A TOOLBOX, NOT A HAMMER — MULTI-TAG: SCALING MATH REASONING WITH MULTI-TOOL AGGREGATION

Bohan Yao[1,2] **Vikas Yadav**[1]
[1]ServiceNow AI [2]University of Washington
s1104@cs.washington.edu

ABSTRACT

Augmenting large language models (LLMs) with external tools is a promising avenue for developing high-performance mathematical reasoning systems. Prior tool-augmented approaches typically finetune an LLM to select and invoke a *single* tool at each reasoning step and show promising results on simpler math reasoning benchmarks such as GSM8K. However, these approaches struggle with more complex math problems that require precise reasoning over multiple steps. To address this limitation, in this work, we propose **Multi-TAG**, a **Multi-T**ool **AG**gregation-based framework. Instead of relying on a single tool, **Multi-TAG** guides an LLM to concurrently invoke multiple tools at each reasoning step. It then aggregates their diverse outputs to verify and refine the reasoning process, enhancing solution robustness and accuracy. Notably, **Multi-TAG** is a finetuning-free, inference-only framework, making it readily applicable to any LLM backbone, including large open-weight models which are computationally expensive to finetune and proprietary frontier models which cannot be finetuned with custom recipes. We evaluate **Multi-TAG** on four challenging benchmarks: MATH500, AIME, AMC, and OlympiadBench. Across both open-weight and closed-source LLM backbones, **Multi-TAG** consistently and substantially outperforms state-of-the-art baselines, achieving average improvements of 6.0% to 7.5% over state-of-the-art baselines. [1]

1 INTRODUCTION

Large Language Models (LLMs) have demonstrated remarkable capabilities across a wide variety of tasks, with reasoning emerging as a core area of research (Jiang et al., 2023; OpenAI, 2023; 2022; Yang et al., 2024). In particular, imbuing LLMs with the ability to perform complex mathematical reasoning remains an active challenge (Ahn et al., 2024). To address this challenge, tool-augmented LLM (TALM) frameworks such as Program-Aided Language Models (PAL) (Gao et al., 2023), Program-of-Thought (PoT) (Chen et al., 2022), Tool-Integrated Reasoning Agent (ToRA) (Gou et al., 2024), and MATHSENSEI (Das et al., 2024) equip LLMs with external tools such as Python code execution or WolframAlpha querying. While previous TALM frameworks have shown notable progress on simpler math benchmarks such as GSM8K Cobbe et al. (2021a), their performance plateaus and remains low on more complex benchmarks such as MATH500 (Lightman et al., 2023), AIME, AMC, and OlympiadBench (He et al., 2024).

Inference-time compute scaling approaches such as OpenAI o1 (OpenAI, 2024), DeepSeek-R1 (DeepSeek-AI et al., 2025) and rStar-Math (Guan et al., 2025) allocate more computational resources to LLMs at inference time to allow them to reason more methodically about problems, and also demonstrate promise towards addressing the challenge of LLM math reasoning. Although these approaches achieve impressive performance on complex math benchmarks, they generally require extensive finetuning, which can be especially brittle and requires training tricks and carefully tuned hyperparameters to be effective (Zeng et al., 2025). Moreover, finetuning can be prohibitively expensive, especially for larger LLMs, and may not be possible for proprietary LLMs whose APIs do

[1]**Multi-TAG** GitHub will be open-sourced soon.

not support the finetuning recipes used in these approaches. Furthermore, recent work has shown that these approaches exhibit non-robust reasoning behaviors in various scenarios, such as number transformations (Yu et al., 2025), adversarial triggers (Rajeev et al., 2025), unanswerable tasks (Hashemi et al., 2025), and executing algorithmic solutions (Shojaee et al., 2025).

In this work, we propose **Multi-TAG**, a **Multi-T**ool **AG**gregation framework that aims to address the limitations of existing TALM frameworks on complex math reasoning tasks by adopting the inference-time scaling paradigm. As opposed to previous TALM frameworks which select a single tool to invoke at each reasoning step, **Multi-TAG** scales up inference-time compute usage by invoking multiple tools at each reasoning step and aggregating their outputs, utilizing the consensus between different tool invocations to ensure that accurate reasoning steps are made.

The core benefit of multi-tool aggregation is cross-validation of different tools' outputs. Since different tools have different strengths and failure modes (see Section 5.1.2), their agreement on a result provides strong evidence of its correctness. For example, a natural language reasoning tool and a Python code execution tool both proposing a consistent result lends significant credence to the validity of the result, as it is improbable that both tools made unique mistakes thematic to their own weaknesses (e.g. calculation mistakes for the natural language tool and logical mistakes for the Python tool) yet coincidentally arrived at the same incorrect result. By invoking and aggregating a diverse set of tools at each reasoning step, **Multi-TAG** harnesses this principle to self-validate candidates for each step, significantly improving the overall reasoning performance.

Beyond superior reasoning performance, another key strength of **Multi-TAG** is that it is a purely inference-time approach, making it readily applicable to any general instruction-tuned LLM. In contrast, finetuning-based TALM approaches such as ToRA (Gou et al., 2024) or MathCoder (Wang et al., 2024a) incur a significant compute overhead for finetuning when applied to new backbone LLMs and cannot be applied to proprietary LLMs lacking finetuning APIs. To demonstrate **Multi-TAG**'s transferability to different LLM backbones, we replicate our main results on three LLMs, including both open-weight and proprietary models, and observe consistent and significant performance improvements over baselines for all three LLMs. Furthermore, **Multi-TAG**'s inference compute costs are adjustable, allowing users to tune **Multi-TAG** to fit their cost-performance tradeoff requirements, as opposed to prior TALM approaches where compute allocation is not easily tunable. We study how performance and compute usage vary for different **Multi-TAG** hyperparameter settings in Section 5.3 and propose a simple hyperparameter heuristic to maximize performance under a fixed compute budget.

Our key contributions are:

1. We introduce **Multi-TAG**, a tool-augmented LLM framework that effectively solves complex math reasoning tasks by aggregating multiple tool invocations at each reasoning step. The code repository for **Multi-TAG** code will be open-sourced in the near future.
2. We present extensive evaluations of **Multi-TAG**, seven simple baselines, and five state-of-the-art TALM baselines on three LLM backbones and four challenging math reasoning benchmarks: MATH500, AIME, AMC, and OlympiadBench. Across the three LLMs, state-of-the-art TALM baselines perform poorly, with the strongest TALM baseline for each model achieving 1.3% to 6.2% *lower* accuracy compared to the strongest simple baseline for each model. Meanwhile, **Multi-TAG** performs much better, achieving 6.0% to 7.5% higher accuracy compared to the strongest simple baseline for each model and 7.9% to 13.7% higher accuracy compared to the strongest TALM baseline for each model.
3. We present comprehensive analyses highlighting the strengths of **Multi-TAG** and benchmarking the cost-performance trade-off. In particular, we analyze **Multi-TAG**'s performance by problem difficulty and problem subject area in Section 5.1. We also study how **Multi-TAG**'s performance and compute cost vary with different hyperparameter settings and present insights for how users can tune **Multi-TAG** to extract maximal performance under various computational budgets in Section 5.3.

2 Related Work

Tool-Augmented Language Models Recent advancements in developing tool-augmented language models (TALMs) have shown promise in improving performance on reasoning tasks. Frame-

works such as ToolFormer (Schick et al., 2023), OlaGPT (Xie et al., 2023), and ART (Paranjape et al., 2023) demonstrate that allowing LLMs to access external tools can significantly boost performance in various domains. Many previous works have also focused on developing TALMs for math reasoning tasks. Program-aided Language Models (PAL) (Gao et al., 2023) and Program of Thoughts (PoT) (Chen et al., 2022) propose to prompt LLMs to generate Python code solutions to math problems instead of natural language Chain-of-Thought (CoT) solutions and achieve strong performance on simple math word problem datasets such as GSM8K (Cobbe et al., 2021a) and SVAMP (Patel et al., 2021). Mammoth (Yue et al., 2023) finetunes models to produce both CoT solutions and Python code solutions to math problems. More recent works have recognized the importance of multi-step tool use for solving more challenging math problems which are too complex to solve with a single tool call. MuMath-Code (Yin et al., 2024) and ToRA (Gou et al., 2024) finetune models to generate solutions with multiple interleaved CoT and Python code reasoning traces. MathSensei (Das et al., 2024) prompts LLMs to solve math problems by following handcrafted workflows of sequences of tool invocations. Code-based Self-Verification prompting (Zhou et al., 2023) leverages GPT4 Code Interpreter's built-in code execution capabilities to verify each CoT reasoning step with a Python script.

While previous TALM approaches have been effective on simpler math reasoning tasks, their use of only one tool invocation per reasoning step may limit their scalability to more complex math problems. To address this limitation, we hypothesize that allowing TALMs to invoke multiple tools at each reasoning step improves their capacity for solving complex math problems by enabling cross-verification and leveraging the complementary strengths of different tools. To this end, the proposed **Multi-TAG** framework aggregates outputs from multiple tools at each step, allowing different tools to validate and reinforce each part of the reasoning process, thus enhancing the accuracy of intermediate reasoning and the final solution. Furthermore, TALM approaches that rely on finetuning are not easily generalizable to all models, requiring significant computational resources to finetune larger models (DeepSeek-AI et al., 2025), and impossible to apply to proprietary models lacking finetuning APIs. In contrast, our framework relies solely on prompting and the inherent instruction following capabilities of LLMs, allowing it to be seamlessly applied to any LLM.

Inference-time Compute Scaling Inference-time compute scaling approaches aim to improve LLM reasoning performance by utilizing more compute at inference time. Self-consistency (Wang et al., 2022) proposes a simple way to implement inference-time scaling, where multiple solutions are sampled from an LLM and the most common answer is chosen. Building on top of these repeated sampling approaches, other works have further proposed to train verifiers to better select the best solution from the sampled solutions. Some works propose to train verifiers to judge entire solutions (Cobbe et al., 2021b), whereas others propose to train verifiers to judge individual steps of solutions (Lightman et al., 2023; Wang et al., 2024b). Inline with step-wise verification approaches, our **Multi-TAG** approach also performs verification at each reasoning step, but utilizes cross-verification between different tools' outputs instead of trained verifiers.

Inference scaling via long CoT reasoning has also been shown to be an effective technique (Muennighoff et al.; DeepSeek-AI et al., 2025), although solving reasoning tasks with only CoT reasoning has limitations in various settings (Sprague et al.; Yu et al., 2025; Rajeev et al., 2025; Hashemi et al., 2025; Shojaee et al., 2025). In a complementary direction, our work explores inference-time scaling through *multi-tool* aggregation to enhance the performance of TALMs.

3 MULTI-TAG

Multi-TAG is an LLM reasoning framework that scales up inference-time compute via multi-tool aggregation to improve reasoning performance. It equips an LLM-based problem solver with a diverse set of tools to solve complex problems in a step-by-step manner. At each step, each tool is invoked multiple times, generating a set of tool-augmented reasoning steps that serve as candidates for the next reasoning step. Note that each candidate may propose a different objective for the next step, creating a more diverse pool of candidates and allowing each candidate to tailor its proposed objective to the strengths of the tool used. These candidates are then aggregated, and the most accurate and productive candidate is selected to continue the solution. By aggregating candidates using different tools, **Multi-TAG** exploits the unique strengths of each tool to cross-validate each candidate's reasoning. Specifically, **Multi-TAG** utilizes final answer estimates derived from

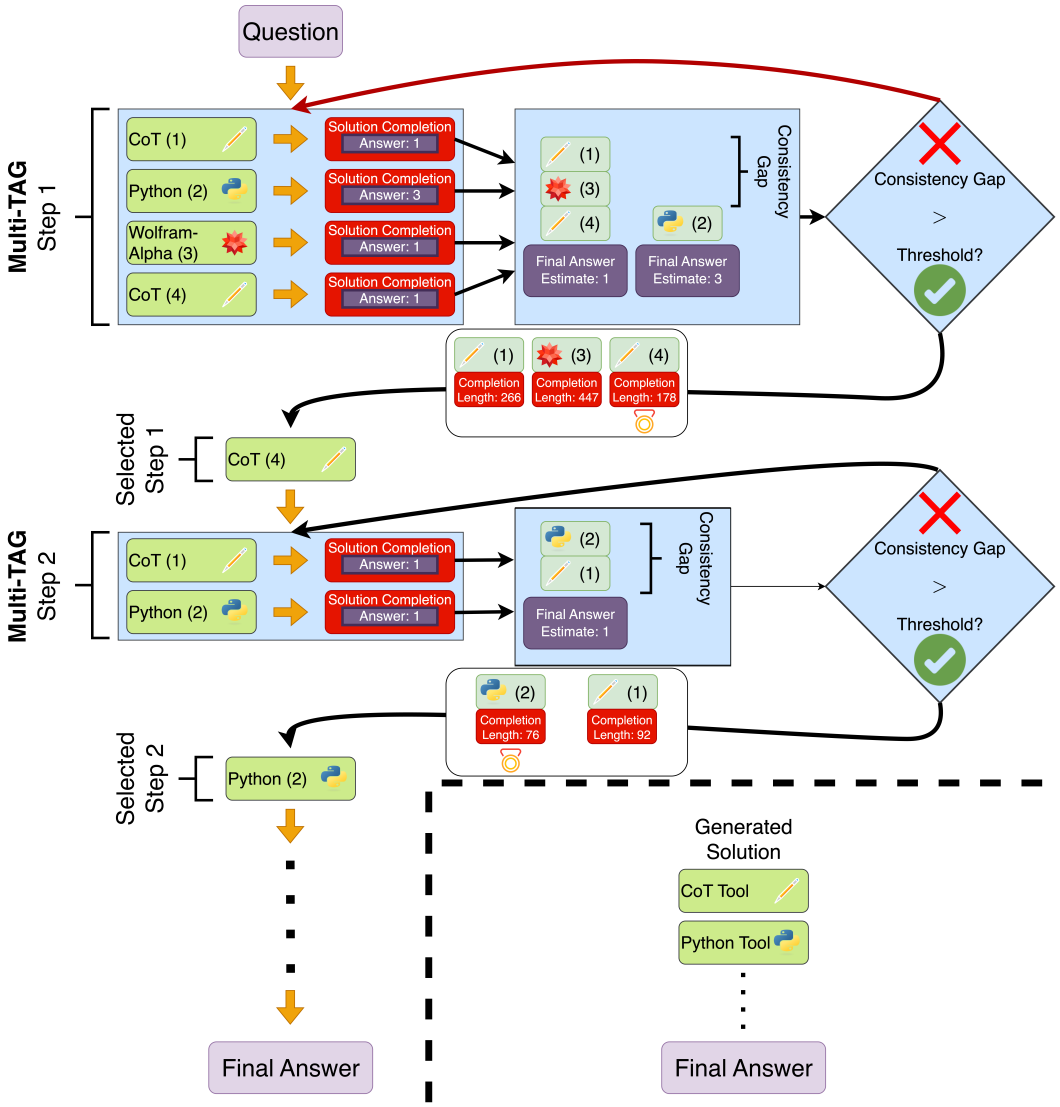

Figure 1: Visualization of the **Multi-TAG** framework with a consistency threshold value of 1. In the first step, after the first four executors are invoked, candidates CoT (1), Python (2), WolframAlpha (3), and CoT (4) are produced. Candidates (1), (3), (4) have final answer estimate 1, while executor (2) has final answer estimate 3. The frequency of the most frequent final answer estimate, 1, is 3, while the frequency of the second most frequent final answer estimate, 3, is 1, so the consistency gap is $3 - 1 = 2$, which is greater than the consistency threshold value. Hence, executor invocation terminates. To select a candidate, first the candidates (1), (3), (4) are shortlisted as they reach the most frequent final answer estimate of 1. Then, (4) is selected as it has the shortest solution completion. In the second step, only two executors were invoked for the consistency gap to exceed the consistency threshold value. Candidate (2) was selected due to having the shorter solution completion. This process repeats until the selected step reaches a final answer for the problem. The full generated solution to the problem is the concatenation of all the selected steps.

each candidate to aggregate candidates that utilize different tools and potentially achieve different objectives.

Figure 1 provides an overview of the **Multi-TAG** system, and Algorithm 1 provides an explicit pseudocode implementation. Given a problem $\mathcal{P}$ and a set of tools $\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$, **Multi-TAG** constructs a step-by-step solution $s_1, s_2, \ldots, s_n$ with each step invoking one of the tools in $\mathcal{T}$. At the p'th step, **Multi-TAG** starts by sequentially invoking a set of $m \times t$ LLM *executors*. An early stopping criteria is checked after each *executor's* invocation to determine if executor invocation should

be terminated early; see Section 3.1 for details. The i'th *executor* is assigned tool $T_{((i-1) \bmod t)+1}$ and given $\mathcal{P}$ and the current partial solution $s_1, s_2, \ldots, s_{p-1}$. It is prompted to propose a candidate s_p^i for the next reasoning step. The value of $m \times t$ is a tunable hyperparameter which we call the max executors value, which can be tuned to adjust the amount of inference compute utilized.

After executor invocation is completed, each candidate s_p^i is appended to the current partial solution, forming a candidate partial solution $\mathrm{cand}_i = s_1, s_2, \ldots, s_{p-1}, s_p^i$. An LLM *completer* is then invoked for each candidate partial solution. The i'th *completer* is given $\mathcal{P}$ and cand_i and is prompted to generate a natural language solution completion comp_i, which when concatenated after cand_i forms a complete solution to $\mathcal{P}$. The final answer reached by this concatenated solution serves as a quick "approximation" of the final answer assuming cand_i is accurate, and we call it the i'th final answer estimate est_i.

Finally, to select the best cand_i to serve as the next step in the current partial solution, a two-step selection procedure is employed. In the first step, the most frequent final answer estimate $\mathrm{maxest} = \mathrm{mode}(\{\mathrm{est}_1, \mathrm{est}_2, \ldots, \mathrm{est}_{m \times t}\})$ is identified, and all candidates cand_i such that $\mathrm{est}_i = \mathrm{maxest}$ are shortlisted. Similar to self consistency's motivation, the more candidates that reach consistent final answers, the more confident we can be about the candidates' accuracy. In the second step, the shortlisted candidate with the shortest solution completion (measured in number of LLM tokens) is selected to be the next step in the current partial solution. Intuitively, selecting this step would lead to the most concise solution, improving **Multi-TAG**'s compute efficiency. Furthermore, we find in our ablation analyses in Section 5.2.3 that using this second selection step also improves performance. This finding can be intuitively explained by the Occam's Razor principle, that concise explanations should be favored over complex ones, and is consistent with recent work demonstrating that concise reasoning can sometimes lead to stronger performance (Sui et al., 2025).

We perform ablation tests on our two-step selection procedure in Section 5.2.3 and find empirically that both steps are necessary to achieve maximum performance. Furthermore, we demonstrate that the second step additionally improves inference efficiency, and removing it results in substantially higher inference costs.

3.1 Consistency Threshold And Early Termination

At each reasoning step, **Multi-TAG** invokes executors sequentially. After each executor invocation, **Multi-TAG** uses the consistency threshold to determine whether executor invocation should be terminated early.

We define the consistency gap as the difference between the frequencies of the most frequent and second most frequent final answer estimates. If the consistency gap exceeds the consistency threshold value, executor invocation is terminated. Intuitively, when the consistency gap is high, the executors are largely consistent with each other, and hence we can be confident that the largest group of consistent executors are accurate. The consistency threshold is a hyperparameter that can be tuned to trade inference cost for performance and vice versa.

We perform ablation tests to validate the effectiveness of the consistency threshold in Section 5.2.2. We find that the consistency threshold lowers inference costs substantially, while incurring a negligible accuracy penalty.

4 Results

4.1 Datasets & Models

Following many recent works on LLM reasoning, we evaluate **Multi-TAG** on challenging short answer math problems. One motivation for this choice is that there are many math datasets with thoroughly vetted ground truth answers publicly available. Furthermore, answers to short answer math problems can be easily and accurately verified by comparing the model answers to ground truth answers with a symbolic equality checker such as SymPy. Hence, evaluating on short answer math problems enhances the reliability and reproducibility of our evaluations.

Algorithm 1 Pseudocode for **Multi-TAG** algorithm

Require: Problem $\mathcal{P}$, Toolset $\mathcal{T} = \{T_1, T_2, \ldots, T_t\}$, Max executors value $m \times t$, Consistency threshold value `thresh`
Ensure: Step-by-step solution $\mathcal{S}_n = [s_1, s_2, \ldots, s_n]$ to $\mathcal{P}$
Initialize current partial solution $\mathcal{S}_0 \leftarrow []$
for $p = 0$ to $n - 1$ **do**
  Initialize candidate pool $\mathcal{C} \leftarrow []$
  for $k = 1$ to $m \times t$ **do**
    Invoke k'th *executor* to generate candidate c_{p+1}^k using tool $T_{((k-1) \bmod t)+1}$, given $\mathcal{P}$ and $\mathcal{S}_p$
    Append c_{p+1}^k to $\mathcal{C}$
    Form candidate partial solution $\mathrm{cand}_k \leftarrow [\mathcal{S}_p, c_{p+1}^k]$
    Use *completer* to generate natural language solution completion comp_k given $\mathcal{P}$ and cand_k
    Extract final answer estimate est_k from comp_k
    if Consistency gap $>$ `thresh` **then**
      break
    end if
  end for
  Identify most frequent final answer estimate maxest among all est_i
  Shortlist candidates $\mathcal{C}_{\mathrm{shortlist}} = \{c_{p+1}^k \mid \mathrm{est}_k = \mathrm{maxest}\}$
  Select c_{p+1}^* from $\mathcal{C}_{\mathrm{shortlist}}$ whose comp_* is shortest
  Append c_{p+1}^* to current partial solution: $\mathcal{S}_{p+1} \leftarrow [\mathcal{S}_p, c_{p+1}^*]$
end for
return $\mathcal{S}_n = [s_1, s_2, \ldots, s_n]$

We select four challenging math reasoning datasets for evaluation: MATH Hendrycks et al. (2021), AMC [2], AIME [3], and OlympiadBench He et al. (2024). Following recent work on math reasoning, we evaluate on the MATH500 subset Lightman et al. (2023) of the full MATH dataset to accelerate evaluations. For OlympiadBench, we only use the two English, text only, open ended (short answer) splits, OE_TO_maths_en_COMP and OE_TO_physics_en_COMP.

Multi-TAG is applicable to any LLM. We run evaluations using LLaMA-3-70B Team (2024), LLaMA-3.3-70B, and GPT-4o (05-13) to illustrate the efficacy of our method on less performant open models, near-frontier level open models, and frontier level proprietary models.

4.2 MULTI-TAG IMPLEMENTATION DETAILS

For all **Multi-TAG** experiments, unless specified otherwise, we use a maximum of 12 executors per step and a consistency threshold of 2. We selected these hyperparameters to achieve near-optimal performance while maintaining a relatively low compute cost; see Section 5.3 for a detailed analysis of **Multi-TAG** hyperparameters. We provide **Multi-TAG** with three tools: CoT reasoning, Python script execution, and WolframAlpha queries. For sampling tool invocations, we use temperature 0.7 and top_p 0.9, and for partial solution completions, we use temperature 0.0. For MATH500, AIME, and AMC, we use Math-Verify Kydlíček to grade model predictions. For OlympiadBench, we use the autograder provided in the OlympiadBench GitHub. **Multi-TAG** prompts are available in Appendix A.

4.3 BASELINES

Simple We evaluate simple baselines that perform a single tool invocation per problem. Specifically, for each of the three tools available to **Multi-TAG**, we create a baseline where an LLM is prompted to solve the problem with a single invocation of the tool, with the tool's output serving as the final answer. For the WolframAlpha baseline, the WolframAlpha API often returns improperly

[2] `https://huggingface.co/datasets/AI-MO/aimo-validation-amc`
[3] `https://huggingface.co/datasets/AI-MO/aimo-validation-aime`

formatted results, such as Unicode math expressions, which are erroneously marked incorrect by the autograder. To address this, we add a second LLM step to the WolframAlpha baseline to reformat the WolframAlpha output into a LaTeX formatted answer. In addition, we evaluate four simple majority voting baselines where multiple single-tool solutions are sampled, and the most frequent tool output is taken as the model's answer. We create a majority voting baseline for each of the three tools available to **Multi-TAG**, each of which only sample solutions using the tool it corresponds to. A fourth majority voting baseline is developed that samples solutions using all three tools. For all majority voting baselines, we sample 12 traces per problem to match the maximum of 12 executors per step used in **Multi-TAG**. The multi-tool majority voting baseline achieves this by sampling four traces from each of the three tools.

Tool Augmented Frameworks We also compare **Multi-TAG** against several state-of-the-art tool augmented LLM frameworks, including PAL (Gao et al., 2023), PoT (Chen et al., 2022), ToRA (Gou et al., 2024), MATHSENSEI (Das et al., 2024), and ReAct (Yao et al., 2023). Since the original ToRA work only finetuned older models, such as LLaMA-2 or CodeLLaMA, which are not used in our study, we adapt its approach by prompting newer models to emulate the ToRA reasoning process, using the few-shot prompt from the ToRA paper (which was originally used to generate ToRA traces for the training data). For MATHSENSEI, we use the PG+WA+SG setting, which was reported to achieve the highest accuracy on MATH in the original work. For ReAct, we provide the same three tools available to **Multi-TAG** and write a custom prompt for using these tools.

For all non-majority voting baselines, LLM generations are conducted at temperature 0.0. For the majority voting baselines, LLM generations are conducted at temperature 0.7 and top_p 0.9. The prompts used for all simple baselines are provided in Appendix A. Prompts for the TALM baselines are available in the **Multi-TAG** GitHub [4]

4.4 Main Results

Table 1 presents the results of the baselines and **Multi-TAG** and demonstrates the superior performance of **Multi-TAG** at solving challenging math reasoning problems. Over the three LLMs, the TALM baselines consistently underperform even the simple baselines, demonstrating the inability of these frameworks to address complex math problems. In contrast, **Multi-TAG** outperforms all baselines on all four benchmarks and all three LLMs, demonstrating the effectiveness of multi-tool aggregation at improving the math reasoning abilities of LLMs. When compared to the strongest baseline for each LLM, **Multi-TAG** achieves an average accuracy improvement of 6.6% with LLaMA-3-70B, 6.0% with LLaMA-3.3-70B, and 7.5% with GPT-4o. The improvements are even more substantial when comparing only to the strongest TALM baselines, with improvements rising to 7.9%, 8.4%, and 13.7%, respectively. Furthermore, the consistent improvements achieved by **Multi-TAG** over both open-weight (LLaMA) and proprietary (GPT-4o) models demonstrate its generalizability to different LLM backbones.

5 Analysis

5.1 **Multi-TAG** Improvement Areas

To better understand where **Multi-TAG** improves performance relative to baselines, we compared the performances of the different methods across different MATH500 difficulty levels (ranging from 1-5 where 5 is the hardest level) and across different MATH500 problem subjects.

5.1.1 Problem Difficulty

Figure 2 shows the performance of **Multi-TAG** and baseline methods on different MATH500 difficulty levels. As shown, the improvements from **Multi-TAG** over baselines are especially prominent at higher difficulty levels. At level 5, **Multi-TAG** outperforms all baselines on LLaMA-3-70B by 6.0%, on LLaMA-3.3-70B by 9.7%, and on GPT-4o by 7.5%. These improvements over previous single-tool TALM frameworks demonstrates the effectiveness of multi-tool aggregation as an inference scaling technique for boosting complex math reasoning performance.

[4] Will be released soon

Model	Method	MATH500	AIME	AMC	OlympiadBench	Average
LLaMA-3-70B	CoT	52.2%	1.1%	26.5%	16.6%	24.1%
	Python	45.2%	7.7%	27.7%	17.9%	24.6%
	WolframAlpha Query	23.4%	0.0%	8.4%	6.4%	9.6%
	CoT MV	58.8%	2.2%	27.7%	21.1%	27.5%
	Python MV	52.0%	10.0%	26.5%	21.4%	27.5%
	WolframAlpha MV	25.2%	0.0%	12.0%	7.3%	11.1%
	CoT + Python + WolframAlpha MV	60.6%	5.6%	33.7%	23.6%	30.9%
	PAL	51.2%	12.2%	36.1%	18.7%	29.6%
	PoT	46.8%	8.9%	27.7%	18.6%	25.5%
	ToRA	54.0%	4.4%	27.7%	21.2%	26.8%
	MATHSENSEI	56.4%	3.3%	20.5%	14.5%	23.7%
	ReAct	39.4%	1.1%	13.3%	10.4%	16.1%
	Multi-TAG (Ours)	**68.6%**	**13.3%**	**39.8%**	**28.1%**	**37.5%**
LLaMA-3.3-70B	CoT	75.8%	26.7%	47.0%	32.4%	45.5%
	Python	67.0%	28.9%	47.0%	30.0%	43.2%
	WolframAlpha	45.4%	18.9%	21.7%	12.6%	24.7%
	CoT MV	79.0%	28.9%	55.4%	36.6%	50.0%
	Python MV	73.0%	35.6%	60.2%	32.6%	50.4%
	WolframAlpha MV	45.6%	20.0%	22.9%	13.2%	25.4%
	CoT + Python + WolframAlpha MV	79.0%	33.3%	60.2%	37.6%	52.5%
	PAL	65.8%	24.4%	47.0%	27.5%	41.2%
	PoT	70.2%	28.9%	48.2%	29.8%	44.3%
	ToRA	77.2%	30.0%	54.2%	38.8%	50.1%
	MATHSENSEI	67.4%	15.6%	30.1%	24.8%	34.5%
	ReAct	72.8%	17.8%	21.7%	35.4%	36.9%
	Multi-TAG (Ours)	**84.2%**	**38.9%**	**67.5%**	**43.5%**	**58.5%**
GPT-4o	CoT	79.6%	10.0%	47.0%	32.5%	42.3%
	Python	66.2%	22.2%	50.6%	30.2%	42.3%
	WolframAlpha Query	54.4%	4.4%	22.9%	16.6%	24.6%
	CoT MV	81.8%	12.2%	49.4%	36.5%	45.0%
	Python MV	74.2%	28.9%	59.0%	34.8%	49.2%
	WolframAlpha MV	56.2%	5.6%	22.9%	16.9%	25.4%
	CoT + Python + WolframAlpha MV	86.0%	22.2%	60.2%	38.2%	51.7%
	PAL	64.6%	20.0%	44.6%	28.8%	39.5%
	PoT	51.2%	15.6%	36.1%	19.3%	30.6%
	ToRA	73.0%	17.8%	42.2%	32.1%	41.3%
	MATHSENSEI	73.4%	5.6%	43.4%	28.9%	37.8%
	ReAct	75.2%	28.9%	45.8%	32.1%	45.5%
	Multi-TAG (Ours)	**87.0%**	**34.4%**	**71.1%**	**44.1%**	**59.2%**

Table 1: Main results comparing **Multi-TAG** with various baselines. Best score in each category is **bolded** and second best score is underlined. MV denotes majority voting.

5.1.2 PROBLEM SUBJECT

Figure 3 shows the performance of **Multi-TAG** and baseline methods on different MATH500 problem subjects. **Multi-TAG** outperforms all baselines in 12/21 subjects in total across the three models, demonstrating its consistent effectiveness across a diverse range of math domains. Moreover, comparing the four simple majority voting baselines, the multi-tool majority voting baseline (CoT + Py + WA) outperformed all three single-tool majority voting baselines in 12/21 subjects in total across the three models. This highlights the synergistic benefits of aggregating different tools together, improving upon the performance of aggregating each of the tools individually.

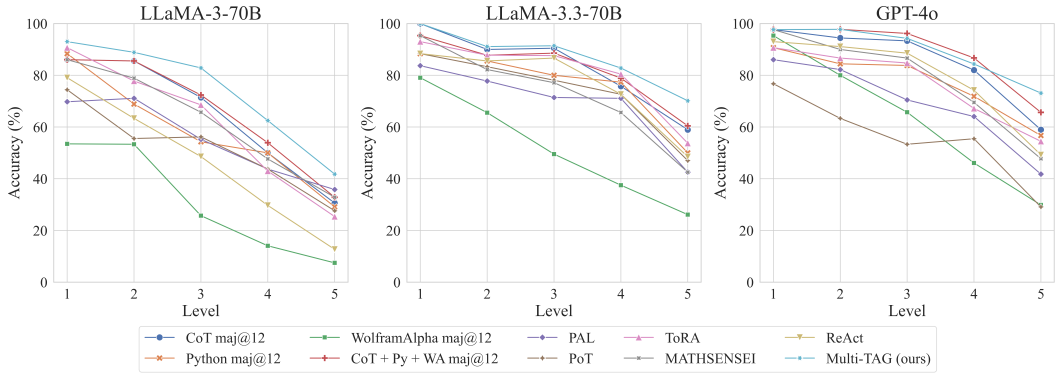

Figure 2: Comparison of baseline methods and **Multi-TAG** on different MATH500 difficulty levels (higher levels contain more difficult problems). As shown, **Multi-TAG** outperforms baselines most substantially on the more challenging problems.

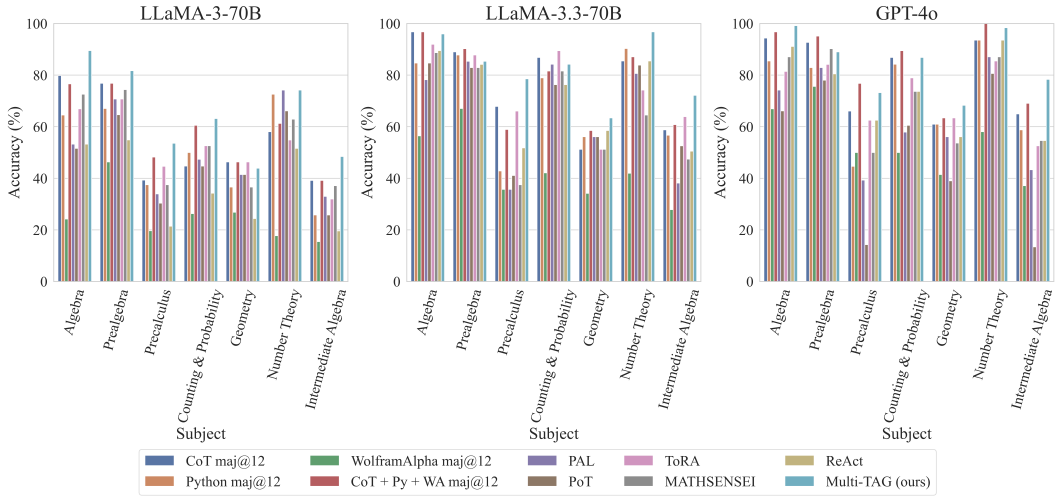

Figure 3: Comparison of baseline methods and **Multi-TAG** on different MATH500 problem subjects. **Multi-TAG** consistently performs well across subjects, outperforming all baselines on a majority of subjects. Furthermore, simple multi-tool aggregation (CoT + Py + WA) also outperforms the three single-tool aggregation baselines on a majority of subjects.

5.2 Ablation Study

5.2.1 Token Consumption Cost

We verify that the improvements from **Multi-TAG** over baselines are not simply a result of **Multi-TAG** utilizing more LLM inference compute (i.e. using more tokens). To do so, for each of our simple majority voting baselines, we modify the number of sampled LLM traces per problem so that each baseline and **Multi-TAG** have matching token consumption costs. Similarly, for each of our TALM baselines, to increase the amount of tokens used to match **Multi-TAG**, we simply sample multiple TALM traces for each problem and apply majority voting over the final answers reached by the traces.

Token consumption cost is defined as $0.25P + O$, where P is the number of prompt tokens and O is the number of generated output tokens. The 0.25 weighting of prompt token cost is based on OpenAI's GPT-4o API pricing, which as of time of writing is $\$2.50$ per million prompt tokens and $\$10$ per million output tokens.

We evaluated **Multi-TAG** and our token consumption-matched baselines and report the results in Table 2. For cost-related reasons, we only evaluate GPT-4o as the backbone LLM for these experiments. **Multi-TAG** continues to outperform the token-matched baselines, achieving superior results on all four benchmarks and achieving a 7.7% average accuracy improvement over the strongest baseline.

Method	MATH500	AIME	AMC	OlympiadBench	Average
CoT (maj@19)	84.2%	10.0%	51.8%	37.7%	45.9%
Python (maj@35)	75.2%	30.0%	63.9%	35.4%	51.1%
WolframAlpha (maj@70)	55.6%	7.8%	22.9%	15.5%	25.5%
CoT + Python + WolframAlpha (maj@33)	84.2%	22.2%	60.2%	39.0%	51.4%
PAL (maj@34)	71.8%	27.8%	57.8%	32.5%	47.5%
PoT (maj@14)	75.4%	32.2%	61.4%	35.8%	51.2%
ToRA (maj@6)	82.2%	27.8%	55.4%	40.7%	51.5%
MATHSENSEI (maj@3)	78.4%	13.3%	44.6%	29.2%	41.4%
ReAct (maj@6)	81.0%	22.2%	62.7%	38.2%	51.0%
Multi-TAG (Ours)	**87.0%**	**34.4%**	**71.1%**	**44.1%**	**59.2%**

Table 2: Results of **Multi-TAG** and token consumption-matched baselines. The number of sampled traces per problem used for each token consumption-matched baseline is given as maj@x. For simple multi-tool majority voting (CoT + Python + WolframAlpha), the 33 traces are split evenly between CoT, Python, and WolframAlpha traces. Best score in each category is **bolded** and second best score is underlined. GPT-4o is used as the LLM for these experiments.

5.2.2 Consistency Threshold

To verify that **Multi-TAG**'s consistency threshold effectively reduces the token consumption cost while incurring minimal performance degradation, we compare the results of running **Multi-TAG** with a consistency threshold of 2 and the results of running **Multi-TAG** without a consistency threshold. We also vary the max executors parameter and the backbone LLM to ensure the effectiveness of the consistency threshold for all settings. We report the MATH500 accuracy and average token consumption cost (as defined in Section 5.2.1) per problem for each of the settings in Table 3.

As shown, the accuracy degradation incurred by applying the consistency threshold is minimal, with a maximum degradation of 2.0% and an average degradation of 0.1%. Meanwhile, the token consumption cost is significantly reduced in all settings, with relative reductions ranging from 14.7% to 63.6% and an average relative reduction of 43.8%. Thus, the consistency threshold effectively reduces the computational cost of **Multi-TAG** without compromising its performance.

	LLaMA-3-70B		LLaMA-3.3-70B		GPT-4o	
Max Executors	With Threshold	Without Threshold	With Threshold	Without Threshold	With Threshold	Without Threshold
6	67.0% ↑ 0.2% (5361 ↓ 14.7%)	66.8% (6286)	82.0% ↓ 1.8% (5967 ↓ 37.6%)	83.8% (9559)	82.6% ↓ 1.4% (6090 ↓ 23.8%)	84.0% (7989)
9	66.2% ↓ 0.4% (6746 ↓ 25.8%)	66.6% (9091)	85.8% ↑ 1.4% (7766 ↓ 44.0%)	84.4% (13859)	86.4% ↑ 1.4% (6157 ↓ 40.9%)	85.0% (10425)
12	68.6% ↑ 0.2% (7916 ↓ 34.7%)	68.4% (12124)	84.2% ↓ 2.0% (7945 ↓ 56.3%)	86.2% (18190)	87.0% ↑ 0.8% (7952 ↓ 48.3%)	86.2% (15376)
15	68.6% ↓ 1.2% (8891 ↓ 40.4%)	69.8% (14922)	86.0% ↑ 0.6% (9274 ↓ 58.8%)	85.4% (22501)	87.6% ↑ 1.2% (8214 ↓ 52.0%)	86.4% (17127)
18	67.8% ↓ 1.2% (9918 ↓ 56.7%)	69.0% (17507)	86.6% ↑ 1.0% (9727 ↓ 63.6%)	85.6% (26743)	85.6% ↓ 0.4% (9005 ↓ 58.8%)	86.0% (21883)

Table 3: MATH500 scores and average token consumption costs (as defined in Section 5.2.1) per problem of **Multi-TAG** with and without the consistency threshold. Token consumption costs are in (parentheses).

5.2.3 Candidate Step Selection

To verify the efficacy of the candidate step selection algorithm, we study the effects of simplifying the procedure on performance and token consumption cost (as defined in Section 5.2.1). Recall that the procedure consists of the following two steps:

(**1**) Identify the most frequent candidate final answer and mark all candidate steps reaching this final answer.
(**2**) From the marked candidates, select the candidate with the shortest solution completion.

We compare four approaches: Full (the unmodified algorithm from **Multi-TAG** with both **(1)** and **(2)**), Answer Only (replacing **(2)** with randomly selecting a marked candidate), Length Only (replacing **(1)** with marking all candidates), and Random (select a random candidate without using either **(1)** or **(2)**). We evaluate **Multi-TAG** with each of the modified candidate selection procedures on MATH500 and report the results in Table 4.

As shown, all of the simplified candidate selection procedures significantly underperform the Full procedure. On average, the performance degradation is 2.6% for Answer Only, 5.5% for Length Only, and 7.9% for Random. This demonstrates the necessity of both steps of the algorithm to maximize performance. Furthermore, the results show the isolated contribution of **(2)** to computational efficiency. The only difference between Full and Answer Only is the inclusion of **(2)** in the former, which reduces the token consumption cost by 27.9% on average. Similarly, the only difference between Length Only and Random is the inclusion of **(2)** in the former, which reduces the token consumption cost by 25.5% on average. These results demonstrate that **(2)** additionally improves the computational efficiency of **Multi-TAG**.

Next Step Selection Procedure	**LLaMA-3-70B**	**LLaMA-3.3-70B**	**GPT-4o**
Full	68.6% (7916)	84.2% (7945)	87.0% (7952)
Answer Only	64.8% ↓ 3.8% (9492 ↑ 19.9%)	83.2% ↓ 1.0% (11538 ↑ 45.2%)	84.0% ↓ 3.0% (9420 ↑ 18.5%)
Length Only	56.8% ↓ 11.8% (7548 ↓ 4.6%)	82.0% ↓ 2.2% (9325 ↑ 17.4%)	84.6% ↓ 2.4% (7412 ↓ 6.8%)
Random	54.8% ↓ 13.8% (10023 ↑ 26.6%)	78.8% ↓ 5.4% (12635 ↑ 59.0%)	82.4% ↓ 4.6% (9976 ↑ 25.5%)

Table 4: MATH500 scores and average token consumption cost (as defined in Section 5.2.1) per problem of **Multi-TAG** with the proposed and simplified candidate step selection procedures. Token consumption costs are in (parentheses).

5.3 Hyperparameters Study

We investigated the influence of **Multi-TAG**'s two primary hyperparameters-the maximum number of executors and the consistency threshold value-on its performance and computational cost. We evaluate **Multi-TAG** with various hyperparameter configurations and with all three backbone LLMs on MATH500. The results are reported in Table 5.

The results show a strong, statistically significant positive correlation between performance and the max executors value. The Spearman correlation coefficients were .832 ($p < .01$), .535 ($p = .04$), and .549 ($p = .03$) for LLaMA-3-70B, LLaMA-3.3-70B, and GPT-4o results, respectively. In contrast, the consistency threshold value showed no statistically significant correlation with performance, with coefficients of .057 ($p = .84$), .028 ($p = .92$), and .162 ($p = .56$). Thus, to increase performance, the max executors value should be increased.

While increasing the max executors value boosts performance, it also significantly increases computational costs. The results demonstrate the crucial role of the consistency threshold to mitigate this increase. For instance, when increasing max executors from 6 to 18, the average increase in token consumption cost across all models was only 49.3% with a consistency threshold of 1. This cost increase was substantially higher for thresholds of 2 (65.3%) and 3 (70.4%). This demonstrates that lower consistency threshold values effectively contain costs, especially for larger max executors settings.

These findings suggest a simple heuristic for setting **Multi-TAG** hyperparameters: the max executors value should be set as high as the compute budget allows to maximize performance, then the

consistency threshold value should be set to a low value, such as 1 or 2, to minimize the token consumption cost.

Model	Consistency Threshold	Max Executors				
		6	**9**	**12**	**15**	**18**
LLaMA-3-70B	**1**	63.0% (4518)	67.2% (5592)	66.2% (6565)	67.8% (7272)	69.2% (7799)
	2	67.0% (5361)	66.2% (6746)	68.6% (7916)	68.6% (8891)	67.8% (9918)
	3	63.8% (5838)	66.6% (7380)	67.2% (9267)	68.0% (10074)	68.8% (11418)
LLaMA-3.3-70B	**1**	84.0% (5023)	84.8% (5913)	83.6% (6400)	86.2% (6752)	85.0% (6793)
	2	82.0% (5967)	85.8% (7766)	84.2% (7945)	86.0% (9274)	86.6% (9727)
	3	83.8% (6460)	86.0% (9061)	84.6% (9289)	85.6% (10497)	84.4% (10555)
GPT-4o	**1**	86.0% (4711)	85.0% (4822)	86.0% (6098)	86.2% (6297)	86.6% (6594)
	2	82.6% (6090)	86.4% (6157)	87.0% (7952)	87.6% (8214)	86.2% (9008)
	3	85.0% (7064)	87.0% (7352)	86.2% (9253)	85.6% (9556)	87.2% (10753)

Table 5: MATH500 scores and average token consumption cost (as defined in Section 5.2.1) per problem of various max executor, consistency threshold configurations of **Multi-TAG**. Token consumption costs are in (parentheses).

6 CONCLUSION

In this paper, we present **Multi-TAG** , a novel tool-augmented LLM framework for math reasoning. Unlike previous TALM frameworks, **Multi-TAG** scales up the inference-time compute allocated by allowing the LLM to invoke and aggregate multiple tools at each reasoning step. As a result, **Multi-TAG** achieves superior results on four complex math reasoning benchmarks, outperforming the strongest baselines by 6.0% to 7.5% over three different backbone LLMs. Furthermore, **Multi-TAG** is widely applicable, enabling the use of any general instruction-tuned LLM and enabling computational costs to be tuned according to specific cost/performance requirements. Our results demonstrate that multi-tool aggregation is a promising avenue for future work on advancing LLM math reasoning capabilities.

REFERENCES

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 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*, pp. 225–237, 2024.

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

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. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021a.

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. Training verifiers to solve math word problems, 2021b. URL https://arxiv.org/abs/2110.14168.

Debrup Das, Debopriyo Banerjee, Somak Aditya, and Ashish Kulkarni. Mathsensei: A tool-augmented large language model for mathematical reasoning. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 942–966, 2024.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

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

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Minlie Huang, Nan Duan, Weizhu Chen, et al. Tora: A tool-integrated reasoning agent for mathematical problem solving. In *The Twelfth International Conference on Learning Representations*, 2024.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking, 2025. URL https://arxiv.org/abs/2501.04519.

Masoud Hashemi, Oluwanifemi Bamgbose, Sathwik Tejaswi Madhusudhan, Jishnu Sethumadhavan Nair, Aman Tiwari, and Vikas Yadav. Dna bench: When silence is smarter-benchmarking over-reasoning in reasoning llms. *CoRR*, 2025.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, Jie Liu, Lei Qi, Zhiyuan Liu, and Maosong Sun. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. In *The 62nd Annual Meeting of the Association for Computational Linguistics*, 2024.

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

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. URL http://arxiv.org/abs/2310.06825.

Hynek Kydlíček. Math-Verify: Math Verification Library. URL https://github.com/huggingface/math-verify.

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

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candes, and Tatsunori Hashimoto. s1: Simple test-time scaling. In *Workshop on Reasoning and Planning for Large Language Models*.

OpenAI. Chatgpt: Optimizing language models for dialogue, 2022. URL https://openai.com/blog/chatgpt. Accessed: 2025-05-20.

OpenAI. Gpt-4 technical report, 2023. URL https://openai.com/research/gpt-4. Accessed: 2025-05-20.

OpenAI. Openai o1 system card, 2024. URL https://openai.com/index/openai-o1-system-card/.

Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014*, 2023.

Arkil Patel, Satwik Bhattamishra, and Navin Goyal. Are NLP models really able to solve simple math word problems? In Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (eds.), *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2080–2094, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.168. URL https://aclanthology.org/2021.naacl-main.168/.

Meghana Rajeev, Rajkumar Ramamurthy, Prapti Trivedi, Vikas Yadav, Oluwanifemi Bamgbose, Sathwik Tejaswi Madhusudan, James Zou, and Nazneen Rajani. Cats confuse reasoning llm: Query agnostic adversarial triggers for reasoning models. *arXiv preprint arXiv:2503.01781*, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

Parshin Shojaee, Iman Mirzadeh, Keivan Alizadeh, Maxwell Horton, Samy Bengio, and Mehrdad Farajtabar. The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity, 2025. URL https://arxiv.org/abs/2506.06941.

Zayne Rea Sprague, Fangcong Yin, Juan Diego Rodriguez, Dongwei Jiang, Manya Wadhwa, Prasann Singhal, Xinyu Zhao, Xi Ye, Kyle Mahowald, and Greg Durrett. To cot or not to cot? chain-of-thought helps mainly on math and symbolic reasoning. In *The Thirteenth International Conference on Learning Representations*.

Yang Sui, Yu-Neng Chuang, Guanchu Wang, Jiamu Zhang, Tianyi Zhang, Jiayi Yuan, Hongyi Liu, Andrew Wen, Shaochen Zhong, Hanjie Chen, et al. Stop overthinking: A survey on efficient reasoning for large language models. *arXiv preprint arXiv:2503.16419*, 2025.

Llama Team. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning. In *ICLR*, 2024a.

Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y. Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations, 2024b. URL https://arxiv.org/abs/2312.08935.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Yuanzhen Xie, Tao Xie, Mingxiong Lin, WenTao Wei, Chenglin Li, Beibei Kong, Lei Chen, Chengxiang Zhuo, Bo Hu, and Zang Li. Olagpt: Empowering llms with human-like problem-solving abilities. *arXiv preprint arXiv:2305.16334*, 2023.

An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, Keming Lu, Mingfeng Xue, Runji Lin, Tianyu Liu, Xingzhang Ren, and Zhenru Zhang. Qwen2.5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

Shuo Yin, Weihao You, Zhilong Ji, Guoqiang Zhong, and Jinfeng Bai. Mumath-code: Combining tool-use large language models with multi-perspective data augmentation for mathematical reasoning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 4770–4785, 2024.

Tong Yu, Yongcheng Jing, Xikun Zhang, Wentao Jiang, Wenjie Wu, Yingjie Wang, Wenbin Hu, Bo Du, and Dacheng Tao. Benchmarking reasoning robustness in large language models. *arXiv preprint arXiv:2503.04550*, 2025.

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

Weihao Zeng, Yuzhen Huang, Qian Liu, Wei Liu, Keqing He, Zejun Ma, and Junxian He. Simplerl-zoo: Investigating and taming zero reinforcement learning for open base models in the wild, 2025. URL https://arxiv.org/abs/2503.18892.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, and Hongsheng Li. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification, 2023. URL https://arxiv.org/abs/2308.07921.

A LLM PROMPTS

We provide the LLM prompts used for all components of **Multi-TAG** in Section A.1 and the prompts used for all simple baselines in Section A.2. Prompts used for TALM baselines can be found in the **Multi-TAG** GitHub [5].

A.1 MULTI-TAG PROMPTS

CoT Executor System Prompt

```
You are a math problem solving agent working on solving a problem iteratively. The
problem and the current progress will be given below. The current progress consists of a
sequence of steps separated by "---" which may consist of natural language reasoning,
Python scripts, and WolframAlpha queries. Python script execution outputs are given at
the bottom of a step within ``` output ```, and WolframAlpha query results are given at
the bottom of a step within ``` result ```. Your task is to write the next step in the
solution in the form of natural language reasoning.

If the solution is complete, you may give the final answer (NOTE: you may not give the
final answer if you also write a step. Only give the final answer if the solution is
complete without you writing an additional step). Express the answer using LaTeX
formatting and do not include units or other unnecessary text in the answer. It's okay to
leave the final answer unsimplified, for example expressed as a decimal. Do not round
final answers that are decimals. Make sure to read the question carefully and answer
exactly what the problem is asking for. Format the answer by enclosing the answer within
<final_answer></final_answer> and putting the answer within \boxed{{}}. For example:
<final_answer>
The final answer is \boxed{{[final answer formatted using LaTeX]}}
</final_answer>
```

Python Executor System Prompt

```
You are a math problem solving agent working on solving a problem iteratively. The
problem and the current progress will be given below. The current progress consists of a
sequence of steps separated by "---" which may consist of natural language reasoning,
Python scripts, and WolframAlpha queries. Python script execution outputs are given at
the bottom of a step within ``` output ```, and WolframAlpha query results are given at
the bottom of a step within ``` result ```. Your task is to write the next step in the
solution in the form of a Python script and a brief explanation of what your script
calculates.

If the solution is complete, you may give the final answer (NOTE: you may not give the
final answer if you also write a step. Only give the final answer if the solution is
complete without you writing an additional step). Express the answer using LaTeX
formatting and do not include units or other unnecessary text in the answer. It's okay to
leave the final answer unsimplified, for example expressed as a decimal. Do not round
final answers that are decimals. Make sure to read the question carefully and answer
exactly what the problem is asking for. Format the answer by enclosing the answer within
<final_answer></final_answer> and putting the answer within \boxed{{}}. For example:
<final_answer>
The final answer is \boxed{{[final answer formatted using LaTeX]}}
</final_answer>

To write the next step, you must follow the following format:
``` python
[Python script, assigning the desired output to the ` result ` global variable]
```
[Brief explanation of what your script calculates]
```

[5] Will be released soon

WolframAlpha Executor System Prompt

You are a math problem solving agent working on solving a problem iteratively. The problem and the current progress will be given below. The current progress consists of a sequence of steps separated by "---" which may consist of natural language reasoning, Python scripts, and WolframAlpha queries. Python script execution outputs are given at the bottom of a step within ``` output ```, and WolframAlpha query results are given at the bottom of a step within ``` result ```. Your task is to write the next step in the solution in the form of a WolframAlpha query and a brief explanation of what your query calculates.

If the solution is complete, you may give the final answer (NOTE: you may not give the final answer if you also write a step. Only give the final answer if the solution is complete without you writing an additional step). Express the answer using LaTeX formatting and do not include units or other unnecessary text in the answer. It's okay to leave the final answer unsimplified, for example expressed as a decimal. Do not round final answers that are decimals. Make sure to read the question carefully and answer exactly what the problem is asking for. Format the answer by enclosing the answer within <final_answer></final_answer> and putting the answer within \boxed{{}}. For example:
<final_answer>
The final answer is \boxed{{[final answer formatted using LaTeX]}}
</final_answer>

To write the next step, you must follow the following format:
```wolfram
[WolframAlpha query]
```
[Brief explanation of what your query calculates]

Executor User Prompt

Problem
{problem}

Partial Solution
{progress}

Final Instructions
Above are the problem and potentially incomplete solution. Note that the partial solution has already been verified for accuracy, so you should assume it is correct. Write the next step or give the final answer if the partial solution is complete. Remember that you must write a step of the specified form above (or give the final answer using the specific format above). You must write a single logical step (or give the final answer), and stop after completing a single step.

Solution Completion System Prompt

You are a math problem solver working on completing a partial solution to a problem. The problem and partial solution will be given below. The partial solution consists of a sequence of steps separated by "---" which may consist of natural language reasoning, Python scripts, and WolframAlpha queries. Python script execution outputs are given at the bottom of a step within ``` output ```, and WolframAlpha query results are given at the bottom of a step within ``` result ```. Your task is to continue the partial solution to finish solving the problem. You may only use natural language reasoning in your response (you may not use Python or WolframAlpha). Enclose the final answer within \boxed{}. Express the answer using LaTeX formatting and do not include units or other unnecessary text in the answer. It's okay to leave the final answer unsimplified, for example expressed as a decimal. Do not round final answers that are decimals. Make sure to read the question carefully and answer exactly what the problem is asking for.

Solution Completion User Prompt

```
# Problem
{problem}

# Partial Solution
{progress}

# Final Instructions
Above are the problem and partial solution to continue. Note that the partial solution
has already been verified for accuracy, so you should assume it is correct. Continue the
partial solution to finish solving the problem. You don't need to follow any specific
format like the step-by-step format of the partial solution.
```

A.2 Baseline Prompts

CoT User Prompt

```
Solve the following problem step by step. Express the final answer using LaTeX
formatting and enclose it within \boxed{{}}. Do not include units or other unnecessary
text in the answer.

{problem}
```

Python User Prompt

```
Solve the following problem by writing a single Python script. Your script should be
enclosed within ``` python ```. You may only write a single Python script in your
response. Do not include units or other unnecessary text in the answer. Your answer
should be submitted by assigning the answer to the "result" global variable.

{problem}
```

WolframAlpha User Prompt

```
Solve the following problem by writing a single WolframAlpha query. Your WolframAlpha
query must be enclosed within ``` wolfram ```. You may only write a single WolframAlpha
query in your response.

{problem}
```

Convert WolframAlpha Output to LaTeX User Prompt

```
Here is the output from your query:
``` result
{output}
```

Now, give the final answer to the problem. Express the final answer using LaTeX
formatting and enclose it within \boxed{{}}. Do not include units or other unnecessary
text in the answer. Your response must only contain the final answer and nothing else.
```

B Costs Analysis

B.1 Main Results Token Consumption Cost

Table 6 reports the average token consumption cost (as defined in Section 5.2.1) per problem for all results in the main results (Table 1).

Model	Method	MATH500	AIME	AMC	OlympiadBench	Average
LLaMA-3-70B	CoT	348	790	583	521	561
	Python	97	153	128	135	128
	WolframAlpha Query	130	194	158	168	163
	CoT MV	3757	6220	5127	5922	5257
	Python MV	1160	1832	1629	1644	1566
	WolframAlpha MV	1584	2318	1939	2049	1973
	CoT + Python + WolframAlpha MV	2236	3928	2868	3203	3059
	PAL	280	355	379	318	333
	PoT	566	621	610	613	603
	ToRA	1261	1437	1299	1420	1354
	MATHSENSEI	2730	3298	3049	3139	3054
	ReAct	1386	2242	1784	1883	1824
	Multi-TAG (Ours)	7916	14448	12125	12567	11764
LLaMA-3.3-70B	CoT	610	929	885	886	828
	Python	206	457	319	250	308
	WolframAlpha	148	231	184	197	190
	CoT MV	7303	11813	10877	10558	10138
	Python MV	2448	4326	3946	3049	3442
	WolframAlpha MV	1808	3167	2329	2330	2409
	CoT + Python + WolframAlpha MV	3845	6381	5572	5286	5271
	PAL	386	677	585	475	531
	PoT	609	817	753	697	719
	ToRA	2149	3824	2610	2913	2874
	MATHSENSEI	3974	5247	5174	5099	4874
	ReAct	1436	2627	3181	2295	2385
	Multi-TAG (Ours)	7945	17809	11756	14186	12924
GPT-4o	CoT	582	992	909	837	830
	Python	287	521	353	390	388
	WolframAlpha Query	129	234	233	197	198
	CoT MV	6485	9889	9306	8885	8641
	Python MV	3195	5375	5142	4373	4521
	WolframAlpha MV	1614	2847	2466	2205	2283
	CoT + Python + WolframAlpha MV	3519	5854	5096	5193	4916
	PAL	328	442	426	394	398
	PoT	837	1140	1013	1006	999
	ToRA	1520	3801	2686	2329	2584
	MATHSENSEI	3915	5579	4948	5009	4863
	ReAct	1518	3355	2857	2496	2557
	Multi-TAG (Ours)	7952	18650	12285	13823	13178

Table 6: Average token consumption cost (as defined in Section 5.2.1) per problem for all results in Table 1 .

B.2 **MULTI-TAG** API CALLS CONSUMPTION

Table 7 reports the average number of LLM API calls per step, average number of steps used per problem, and average number of LLM API calls per problem for **Multi-TAG** results in Table 1.

C FULL **MULTI-TAG** TRACE

Below is a full **Multi-TAG** solution trace demonstrating **Multi-TAG**'s ability to effectively leverage different tools to solve a challenging math problem. This solution trace is extracted from the GPT-4o MATH500 main results.

Problem:

Model	MATH500			AIME			AMC			OlympiadBench		
	Calls per Step	Steps per Problem	Calls per Problem	Calls per Step	Steps per Problem	Calls per Problem	Calls per Step	Steps per Problem	Calls per Problem	Calls per Step	Steps per Problem	Calls per Problem
LLaMA-3-70B	9.99	2.85	28.49	13.56	2.79	37.82	11.86	2.93	34.72	12.09	2.96	35.79
LLaMA-3.3-70B	7.31	2.52	18.41	9.67	2.72	26.31	8.21	2.58	21.16	9.20	2.66	24.52
GPT-4o	7.98	2.39	19.11	11.47	2.69	30.84	9.44	2.47	23.33	10.25	2.56	26.25

Table 7: Average number of LLM API calls per step, average number of steps per problem, and average number of LLM API calls per problem for **Multi-TAG** results in Table 1

.

Find the maximum value of

$$f(x, y) = x\sqrt{1 - y^2} + y\sqrt{1 - x^2},$$

where $-1 \leq x, y \leq 1$.

Step 1 (CoT):

To find the maximum value of the function

$$f(x, y) = x\sqrt{1 - y^2} + y\sqrt{1 - x^2},$$

we can start by using the method of Lagrange multipliers. We will define a new function

$$g(x, y) = x^2 + y^2 - 1$$

to represent the constraint $x^2 + y^2 = 1$.

The method of Lagrange multipliers states that we need to solve the system of equations given by

$$\nabla f = \lambda \nabla g$$

and the constraint equation $g(x, y) = 0$.

First, we will compute the gradients of f and g.

The gradient of f is:

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right).$$

The gradient of g is:

$$\nabla g = \left(\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y} \right).$$

Let's find these partial derivatives.

Step 2 (Python):

Python Code

```
import sympy as sp

# Define the variables
x, y = sp.symbols('x y')

# Define the function f and the constraint g
f = x * sp.sqrt(1 - y**2) + y * sp.sqrt(1 - x**2)
g = x**2 + y**2 - 1

# Calculate the gradient of f
grad_f = [sp.diff(f, var) for var in (x, y)]

# Calculate the gradient of g
grad_g = [sp.diff(g, var) for var in (x, y)]

result = (grad_f, grad_g)
```

This script calculates the gradients of the functions f and g. The gradients are essential for setting up the system of equations for the method of Lagrange multipliers.

Python Code Output

```
([-x*y/ sqrt (1 - x**2) + sqrt (1 - y**2), -x*y/sqrt (1 - y**2) + sqrt (1 - x**2)], [2*x, 2*
y])
```

Step 3 (WolframAlpha):

WolframAlpha Query

```
solve [-x*y/ sqrt (1 - x**2) + sqrt (1 - y**2) = 2*x lambda, -x*y/sqrt (1 - y**2) + sqrt (1 -
 x**2) = 2*y lambda, x^2 + y^2 = 1]
```

This query solves the system of equations derived from the gradients of f and g using the method of Lagrange multipliers, along with the constraint equation $x^2 + y^2 = 1$.

WolframAlpha Result

```
y = -sqrt (1 - x\^2), lambda = 0, y = sqrt (1 - x\^2), lambda = 0
```

Step 4 (CoT):

Given the solutions $y = \sqrt{1 - x^2}$ and $y = -\sqrt{1 - x^2}$ from the Lagrange multipliers method, we need to evaluate the function $f(x, y)$ at these points to find the maximum value.

First, consider $y = \sqrt{1 - x^2}$:

$$\begin{aligned} f(x, \sqrt{1 - x^2}) &= x\sqrt{1 - (\sqrt{1 - x^2})^2} + \sqrt{1 - x^2}\sqrt{1 - x^2} \\ &= x\sqrt{1 - (1 - x^2)} + (1 - x^2) \\ &= x\sqrt{x^2} + (1 - x^2) \\ &= x|x| + (1 - x^2) \end{aligned}$$

Since $-1 \leq x \leq 1$, $x|x| = x^2$:

$$\begin{aligned} f(x, \sqrt{1 - x^2}) &= x^2 + (1 - x^2) \\ &= 1 \end{aligned}$$

Next, consider $y = -\sqrt{1 - x^2}$:

$$\begin{aligned} f(x, -\sqrt{1 - x^2}) &= x\sqrt{1 - (-\sqrt{1 - x^2})^2} + (-\sqrt{1 - x^2})\sqrt{1 - x^2} \\ &= x\sqrt{1 - (1 - x^2)} - (1 - x^2) \\ &= x\sqrt{x^2} - (1 - x^2) \\ &= x|x| - (1 - x^2) \end{aligned}$$

Again, since $-1 \leq x \leq 1$, $x|x| = x^2$:

$$\begin{aligned} f(x, -\sqrt{1 - x^2}) &= x^2 - (1 - x^2) \\ &= x^2 - 1 + x^2 \\ &= 2x^2 - 1 \end{aligned}$$

The maximum value of $2x^2 - 1$ over the interval $-1 \leq x \leq 1$ is 1, which occurs at $x = \pm 1$.

Thus, the maximum value of $f(x, y)$ is $\boxed{1}$.