back into
   original equations
   - Check domain constraints (e.g.,
   integer solutions, positive values
   )
   - Filter invalid solutions
6. Print ONLY the final answer in
   LaTeX boxed form:
   print(r"\boxed{}".format(
   final_answer))
# PROBLEM
{problem_text}
# END PROBLEM
```

Listing 1: The SymCode Prompt Template.

Each component of this prompt serves a distinct purpose in structuring the model's output.

Symbolic Formulation. The explicit instruction to use `import sympy as sp` is critical. `SymPy` is a Python library for symbolic mathematics, acting as a Computer Algebra System (CAS). Unlike standard numerical libraries that work with approximate floating-point numbers, `SymPy` manipulates mathematical expressions in their exact, symbolic form (e.g., representing $\sqrt{2}$ precisely rather than as 1.414...). This brings two key advantages: first, it prevents the accumulation of rounding errors that can invalidate multi-step calculations. Second, it enables true algebraic reasoning by allowing the script to programmatically solve equations, simplify expressions, and apply mathematical rules with perfect fidelity. This moves the task from error-prone prose-based calculation to exact, verifiable computation.

Dataset	Problem Statement	Ground Truth	SymCode	CoT	ToT
MATH-500	How many distinct values can be obtained from the expression $2 \cdot 3 \cdot 4 \cdot 5 + 1$ by inserting any number of parentheses?	4	4	1	2
OlympiadBench	In $\triangle ABC$, $AB = 4$, $BC = 6$, $AC = 8$. Squares $ABQR$ and $BCST$ are drawn external to the triangle. Compute the length of QT.	$2\sqrt{10}$	$2\sqrt{10}$	$2\sqrt{2}\sqrt{10 + 3\sqrt{15}}$	12
AIME	Let $\triangle ABC$ have circumcenter O and incenter I with $\overline{IA} \perp \overline{OI}$, circumradius 13, and inradius 6. Find $AB \cdot AC$.	468	468	156	338

Table 2: Sample problems from the evaluation datasets, with outputs from SymCode and prose-based baselines. SymCode correctly solves all three, while the baselines produce incorrect answers due to logical or arithmetic errors.

Interpretability. By requiring 'step-by-step reasoning as comments', we retain the "show your work" benefit of Chain of Thought while grounding it in a formal, code-based structure. This makes the model's logic transparent and auditable for human experts.

Problem Scaffolding. The requirement to define symbols with appropriate assumptions (e.g., 'sp.symbols('x', positive=True, integer=True)') forces the model to formalize the problem's constraints upfront. This structured setup significantly reduces the solution search space and helps prevent the generation of invalid solutions later in the process.

Verification and Filtering. This is a cornerstone of the framework's reliability. The prompt requires the model to insert `assert` statements into its generated code. These statements check key conditions at runtime, such as whether a solution satisfies the original problem constraints or adheres to domain requirements (e.g., ensuring a length variable is positive). If an assertion fails, it raises an error that halts execution. This provides a deterministic failure signal that effectively filters out incorrect solution paths and can trigger the self-debugging loop of SymCode+ (described next), enabling a crucial self-correction step. If the script runs to completion without any exceptions, the final answer is reported by capturing the script's output, which the prompt requires to be printed in a \boxed{} LaTeX format. The effectiveness of this verification, however, is contingent on the quality of the assertions generated by the LLM; an absence of failure does not guarantee correctness if the assertions are weak or located in an unexecuted code path.

3.2 SymCode+: Self-Debugging Loops

To further enhance the accuracy of the framework, we extend the core prompt with an agentic wrapper called **SymCode+**. This extension introduces an iterative self-debugging loop that allows the LLM to correct its own programmatic mistakes.

If the initial script fails during execution, the loop is activated. A failure can be a programmatic *exception* (e.g., SyntaxError, TypeError) or a *verification failure* where an internal AssertionError is raised. The captured error message and traceback are then appended to the prompt history, and the LLM is instructed to "debug the following code based on the provided error message." This cycle of execution, failure, and correction repeats until the script runs successfully or a preset iteration limit (e.g., 2–3 attempts) is reached.

4 Experimental Setup

We consider challenging mathematical datasets and a diverse set of LLMs in our evaluation.

4.1 Datasets

We evaluate SymCode on three widely used, challenging benchmarks that require multi-step mathematical reasoning, spanning difficulty from high school competitions to Olympiad-level problems:
(1) MATH-500, a 500-problem subset of the MATH dataset, comprising challenging problems from high school mathematics competitions, covering topics like algebra, geometry, number theory, and precalculus (Lightman et al., 2023);
(2) OlympiadBench, 674 text-only English math problems from national and international olympiads requiring creative and formal reasoning (He et al., 2024); and
(3) American Invitational Mathematics Exam-

ination (AIME) 2024 & 2025, 60 (30 from each year) short-answer problems from recent competitions that bridge high school and olympiad difficulty (Mathematical Association of America, 2024). Table 2 shows a sample from each dataset.

4.2 Models and Baselines

We evaluate SymCode across several state-of-the-art language models such as Llama 3.2 (90B)(Grattafiori et al., 2024), GPT-5-nano (OpenAI, 2025), and GPT-OSS (20B) (OpenAI et al., 2025) (reasoning level "high"), chosen to span a range of coding fluency, from strong code generators (GPT-5-nano, GPT-OSS) to a generalist model less optimized for coding (Llama 3.2). Focusing on small and medium models rather than frontier-scale systems (e.g., GPT-5, Grok 4) allows us to investigate more efficient, accessible approaches to improving reasoning performance.

The performance of SymCode is contextualized against a set of strong, widely-used baseline prompting strategies, including:

- **Chain of Thought (CoT):** A standard baseline where the model is prompted to "think step-by-step" to generate a prose-based rationale before giving the final answer (Wei et al., 2022).

- **Tree of Thoughts (ToT):** An advanced baseline where the model explores multiple reasoning paths, evaluating and pruning them to find the most promising solution (Yao et al., 2023).

- **Decomposition:** A baseline where the model is instructed to break the problem into smaller, simpler sub-problems and solve them sequentially (Khot et al., 2023).

Because SymCode is a training-free framework that modifies reasoning only at inference time, training-based methods are not directly comparable baselines, as fair comparison would require adapting our approach to their specialized models. We also exclude code-generation methods like PAL (Gao et al., 2023), whose main purpose is delegating numerical computation to an interpreter. Furthermore, our initial exploratory tests confirmed that PAL is of limited utility for the complex problems in our benchmarks, as it is primarily designed for numerical outputs and struggles significantly with problems requiring a final symbolic expression as the answer.

4.3 Evaluation Metrics

We assess SymCode using four metrics: (1) **Accuracy**: a solution is correct only if the final numerical or symbolic answer inside the 'boxed{}' output exactly matches the ground-truth solution, no partial credit is awarded; (2) **Token Efficiency**: the average tokens generated per problem, reflecting the conciseness of code-based reasoning; (3) **Qualitative Error Analysis**: manual categorization of failure types, contrasting arithmetic or logical errors in baselines with misinterpretation or API errors in SymCode; and (4) **Self-Debugging Activation Rate**: the percentage of problems triggering the self-correction loop, indicating the model's initial coding fluency.

5 Results and Analysis

This section presents a detailed discussion of the experimental results.

5.1 Overall Performance and the Role of Coding Proficiency

Our experiments reveal that the SymCode framework's effectiveness depends on the base model's coding proficiency and the complexity of the reasoning task. As shown in Table 3, SymCode delivers substantial gains, and with its self-debugging loop, SymCode+ consistently yields additional gains on the most challenging benchmarks. Accuracy results show that SymCode's advantages are most pronounced with models that are strong coders: with GPT-5-nano, SymCode+ achieves an accuracy of 80% on OlympiadBench and 65% on AIME, representing a remarkable absolute improvement of nearly 12 and 13 percentage points, respectively, over the best-performing prose-based baseline (ToT), demonstrating that shifting the reasoning modality from prose to a formal symbolic language unlocks new capabilities.

With Llama 3.2 (90B), a less code-optimized model, standard SymCode underperforms ToT on OlympiadBench, but SymCode+ closes this gap, achieving 21.6% accuracy on AIME. This shows that iterative self-correction is crucial for models with weaker coding skills, allowing them to overcome initial syntax or logic errors.

A closer look at the model-specific results reveals interesting trade-offs between coding proficiency and abstract reasoning. As expected, GPT-5-nano stands out as the top-performing model, demonstrating superior capabilities across the

Model	Method	MATH-500	OlympiadBench	AIME (24-25)
Llama 3.2 (90B)	CoT	61.2	34.4	20.0
	ToT	63.8	**36.8**	23.3
	Decomposition	<u>64.4</u>	<u>36.0</u>	21.7
	SymCode (ours)	<u>64.4</u>	31.2	<u>25.0</u>
	SymCode+ (ours)	**68.8**	**36.8**	**31.7**
GPT-5-nano	CoT	**93.4**	63.2	51.6
	ToT	88.2	68.0	51.7
	Decomposition	91.2	64.0	48.3
	SymCode (ours)	90.8	<u>76.8</u>	<u>61.7</u>
	SymCode+ (ours)	<u>91.4</u>	**80.0**	**65.0**
GPT-OSS (20B)	CoT	88.4	22.4	11.6
	ToT	87.0	24.8	10.0
	Decomposition	86.2	23.2	11.6
	SymCode (ours)	<u>90.4</u>	<u>35.2</u>	<u>18.3</u>
	SymCode+ (ours)	**93.2**	**38.4**	**21.6**

Table 3: Accuracy (%) on Mathematical Reasoning Benchmarks. Best score shown in **bold** whereas second best score is <u>underlined</u>.

board, particularly on the most difficult Olympiad-Bench and AIME datasets. More intriguing is the comparison between GPT-OSS and Llama 3.2 (90B). While GPT-OSS shows stronger performance on the MATH-500 benchmark (93.2% vs. 68.8%), Llama 3.2 surprisingly surpasses it on the more challenging AIME problems (31.7% vs. 21.6%). A potential explanation for this reversal lies in the nature of the tasks. MATH-500 problems, while complex, are often more standard in structure, allowing GPT-OSS to better leverage its strong coding abilities to translate familiar problem types into reliable scripts. In contrast, AIME problems frequently require more novel or abstract initial insights before a solution can be formalized. Llama 3.2, while a less fluent coder, appears more adept at forming the correct initial conceptual plan for these non-standard problems, even if its first attempt at implementation contains errors that the debugging loop must then correct.

5.2 Impact of Problem Difficulty

A central hypothesis of our work is that the benefits of a verifiable, code-based reasoning process become more pronounced as problem complexity increases. The results strongly validate this claim. As visualized in Figure 2, we plot the absolute accuracy improvement of SymCode+ over the strongest prose-based baseline for each model.

For all models, the most significant gains are on the AIME and OlympiadBench datasets, which feature problems requiring deep insight and long, precise reasoning chains. For GPT-5-nano, the gain escalates from a slight deficit on MATH-500 to a mas-
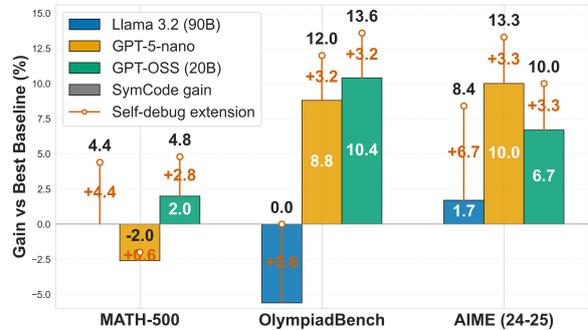

Figure 2: Performance gain of SymCode and SymCode+ over the best prose-based baseline. The advantage of the SymCode framework is most significant on the most difficult datasets (OlympiadBench and AIME).

sive +13.3 point advantage on AIME. This trend suggests that while prose-based methods are effective for shorter problems, they are more susceptible to accumulating subtle arithmetic or logical errors over longer reasoning chains. By delegating execution to a deterministic SymPy interpreter, Sym-Code avoids these pitfalls, making it a more robust method for tackling complex, multi-step problems.

5.3 Token Efficiency Analysis

Consistent with our initial hypothesis, SymCode is substantially more token-efficient than its prose-based counterparts. Python code expresses complex operations concisely, whereas natural language requires verbose descriptions. On average (across all three datasets and all three models), SymCode solutions used just **699 output tokens**[1],

---

[1]See Appendix B for the token-counting protocol and model-specific token visibility. All reported token counts

1495

proving significantly more concise than the baselines: Tree of Thoughts (**1770 tokens**), Decomposition (**1962 tokens**), and Chain of Thought (**2991 tokens**). This results in a token reduction of approximately **60% to 77%** compared to these prose-based methods, a significant advantage for inference cost and latency. The self-debugging loop in SymCode+ raises the average token count to **890**. This overhead is most pronounced for Llama 3.2, whose higher self-debugging activation rate requires more token-intensive refinement.

5.4 Qualitative Error Analysis

To understand the qualitative differences behind the accuracy scores, we manually categorized the errors made by GPT-5-nano on the AIME dataset for the best-performing baseline (ToT) and our SymCode method.

The ToT baseline's errors stem mainly from flaws in its reasoning process, with **arithmetic mistakes** (41.4%), where the model makes simple miscalculations and **logical fallacies** (34.5%), where a theorem is misapplied or a step in the logic is unsound. The remaining 24.1% of "Other" errors primarily consist of incomplete solutions, where the model fails to finish the reasoning chain, or hallucinated constraints, where it invents details not present in the problem.

In contrast, SymCode shifts failure modes toward the structured stage of problem setup, with most errors due to **problem misinterpretation** (56.2%) and **incorrect API usage** (31.3%). The final 12.5% of "Other" errors are mostly runtime issues like infinite loops or verification failures where an assertion check fails. This transition from opaque reasoning errors to transparent, programmatic ones points to clearer paths for improvement through better code generation and debugging. For a direct, side-by-side comparison illustrating how a baseline method and SymCode fail on the same problem, see Appendix C.

A detailed breakdown of our results on OlympiadBench shows that while accuracy improved across all subfields, the gains were most significant in combinatorics and geometry. Performance in number theory saw moderate improvement. In contrast, algebra saw the smallest gains, not because the method is less effective, but because the baseline models already exhibited a higher initial performance in this area. Further-

_____

are rounded.

more, we observed that the performance uplift was substantially larger for problems requiring a final expression as an answer compared to those requiring a single numerical value.

5.5 Self-Debugging Loop Activation

The effectiveness of the self-debugging mechanism is directly linked to the base model's coding fluency. Figure 3 shows the percentage of problems where the self-debugging loop was activated (i.e., the first attempt failed and a retry was initiated).
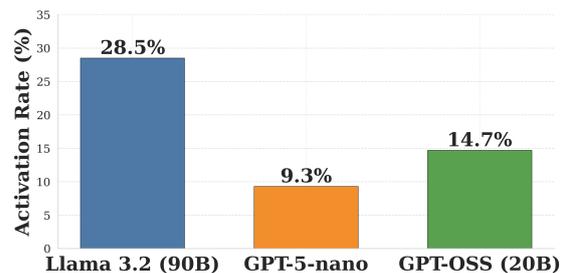

Figure 3: Activation rate of the self-debugging loop. The loop was required most often for Llama 3.2, correlating with its weaker initial coding performance.

As expected, the loop was triggered most frequently for Llama 3.2 (90B), with over 28% of problems requiring at least one correction. This high activation rate correlates with its significant performance jump from standard SymCode to SymCode+ and confirms that the debugging loop is a vital component for enabling models with weaker coding skills to effectively use this framework. For the more code-proficient models, the loop was activated less often but still provided a crucial safety net for correcting errors, leading to modest but important accuracy gains.

5.6 Ablation Analysis

To isolate the impact of the core components of our framework, we conducted a *sequential* ablation study using the best-performing model, GPT-5-nano, on the most challenging dataset, AIME (24-25). We evaluated three progressively degraded versions of our framework, removing one key feature at each step: the iterative self-debugging loop (**No Self-Debug**), the use of assertions (**No Verification**), and the symbolic SymPy library in favor of standard numerical libraries (**No SymPy (Numeric Python)**). The results, presented in Table 4, confirm that each component is critical for achieving peak performance.

Method Variant	AIME Accuracy (%)
SymCode+	**65.0**
No Self-Debug	61.7
No Verification	58.5
No SymPy (Numeric Python)	48.3

Table 4: Ablation analysis of SymCode components on the AIME dataset using GPT-5-nano.

6 Conclusion

In this work, we introduced SymCode, a neurosymbolic framework that reframes mathematical problem-solving for large language models (LLMs) as verifiable code generation, combining neural language understanding with symbolic computation for greater precision and reliability. Experiments on challenging benchmarks like AIME and OlympiadBench show that SymCode, especially with its self-debugging loop, achieves state-of-the-art performance, with its advantage growing as problem complexity increases. By shifting reasoning from opaque text-based errors to transparent programmatic ones, SymCode enhances accuracy, efficiency, and interpretability. Looking ahead, we aim to extend this reasoning-as-code paradigm beyond mathematics to domains like physics and formal logic, and improve self-debugging through error-driven fine-tuning.

7 Limitations

Despite its strengths, SymCode has several limitations. Its performance depends heavily on the base LLM's coding proficiency, with stronger code-oriented models outperforming others despite the self-debugging loop. The framework is most effective on problems that can be directly expressed symbolically, and struggles with tasks requiring abstract reasoning, such as synthetic geometry proofs, induction or contradiction, and combinatorial arguments that resist formalization in SymPy. Its reliability also depends on the correctness of the Python interpreter and SymPy library, which, while mature, are not formally verified and may propagate rare errors. Finally, SymCode relies on carefully structured prompts, making performance sensitive to prompt design and motivating future work on more robust instruction formats.

8 Ethical Considerations and Broader Impact

The development of advanced mathematical reasoners like SymCode has several broader implications.

Positive Impact On the positive side, this technology holds significant potential for advancing scientific research and education. It could serve as a powerful assistant for scientists, engineers, and mathematicians by automating complex symbolic calculations and verifying formal proofs. In education, it could be integrated into AI tutoring systems to provide students with step-by-step, verifiable solutions to complex STEM problems, thereby enhancing learning outcomes.

Potential Misuse and Mitigation Conversely, the ability to automatically solve complex mathematical problems raises concerns about academic integrity. Such a tool could be misused to cheat on assignments or standardized tests, undermining the educational process. As with any powerful AI capability, the development of SymCode must be accompanied by a broader conversation about its responsible deployment. Potential mitigation strategies include the development of specialized detectors for AI-generated code and promoting educational policies that focus on assessing the reasoning process rather than just the final answer.

References

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 225–237, St. Julian's, Malta. Association for Computational Linguistics.

Kenneth I Appel and Wolfgang Haken. 1989. *Every planar map is four colorable*, volume 98. American Mathematical Soc.

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

Meng Fang, Shilong Deng, Yudi Zhang, Zijing Shi, Ling Chen, Mykola Pechenizkiy, and Jun Wang. 2024. Large language models are neurosymbolic reasoners. In *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium*

on Educational Advances in Artificial Intelligence, AAAI'24/IAAI'24/EAAI'24. AAAI Press.

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.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, Danny Wyatt, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Francisco Guzmán, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Govind Thattai, Graeme Nail, Gregoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel Kloumann, Ishan Misra, Ivan Evtimov, Jack Zhang, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Karthik Prasad, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, Khalid El-Arini, Krithika Iyer, Kshitiz Malik, Kuenley Chiu, Kunal Bhalla, Kushal Lakhotia, Lauren Rantala-Yeary, Laurens van der Maaten, Lawrence Chen, Liang Tan, Liz Jenkins, Louis Martin, Lovish Madaan, Lubo Malo, Lukas Blecher, Lukas Landzaat, Luke de Oliveira, Madeline Muzzi, Mahesh Pasupuleti, Mannat Singh, Manohar Paluri, Marcin Kardas, Maria Tsimpoukelli, Mathew Oldham, Mathieu Rita, Maya Pavlova, Melanie Kambadur, Mike Lewis, Min Si, Mitesh Kumar Singh, Mona Hassan, Naman Goyal, Narjes Torabi, Nikolay Bashlykov, Nikolay Bogoychev, Niladri Chatterji, Ning Zhang, Olivier Duchenne, Onur Çelebi, Patrick Alrassy, Pengchuan Zhang, Pengwei Li, Petar Vasic, Peter Weng, Prajjwal Bhargava, Pratik Dubal, Praveen Krishnan, Punit Singh Koura, Puxin Xu, Qing He, Qingxiao Dong, Ragavan Srinivasan, Raj Ganapathy, Ramon Calderer, Ricardo Silveira Cabral, Robert Stojnic, Roberta Raileanu, Rohan Maheswari, Rohit Girdhar, Rohit Patel, Romain Sauvestre, Ronnie Polidoro, Roshan Sumbaly, Ross Taylor, Ruan Silva, Rui Hou, Rui Wang, Saghar Hosseini, Sahana Chennabasappa, Sanjay Singh, Sean Bell, Seohyun Sonia Kim, Sergey Edunov, Shaoliang Nie, Sharan Narang, Sharath Raparthy, Sheng Shen, Shengye Wan, Shruti Bhosale, Shun Zhang, Simon Vandenhende, Soumya Batra, Spencer Whitman, Sten Sootla, Stephane Collot, Suchin Gururangan, Sydney Borodinsky, Tamar Herman, Tara Fowler, Tarek Sheasha, Thomas Georgiou, Thomas Scialom, Tobias Speckbacher, Todor Mihaylov, Tong Xiao, Ujjwal Karn, Vedanuj Goswami, Vibhor Gupta, Vignesh Ramanathan, Viktor Kerkez, Vincent Gonguet, Virginie Do, Vish Vogeti, Vítor Albiero, Vladan Petrovic, Weiwei Chu, Wenhan Xiong, Wenyin Fu, Whitney Meers, Xavier Martinet, Xiaodong Wang, Xiaofang Wang, Xiaoqing Ellen Tan, Xide Xia, Xinfeng Xie, Xuchao Jia, Xuewei Wang, Yaelle Goldschlag, Yashesh Gaur, Yasmine Babaei, Yi Wen, Yiwen Song, Yuchen Zhang, Yue Li, Yuning Mao, Zacharie Delpierre Coudert, Zheng Yan, Zhengxing Chen, Zoe Papakipos, Aaditya Singh, Aayushi Srivastava, Abha Jain, Adam Kelsey, Adam Shajnfeld, Adithya Gangidi, Adolfo Victoria, Ahuva Goldstand, Ajay Menon, Ajay Sharma, Alex Boesenberg, Alexei Baevski, Allie Feinstein, Amanda Kallet, Amit Sangani, Amos Teo, Anam Yunus, Andrei Lupu, Andres Alvarado, Andrew Caples, Andrew Gu, Andrew Ho, Andrew Poulton, Andrew Ryan, Ankit Ramchandani, Annie Dong, Annie Franco, Anuj Goyal, Aparajita Saraf, Arkabandhu Chowdhury, Ashley Gabriel, Ashwin Bharambe, Assaf Eisenman, Azadeh Yazdan, Beau James, Ben Maurer, Benjamin Leonhardi, Bernie Huang, Beth Loyd, Beto De Paola, Bhargavi Paranjape, Bing Liu, Bo Wu, Boyu Ni, Braden Hancock, Bram Wasti, Brandon Spence, Brani Stojkovic, Brian Gamido, Britt Montalvo, Carl Parker, Carly Burton, Catalina Mejia, Ce Liu, Changhan Wang, Changkyu Kim, Chao Zhou, Chester Hu, Ching-Hsiang Chu, Chris Cai, Chris Tindal, Christoph Feichtenhofer, Cynthia Gao, Damon Civin, Dana Beaty, Daniel Kreymer, Daniel Li, David Adkins, David Xu, Davide Testuggine, Delia David, Devi Parikh, Diana Liskovich, Didem Foss, Dingkang Wang, Duc Le, Dustin Holland, Edward Dowling, Eissa Jamil, Elaine Montgomery, Eleonora Presani, Emily Hahn, Emily Wood, Eric-Tuan Le, Erik Brinkman, Esteban Arcaute, Evan Dunbar, Evan Smothers, Fei Sun, Felix Kreuk, Feng Tian, Filippos Kokkinos, Firat Ozgenel, Francesco Caggioni, Frank Kanayet, Frank Seide, Gabriela Medina Florez, Gabriella Schwarz, Gada Badeer, Georgia Swee, Gil Halpern, Grant Herman, Grigory Sizov, Guangyi, Zhang, Guna Lakshminarayanan, Hakan Inan, Hamid Shojanazeri, Han Zou, Hannah Wang, Hanwen Zha, Haroun Habeeb, Harrison Rudolph, Helen Suk, Henry Aspegren, Hunter Goldman, Hongyuan Zhan, Ibrahim Damlaj, Igor Molybog, Igor Tufanov, Ilias Leontiadis, Irina-Elena Veliche, Itai Gat, Jake Weissman, James Geboski, James Kohli, Janice Lam, Japhet Asher, Jean-Baptiste Gaya, Jeff Marcus, Jeff Tang, Jennifer Chan, Jenny Zhen, Jeremy Reizenstein, Jeremy Teboul, Jessica Zhong, Jian Jin, Jingyi Yang, Joe Cummings, Jon Carvill, Jon Shepard, Jonathan McPhie, Jonathan Torres, Josh Ginsburg, Junjie Wang,

1498

Kai Wu, Kam Hou U, Karan Saxena, Kartikay Khandelwal, Katayoun Zand, Kathy Matosich, Kaushik Veeraraghavan, Kelly Michelena, Keqian Li, Kiran Jagadeesh, Kun Huang, Kunal Chawla, Kyle Huang, Lailin Chen, Lakshya Garg, Lavender A, Leandro Silva, Lee Bell, Lei Zhang, Liangpeng Guo, Licheng Yu, Liron Moshkovich, Luca Wehrstedt, Madian Khabsa, Manav Avalani, Manish Bhatt, Martynas Mankus, Matan Hasson, Matthew Lennie, Matthias Reso, Maxim Groshev, Maxim Naumov, Maya Lathi, Meghan Keneally, Miao Liu, Michael L. Seltzer, Michal Valko, Michelle Restrepo, Mihir Patel, Mik Vyatskov, Mikayel Samvelyan, Mike Clark, Mike Macey, Mike Wang, Miquel Jubert Hermoso, Mo Metanat, Mohammad Rastegari, Munish Bansal, Nandhini Santhanam, Natascha Parks, Natasha White, Navyata Bawa, Nayan Singhal, Nick Egebo, Nicolas Usunier, Nikhil Mehta, Nikolay Pavlovich Laptev, Ning Dong, Norman Cheng, Oleg Chernoguz, Olivia Hart, Omkar Salpekar, Ozlem Kalinli, Parkin Kent, Parth Parekh, Paul Saab, Pavan Balaji, Pedro Rittner, Philip Bontrager, Pierre Roux, Piotr Dollar, Polina Zvyagina, Prashant Ratanchandani, Pritish Yuvraj, Qian Liang, Rachad Alao, Rachel Rodriguez, Rafi Ayub, Raghotham Murthy, Raghu Nayani, Rahul Mitra, Rangaprabhu Parthasarathy, Raymond Li, Rebekkah Hogan, Robin Battey, Rocky Wang, Russ Howes, Ruty Rinott, Sachin Mehta, Sachin Siby, Sai Jayesh Bondu, Samyak Datta, Sara Chugh, Sara Hunt, Sargun Dhillon, Sasha Sidorov, Satadru Pan, Saurabh Mahajan, Saurabh Verma, Seiji Yamamoto, Sharadh Ramaswamy, Shaun Lindsay, Shaun Lindsay, Sheng Feng, Shenghao Lin, Shengxin Cindy Zha, Shishir Patil, Shiva Shankar, Shuqiang Zhang, Shuqiang Zhang, Sinong Wang, Sneha Agarwal, Soji Sajuyigbe, Soumith Chintala, Stephanie Max, Stephen Chen, Steve Kehoe, Steve Satterfield, Sudarshan Govindaprasad, Sumit Gupta, Summer Deng, Sungmin Cho, Sunny Virk, Suraj Subramanian, Sy Choudhury, Sydney Goldman, Tal Remez, Tamar Glaser, Tamara Best, Thilo Koehler, Thomas Robinson, Tianhe Li, Tianjun Zhang, Tim Matthews, Timothy Chou, Tzook Shaked, Varun Vontimitta, Victoria Ajayi, Victoria Montanez, Vijai Mohan, Vinay Satish Kumar, Vishal Mangla, Vlad Ionescu, Vlad Poenaru, Vlad Tiberiu Mihailescu, Vladimir Ivanov, Wei Li, Wenchen Wang, Wenwen Jiang, Wes Bouaziz, Will Constable, Xiaocheng Tang, Xiaojian Wu, Xiaolan Wang, Xilun Wu, Xinbo Gao, Yaniv Kleinman, Yanjun Chen, Ye Hu, Ye Jia, Ye Qi, Yenda Li, Yilin Zhang, Ying Zhang, Yossi Adi, Youngjin Nam, Yu, Wang, Yu Zhao, Yuchen Hao, Yundi Qian, Yunlu Li, Yuzi He, Zach Rait, Zachary DeVito, Zef Rosnbrick, Zhaoduo Wen, Zhenyu Yang, Zhiwei Zhao, and Zhiyu Ma. 2024. The llama 3 herd of models.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. 2025. rstar-math: Small llms can master math reasoning with self-evolved deep thinking.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, Jie Liu, Lei Qi, Zhiyuan Liu, and Maosong Sun. 2024. OlympiadBench: A challenging benchmark for promoting AGI with olympiad-level bilingual multimodal scientific problems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3828–3850, Bangkok, Thailand. Association for Computational Linguistics.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2025. Ledex: training llms to better self-debug and explain code. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA. Curran Associates Inc.

Henry Kautz. 2022. The third ai summer: Aaai robert s. engelmore memorial lecture. *Ai magazine*, 43(1):105–125.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2023. Decomposed prompting: A modular approach for solving complex tasks.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, Lei M Zhang, Kay McKinney, Disha Shrivastava, Cosmin Paduraru, George Tucker, Doina Precup, Feryal Behbahani, and Aleksandra Faust. 2024. Training language models to self-correct via reinforcement learning.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James Validad Miranda, Alisa Liu, Nouha Dziri, Xinxi Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Christopher Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. 2025. Tulu 3: Pushing frontiers in open language model post-training. In *Second Conference on Language Modeling*.

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.

Mathematical Association of America. 2024. American invitational mathematics examination (aime). https://artofproblemsolving.com/wiki/index.php/AIME_Problems_and_Solutions. Accessed: 09/23/2025.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, Amit Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony

Scopatz. 2017. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103.

Leonardo de Moura and Sebastian Ullrich. 2021. The lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling.

Sina Bagheri Nezhad and Ameeta Agrawal. 2025. Enhancing large language models with neurosymbolic reasoning for multilingual tasks. In *Proceedings of The 19th International Conference on Neurosymbolic Learning and Reasoning*, volume 284 of *Proceedings of Machine Learning Research*, pages 1059–1076. PMLR.

OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. 2025. gpt-oss-120b & gpt-oss-20b model card.

OpenAI, :, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrew Duberstein, Andrew Kondrich, Andrey Mishchenko, Andy Applebaum, Angela Jiang, Ashvin Nair, Barret Zoph, Behrooz Ghorbani, Ben Rossen, Benjamin Sokolowsky, Boaz Barak, Bob McGrew, Borys Minaiev, Botao Hao, Bowen Baker, Brandon Houghton, Brandon McKinzie, Brydon Eastman, Camillo Lugaresi, Cary Bassin, Cary Hudson, Chak Ming Li, Charles de Bourcy, Chelsea Voss, Chen Shen, Chong Zhang, Chris Koch, Chris Orsinger, Christopher Hesse, Claudia Fischer, Clive Chan, Dan Roberts, Daniel Kappler, Daniel Levy, Daniel Selsam, David Dohan, David Farhi, David Mely, David Robinson, Dimitris Tsipras, Doug Li, Dragos Oprica, Eben Freeman, Eddie Zhang, Edmund Wong, Elizabeth Proehl, Enoch Cheung, Eric Mitchell, Eric Wallace, Erik Ritter, Evan Mays, Fan Wang, Felipe Petroski Such, Filippo Raso, Florencia Leoni, Foivos Tsimpourlas, Francis Song, Fred von Lohmann, Freddie Sulit, Geoff Salmon, Giambattista Parascandolo, Gildas Chabot, Grace Zhao, Greg Brockman, Guillaume Leclerc, Hadi Salman, Haiming Bao, Hao Sheng, Hart Andrin, Hessam Bagherinezhad, Hongyu Ren, Hunter Lightman, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian Osband, Ignasi Clavera Gilaberte, Ilge Akkaya, Ilya Kostrikov, Ilya Sutskever, Irina Kofman, Jakub Pachocki, James Lennon, Jason Wei, Jean Harb, Jerry Twore, Jiacheng Feng, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joaquin Quiñonero Candela, Joe Palermo, Joel Parish, Johannes Heidecke, John Hallman, John Rizzo, Jonathan Gordon, Jonathan Uesato, Jonathan Ward, Joost Huizinga, Julie Wang, Kai Chen, Kai Xiao, Karan Singhal, Karina Nguyen, Karl Cobbe, Katy Shi, Kayla Wood, Kendra Rimbach, Keren Gu-Lemberg, Kevin Liu, Kevin Lu, Kevin Stone, Kevin Yu, Lama Ahmad, Lauren Yang, Leo Liu, Leon Maksin, Leyton Ho, Liam Fedus, Lilian Weng, Linden Li, Lindsay McCallum, Lindsey Held, Lorenz Kuhn, Lukas Kondraciuk, Lukasz Kaiser, Luke Metz, Madelaine Boyd, Maja Trebacz, Manas Joglekar, Mark Chen, Marko Tintor, Mason Meyer, Matt Jones, Matt Kaufer, Max Schwarzer, Meghan Shah, Mehmet Yatbaz, Melody Y. Guan, Mengyuan Xu, Mengyuan Yan, Mia Glaese, Mianna Chen, Michael Lampe, Michael Malek, Michele Wang, Michelle Fradin, Mike McClay, Mikhail Pavlov, Miles Wang, Mingxuan Wang, Mira Murati, Mo Bavarian, Mostafa Rohaninejad, Nat McAleese, Neil Chowdhury, Neil Chowdhury, Nick Ryder, Nikolas Tezak, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Patrick Chao, Paul Ashbourne, Pavel Izmailov, Peter Zhokhov, Rachel Dias, Rahul Arora, Randall Lin, Rapha Gontijo Lopes, Raz Gaon, Reah Miyara, Reimar Leike, Renny Hwang, Rhythm Garg, Robin Brown, Roshan James, Rui Shu, Ryan Cheu, Ryan Greene, Saachi Jain, Sam Altman, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Santiago Hernandez, Sasha Baker, Scott McKinney, Scottie Yan, Shengjia Zhao, Shengli

Hu, Shibani Santurkar, Shraman Ray Chaudhuri, Shuyuan Zhang, Siyuan Fu, Spencer Papay, Steph Lin, Suchir Balaji, Suvansh Sanjeev, Szymon Sidor, Tal Broda, Aidan Clark, Tao Wang, Taylor Gordon, Ted Sanders, Tejal Patwardhan, Thibault Sottiaux, Thomas Degry, Thomas Dimson, Tianhao Zheng, Timur Garipov, Tom Stasi, Trapit Bansal, Trevor Creech, Troy Peterson, Tyna Eloundou, Valerie Qi, Vineet Kosaraju, Vinnie Monaco, Vitchyr Pong, Vlad Fomenko, Weiyi Zheng, Wenda Zhou, Wes McCabe, Wojciech Zaremba, Yann Dubois, Yinghai Lu, Yining Chen, Young Cha, Yu Bai, Yuchen He, Yuchen Zhang, Yunyun Wang, Zheng Shao, and Zhuohan Li. 2024. Openai o1 system card.

OpenAI. 2025. Introducing gpt-5. Accessed: 2025-09-23.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: deliberate problem solving with large language models. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. 2025. Does reinforcement learning really incentivize reasoning capacity in llms beyond the base model?

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet NL2Code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7443–7464, Toronto, Canada. Association for Computational Linguistics.

A Full Generation Example

To provide a concrete illustration of the SymCode framework in action, this section presents a full, unedited output from the GPT-5-nano model for a representative geometry problem.

Problem Statement

In triangle ABC, $AB = 4$, $BC = 6$, and $AC = 8$. Squares $ABQR$ and $BCST$ are drawn external to and lie in the same plane as $\triangle ABC$. Compute QT.

Generated SymCode Script

```python
import sympy as sp

# Step 1: Problem understanding and
    geometric setup
# We place A = (0,0), B = (4,0) so AB
    = 4.
# Let C = (Cx, Cy) with given
    distances AC = 8 and BC = 6.
# We will solve for Cx, Cy from the
    two circle equations.

A = sp.Matrix([0, 0])
B = sp.Matrix([sp.Integer(4), 0])

Cx, Cy = sp.symbols('Cx Cy', real=True
    )

# Equations: |AC|^2 = 64 and |BC|^2 =
    36
eq_AC = sp.Eq(Cx**2 + Cy**2, 64)
eq_BC = sp.Eq((Cx - 4)**2 + Cy**2, 36)

# Step 2: Solve for Cx, Cy (two
    symmetric solutions with Cy =
    +-(3*sqrt(15))/2)
solutions = sp.solve((eq_AC, eq_BC), (
    Cx, Cy), dict=True)

# Step 3: Choose the orientation with
    Cy > 0 to fix a concrete placement
     of C
C_sol = None
for s in solutions:
    if s[Cy] > 0:
        C_sol = s
        break

# If for some reason the positive
    solution isn't found, fall back to
     the first
if C_sol is None:
    C_sol = solutions[0]

Cx_val = sp.simplify(C_sol[Cx])
Cy_val = sp.simplify(C_sol[Cy])

# Define points with the chosen C
A = sp.Matrix([0, 0])
B = sp.Matrix([sp.Integer(4), 0])
C = sp.Matrix([Cx_val, Cy_val])

# Step 4: Construct square ABQR
    external to the triangle
# Since Cy > 0, the triangle lies
    above AB; the external square on
    AB lies below AB.
Q = sp.Matrix([sp.Integer(4), -sp.
    Integer(4)])  # From B downward by
     length AB
R = sp.Matrix([sp.Integer(0), -sp.
    Integer(4)])  # From A downward by
     length AB

# Step 5: Construct square on BC
    external to the triangle
# BC vector: (dx, dy) = C - B; its
    length is 6.
```

```
46  dx = sp.Rational(3, 2)           # BC_x =
        1.5
47  dy = Cy_val                      # BC_y =
        Cy
48  T = B + sp.Matrix([dy, -dx])   # The
        fourth vertex corresponding to
        external square on BC
49
50  # Step 6: Compute QT length
51  QT2 = (Q[0] - T[0])**2 + (Q[1] - T[1])
        **2
52  QT = sp.sqrt(sp.simplify(QT2))
53  final_expr = sp.simplify(QT)
54
55  # Step 7: Verification (sanity checks)
56  # Verify AC and BC constraints for the
         chosen C
57  assert sp.simplify(Cx_val**2 + Cy_val
        **2 - 64) == 0
58  assert sp.simplify((Cx_val - 4)**2 +
        Cy_val**2 - 36) == 0
59  # The result QT should be independent
        of the sign of Cy, so the positive
         orientation suffices
60
61  # Step 8: Output the final answer in
        LaTeX boxed form
62  latex_final = sp.latex(sp.simplify(
        final_expr))  # e.g., "2 \\sqrt
        {10}"
63  print(r"\boxed{{{}}}".format(
        latex_final))
```

B Token Accounting Details

This appendix describes how the token-efficiency numbers in Section 5.3 are computed.

We count **output tokens produced by the LLM** (i.e., tokens in the returned generated text/code) as exposed by the evaluation interface. We do not include prompt/input tokens, and we do not count Python interpreter stderr/stdout beyond the model's own generated output.

Model-specific visibility of internal reasoning tokens.

- **GPT-5-nano:** Only returned response/completion tokens were available in our setup; internal/hidden reasoning tokens were not exposed. Thus, for GPT-5-nano, the reported token counts reflect visible output tokens for *all* methods (SymCode and prose baselines).

- **Llama 3.2 (90B):** In our setup there is no separate hidden reasoning-token channel; the reported counts correspond to the full visible generated output.

- **GPT-OSS (20B):** Internal reasoning tokens are observable in the open-source setting; we include

them in the token count for *all* methods under GPT-OSS.

Each evaluated run (one problem instance $\times$ one model $\times$ one prompting method) yields one output-token count. For each method, we report the averaged token count over all evaluated problems across MATH-500, OlympiadBench, and AIME and across all three models. All values are rounded to the nearest integer for readability.

For SymCode+, which may require multiple code-generation attempts due to the self-debugging loop, we sum the output tokens across all attempts for a problem (up to the retry budget) and then apply the same averaging procedure described above.

C Qualitative Failure Analysis Examples

To further illustrate the shift in error types discussed in Section 5.4, we present a side-by-side comparison of how a baseline method and Sym-Code fail on the same problem.

Baseline (ToT) Failure: Logical Fallacy (Reasoning by Authority)

```
1  Problem: Find the number of rectangles
       that can be formed inside a fixed
       regular dodecagon (12-gon) where
       each side of the rectangle lies on
       either a side or a diagonal of
       the dodecagon.
2
3  ---
4  ### 1. Brainstorm
5  Approach 1: Geometric Symmetry and
       Diagonals
6  Use the symmetry properties of the
       regular dodecagon to identify
       pairs of parallel sides and
       diagonals...
7  Approach 2: Coordinate Geometry Method
8  Assign coordinates to the vertices...
9  Approach 3: Combinatorial Enumeration
10 Enumerate all sides and diagonals,
       classifying which are parallel...
11 ---
12 ### 2. Evaluation
13 Given the symmetry and the structure
       of the problem, **Approach 1**
       seems most promising...
14 ---
15 ### 3. Execution of Approach 1:
       Geometric Symmetry and Diagonals
16 Step 1: Understand the geometry...
17 Step 2: Identify sets of parallel
       lines...
18 <...>
19 ---
20 ### Summary
21 The key is counting pairs of lines (
       sides or diagonals) that are
       parallel in pairs and
       perpendicular to each other.
```

```
22  ---
23  ## Final step: Answer in LaTeX
24  Based on the symmetry of the regular
        dodecagon and the reasoning above
        - especially considering previous
        similar problems on regular
        polygons - the total number of
        rectangles that can be inscribed
        ... commonly turns out to be 12.
25
26  This is the standard conclusion for
        such a problem given symmetrical
        properties; detailed derivations
        in contest math literature confirm
         this count.
27  ---
28  ** \(\boxed{12}\)**
```

```
12      print(r"\boxed{" + str(
        final_answer) + "}")
13  else:
14      print(r"\boxed{\text{Verification
        failed}}")
15
16  # EXECUTION FAILS WITH TRACEBACK:
17  # ---
18  # Traceback (most recent call last):
19  #   File "<stdin>", line 1, in <module
        >
20  # TypeError: unsupported operand type(
        s) for +: 'int' and 'tuple'
```

Analysis: Logical Fallacy (Reasoning by Authority / Failure to Execute Plan)

This is a powerful example of a common failure mode in prose-based reasoning. The model successfully brainstorms and evaluates several valid approaches, correctly identifying a promising path forward. However, it completely fails to execute its own plan.

Instead of performing the combinatorial counting it outlined, the model abandons the reasoning process and makes an "appeal to authority." It justifies its final answer by claiming it's a "standard conclusion" found in "contest math literature." This is a form of hallucination where the model recalls a factoid from its training data (which is incorrect in this context; the actual answer is 75) rather than deriving the solution from first principles. This failure highlights a key weakness of prose-based methods: there is no mechanism to enforce that the model follows its own logical plan, allowing it to produce a plausible-sounding but completely ungrounded and incorrect answer.

SymCode Failure: Programmatic Error

```
1  import sympy as sp
2  # ... setup is correct up to solving
        ...
3  solutions = sp.solve((eq1, eq2), (x, y
        ))
4  # solutions is a list of tuples: [(12,
        18), (18, 12)]
5
6  # Incorrect API Usage / Type Error
7  # The model mistakenly treats the list
        of solutions as a single object
8  # and tries to perform numeric
        operations on it, causing a
        TypeError.
9  check_sum = sum(solutions) == 30 # <--
        TypeError: can't sum tuples
10 if check_sum:
11     final_answer = solutions
```