From Awareness to Adaptability: Enhancing Tool Utilization for Scientific Reasoning

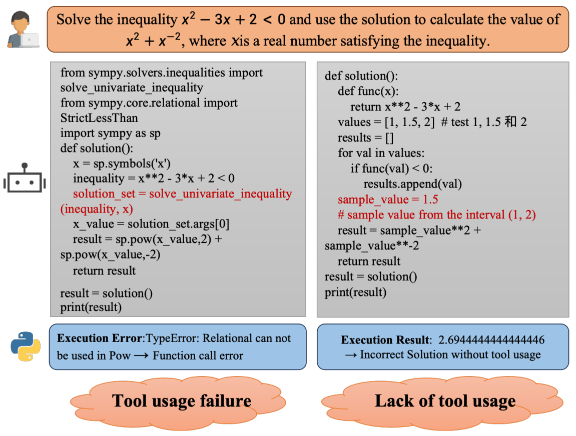
Wenjing Xie[1], **Xiaobo Liang**[1], **Juntao Li**[1*], **Wanfu Wang**[1]
Qiaoming Zhu[1], **Kehai Chen**[2], **Min Zhang**[1]
[1]School of Computer Science and Technology, Soochow University
[2]Harbin Institute of Technology, Shenzhen, China
{wjxie,xbliang3,wfwang}@stu.suda.edu.cn
{ljt,qmzhu,minzhang}@suda.edu.cn, chenkehai@hit.edu.cn

Abstract

As large language models (LLMs) are increasingly applied to complex scientific problem-solving, their effectiveness is often limited by unconscious or failed tool usage. To address this issue, we introduce the Tool-Awareness Training (TAT) method, designed to enhance scientific reasoning. This approach leverages both forward and backward data generation strategies to strengthen the model's conscious and selective tool utilization in multi-step reasoning tasks. Our method unfolds in three stages: (1) developing tool-knowledge through backward tooluse data generation (2) enhancing tool-awareness in multi-step reasoning by utilizing forward reasoning data, and (3) improving domain adaptability through large-scale domain-specific data for multi-task learning. These three stages progressively establish the foundation for tool learning and scientific reasoning, effectively integrating both, enabling the model to tackle multi-domain scientific tasks while optimizing tool usage. Our experimental results demonstrate that TAT significantly enhances LLM performance in mathematical and scientific reasoning tasks, particularly by improving the model's tool utilization capabilities, including proactivity and execution success rates.

1 Introduction

Recent advancements in large language models (LLMs) have significantly expanded AI's capabilities, excelling in complex reasoning tasks and showing promising potential for scientific research and discovery (Ma et al., 2024; Kumar et al., 2023; Rane et al., 2023). However, current studies consistently report that LLMs' capabilities in scientific domains are largely confined to a high school level (Rein et al., 2023; Wang et al., 2023b), thereby falling short in tackling more complex, specialized


Solve the inequality $x^2 - 3x + 2 < 0$ and use the solution to calculate the value of $x^2 + x^{-2}$, where xis a real number satisfying the inequality.

```
from sympy.solvers.inequalities import
solve_univariate_inequality
from sympy.core.relational import
StrictLessThan
import sympy as sp
def solution():
   x = sp.symbols('x')
   inequality = x**2 - 3*x + 2 < 0
   solution_set = solve_univariate_inequality
(inequality, x)
   x_value = solution_set.args[0]
   result = sp.pow(x_value,2) +
sp.pow(x_value,-2)
   return result

result = solution()
print(result)
```

```
def solution():
   def func(x):
      return x**2 - 3*x + 2
   values = [1, 1.5, 2]  # test 1, 1.5 和 2
   results = []
   for val in values:
      if func(val) < 0:
         results.append(val)
   sample_value = 1.5
   # sample value from the interval (1, 2)
   result = sample_value**2 +
sample_value**-2
   return result
result = solution()
print(result)
```

Execution Error:TypeError: Relational can not be used in Pow ⟶ Function call error

Execution Result: 2.6944444444444446 → Incorrect Solution without tool usage

Figure 1: Illustration of two error types: (1) Tool Usage failure, reflecting challenges in accurately invoking unfamiliar or complex functions, and (2) Lack of Tool Usage,stemming from the failure to proactively leverage Python libraries for computational optimization.

problems. This limitation underscores the importance of integrating LLMs with domain-specific tools, as Python-based solutions (Gou et al., 2023) coupled with specialized scientific tools (Schick et al., 2024; Lu et al., 2024a; Yuan et al., 2023).

Existing approaches enhance reasoning through Chain-of-Thought (Wei et al., 2022), Program-of-Thought (Chen et al., 2022), and Tool-Integration (Gou et al., 2023), primarily relying on Python interpreters for computation. However, they overlook active tool awareness and the use of advanced tools like Python libraries, as shown in Figure 11. LLMs struggle with proactive library usage and invoking complex functions, reducing problem-solving efficiency (Gou et al., 2023). Addressing these gaps is crucial for improving LLMs' effectiveness in advanced scientific reasoning.

Inspired by human learning behavior as detailed in Section 3, we propose the Tool-Awareness Training (TAT) framework to enhance the ability of LLMs to effectively utilize tools in multi-step reasoning tasks. TAT combines forward and backward data generation strategies to guide the model's tool

[*] Corresponding author.

usage. Initially, we build tool knowledge through backward data, designing problems around tools to establish a foundational understanding. Next, we enhance multi-step reasoning with forward data that helps the model decide when to use tools. Finally, large-scale domain-specific data is used to fine-tune the model through multi-task learning, optimizing performance across various scientific fields. The core idea of TAT is to teach the model when and how to apply external tools, especially specialized Python libraries, empowering the model to solve complex problems by applying the right tools at the right time.

The effectiveness of our approach is validated on scientific, mathematical, and tabular reasoning tasks. We evaluate the model's performance across several aspects, including accuracy, code executability, and tool usage proficiency. Our results show that TAT substantially enhances the model's accuracy and tool utilization in solving complex problems requiring tool integration.

2 Related Work

Scientific Reasoning Recent benchmarks have been proposed to evaluate the scientific problem-solving capabilities of LLMs (Lu et al., 2022; Wang et al., 2023b; Arora et al., 2023). Current research has mainly focused on three prominent approaches: Chain of Thought (CoT) (Wei et al., 2022), Program-of-Thought (PoT) (Chen et al., 2022), and tool-integrated reasoning (Tool-based) (Gou et al., 2023). CoT leverages natural language to guide multi-step reasoning, PoT employs procedural code to optimize computation, while tool-based approaches aim to integrate the semantic reasoning strengths of CoT with the precise computational abilities of PoT, leading to significant improvements in the accuracy of solutions produced by large language models. Despite these advancements, most current tool-based methods simply combine text and code (Gou et al., 2023; Wang et al., 2023a) without fully leveraging the model's ability to understand and utilize external tools. Our work aims to address this gap by generating training data from scratch based on tool information to foster the model's tool-awareness and help it learn how to effectively use tools.

Tool-Augmented Language Models Due to their reliance on static, parametric knowledge, LLMs often struggle with complex computations. Enhancing LLMs with tools can significantly alleviate these limitations and improve their reasoning and generation performance (Wang et al., 2024; Yuan et al., 2024; Chen et al., 2024; Patil et al., 2023). Many current studies explore the potential of LLMs in tool usage: for example, ToolLlama (Qin et al., 2023) generates data that stimulates the model's ability to use real-world APIs, while ToolAlpaca (Tang et al., 2023) creates tool usage data through a model-based toolkit and multi-agent interactions. While existing approaches offer some enhancement, they struggle to actively leverage advanced libraries and fail to improve the model's domain knowledge and reasoning. Our work aims to build on Python toolkits, fostering the model's tool awareness, usage and reasoning ability to use python tools for scientific problem solving.

3 Motivations

This section investigates the dual role of Python tool in scientific reasoning: its potential to enhance complex problems while risking unnecessary reliance in simpler scenarios. We analyze level-dependent tool necessity, evaluate library optimization strategies, and provide insights into strategic tool utilization based on task complexity.

Preliminary Study 1: Marginal Effect of Tool Usage We first analyze 500 problems (Levels 1-5) sampled from MATH(Hendrycks et al., 2021) to compare zero-shot CoT and tool-integrated prompts, assessing tool necessity in complex reasoning. Given the base model's limitations in code generation, we apply multiple sampling to reduce errors and evaluate performance using pass@10 accuracy, considering a correct result within the top 10 samples as success.

As shown in Figure 2 (left), tool-assisted performance surpasses CoT as problem difficulty increases, showing diminishing returns on simpler tasks but significant gains on complex ones. However, tool usage incurs higher computational costs, with code generation and execution taking 3199.4s vs. CoT's 1996.7s. These findings highlight the marginal effect of tool usage: for simpler tasks, tools offer little benefit and may introduce errors, while for complex tasks, they enable symbolic computations beyond CoT's capabilities, justifying the additional reasoning cost.

Preliminary Study 2: Optimization of Tool Usage This study explores optimizing tool usage in complex problems by employing In-Context Learn-

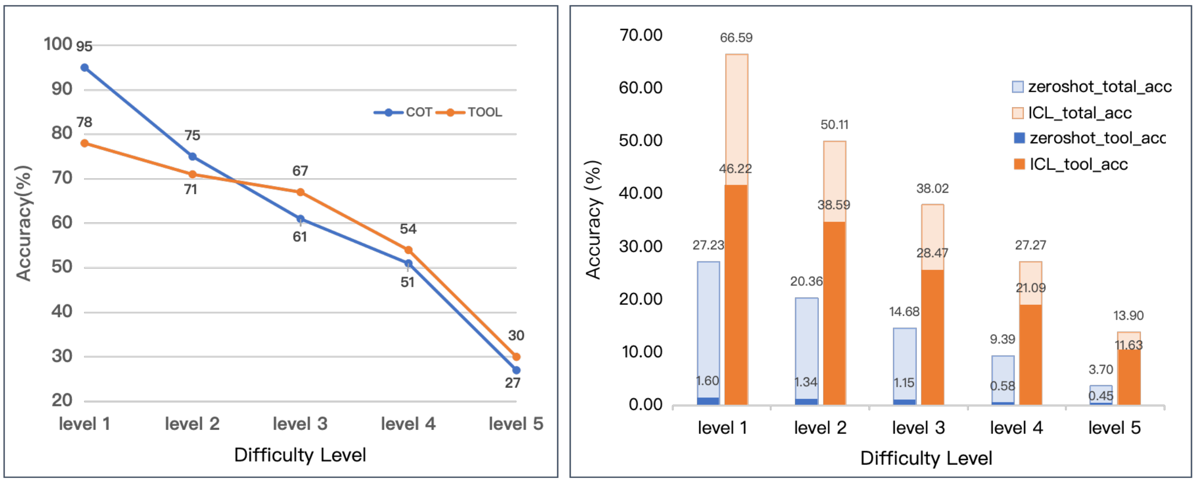


Figure 2: This figure compares **Pass@10 accuracy** across difficulty levels (left), highlighting the advantages of CoT in simpler problems and tool usage in more complex ones. The right side compares **Pass@1 accuracy** between Zeroshot and ICL settings, emphasizing tool-driven improvements and overall performance across difficulty levels.

ing (ICL) with explicit SymPy guidance, leveraging tool-integrated few-shot prompts from ToRA (Gou et al., 2023) (see Table 5). We evaluate performance on the MATH dataset using pass@1 accuracy, where correctness is determined by the first sampled output.

As shown in Figure 2 (right), tool-assisted accuracy improves with task difficulty, demonstrating the effectiveness of Python libraries in complex reasoning. Additionally, the proportion of correct tool-usage samples increases, indicating improved model proficiency in Python-based computations. Explicit SymPy guidance significantly enhances performance in symbolic computation, highlighting its role in tackling complex mathematical reasoning tasks.

Conclusion and Insights Based on the above findings, we conclude that effective tool usage in scientific problem-solving requires both **selective application** and **optimization of Python libraries**. The model must determine when to use tools, balancing the risk of errors in simpler tasks with the potential for enhanced performance in more complex ones. Additionally, providing explicit guidance on advanced Python libraries, such as SymPy, can significantly improve performance on challenging tasks, making the model a more effective assistant in scientific domains. These insights underscore the importance of strategically leveraging tools and optimizing library usage, laying a foundation for the development of adaptive reasoning systems capable of effectively utilizing Python tools across tasks of varying complexity.

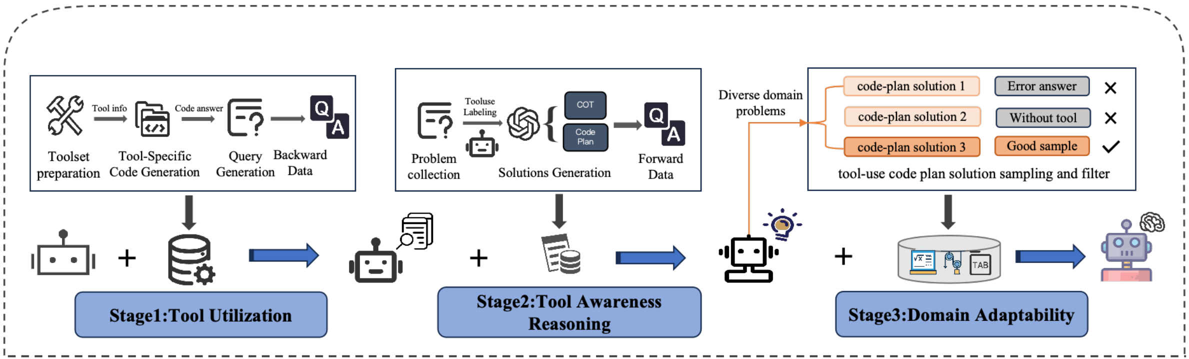
Inspiration Inspired by human learning, we deconstruct tool usage into three progressive stages: knowledge acquisition, where individuals consult documentation and example code to understand a tool's functionality; strategic awareness, where they decide when to use the tool based on context; and adaptive expertise, where repeated practice deepens their mastery. Our goal is to instill a similar learning process in LLMs, fostering their abilities in Utilization, Awareness, and Generalization of tools, making them reliable and accurate assistants in the scientific domain.

4 TAT Framework

Overview We introduce the TAT framework, a three-stage paradigm to enhance an LLM's autonomous tool usage in multi-step reasoning tasks: Utilization (learning tool usage through examples), Awareness (developing decision-making for tool application), and Generalization (improving adaptability across domains). Together, these stages enable effective tool application, enhancing performance in complex scientific tasks.

4.1 Fine-Grained Multi-Step Reasoning Paradigm

We propose a fine-grained multi-step reasoning paradigm that integrates natural language multi-step reasoning with executable code. This approach decomposes reasoning into explicit sub-tasks, each paired with code generation. An illustration of solution paradigm is provided in Figure 3.



Problem: To a good approximation, the microwave spectrum of $H^{35}Cl$ consists of a series of equally spaced lines, separated by 6.26×10^{11} Hz. Calculate the bond length of $H^{35}Cl$.

Code Plan:

Step 1: Import `numpy` for mathematical operations, `scipy.constants` for physical constants and define constants.

```python
import numpy as np
import scipy.constants as const
h = const.h  # Planck's constant (J·s)
Hz = 6.26e11  # Line spacing in Hz (given)
pi = np.pi  # Pi constant
```

Step 2: Calculate the Moment of Inertia (I) using the given relationship (2B = frac{h}{4 pi^2 I}).

```python
I = h / (4 * pi**2 * Hz)
```

Step 3: Calculate the reduced mass (μ) for HCl using the formula (mu = frac{m_H cdot m_Cl}{m_H + m_Cl}). We will define the masses of hydrogen and chlorine in kilograms and then compute the reduced mass.

```python
m_H = 1.661e-27  # mass of hydrogen in kg
m_Cl = 35.0 * 1.661e-27  # mass of chlorine in kg
mu = (m_H * m_Cl) / (m_H + m_Cl)  # reduced mass in kg
```

Step 4: Calculate the bond length (l) in meters using the formula (l = sqrt{frac{I}{mu}}).

```python
l = np.sqrt(I / mu)
```

Step 5: Convert bond length from meters to picometers (1 m = 1e12 pm) and print the result.

```python
bond_length_pm = round(l * 1e12) # bond length in picometers
print(f'The answer is: {bond_length_pm }')
```

Extract python code and execute:

```
>>>   The answer is 129
```

Figure 3: The workflow of our method consists of three stages: (1) Using tool information to generate backward data for establishing tool knowledge. (2) Prompting the LLM to generate tool awareness solution for real-world problems, building reasoning capabilities. (3) Conducting large-scale sampling of diverse, high-quality reasoning paths from domain-specific tasks for multi-task fine-tuning. An example of CodePlan is attached.

4.2 Bidirectional Data Generation

Our work focuses on systematically cultivating the model's tool-usage capabilities: backward generation obtains clear supervision signals through code executability, while forward generation introduces tool-usage labeling signals to facilitate tool-aware reasoning of models.

Backward Data Generation This stage establishes a tool knowledge base by generating supervised data that aligns tool usage with problem-solving. It consists of three key steps that collaboratively automate the creation of labeled data in an unsupervised manner. Implementation details are provided in Appendix A.1.1.

Toolset Collection and Processing: We start with collecting Python libraries through web crawling to extract detailed tool information, including functionality, parameters, and usage. Then we employ an LLM to optimize this information by assigning functional labels (e.g., Algebra, Geometry) and creating concise descriptions. This structured data forms the foundation for generating code snippets and queries in subsequent stages.

Tool-Specific Code Generation: We generate code cases based on the above toolset to illustrate the practical application of specific tools. Beginning with a handcrafted seed pool, we iteratively expand it using few-shot generation combined with filtering techniques to ensure diversity and reduce redundancy. This process produces a rich collection of executable code cases that effectively demonstrate each tool's usage.

Query and Solution Construction: To maximize the utility of the generated code cases, we sample and exclude those without numerical outputs to ensure relevance for downstream tasks. The sampled cases are executed to generate potential user queries, with the LLM providing a natural language explanation detailing the reasoning process behind tool usage. These explanations, along with the code, form structured code plan solutions paired with generated queries, creating a forward dataset for initial training.

Forward Data Generation This stage focuses on developing the model's tool awareness — the ability to autonomously determine whether and when to invoke tools during multi-step reasoning. To achieve this, we implement a three-phase strategy: automated problem labeling combining rule-based accuracy and LLM self-assessment to filter tool-dependent tasks, followed by reasoning path synthesis and verification. Key steps include:

Problem Collection and Labeling: We begin with collecting real-world problems and labeling them based on the necessity of tool usage. To distill the model's tooluse confidence, we use a dual approach for obtaining tool usage labels:

- **Rule-Based Labeling**: We compare CoT with tool-integrated reasoning using multiple sampling and assign a "tool needed" label if the tool-integrated approach achieves higher accuracy. However, both methods may fail on complex problems.

- **LLM-as-Judge Labeling**: In cases where rule-based metrics are insufficient, we leverage the model's self-confidence to evaluate tool necessity. A well-designed prompt (see Table 6) guides the model to reason through the problem, assess the need for tool usage, and provide a self-judged decision, including both a tooluse judgment and an explanation.

Solution Generation: We prompt an LLM to generate high-quality reasoning paths that incorporate tool awareness and problem-solving strategies, including both CoT and CodePlan solutions. While CoT paths are generated directly, CodePlan solutions are obtained by prompting the LLM to transform CoT-based reasoning into structured format.

Solution Verification and Filtering: The generated reasoning paths are verified and filtered. CodePlan outputs are executed to confirm correctness, while CoT solutions are compared to expected answers. Errors lead to rejection of the reasoning path, ensuring only accurate, verified paths are included in the training corpus.

4.3 Diverse and High-Quality Reasoning Paths for Multi-Task Fine-Tuning

After establishing tool utilization and awareness, we improve the model's problem-solving abilities by sampling real-world domain data to create task-specific fine-tuning datasets. We collect problems from open-source communities and use temperature sampling to generate diverse reasoning paths. These are evaluated for correctness and tool usage, as defined in Section 5.1. Synthesizing filtered data into a cross-domain training framework to cultivate generalization capabilities. Implementation details are provided in Appendix A.1.1.

4.4 Three-stage Training Framework

Building on the rich data from previous steps, we propose a three-stage training strategy to progressively enhance and optimize the model's capabilities. Each stage builds upon the previous one, ensuring sustained efficiency and effectiveness in solving complex tasks.

To achieve this, the training objective at each stage is to minimize the cross-entropy loss over the training data. Let $\mathcal{D} = \{(x_i, y_i)\}$ denote a given QA dataset, where x_i represents an input sequence and $y_i = (y_{i,1}, y_{i,2}, \ldots, y_{i,T})$ is the corresponding target output sequence. The model parameters are denoted by θ. The training objective at each stage can be written as:

$$\mathcal{L}(\theta) = -\sum_{(x,y)\in\mathcal{D}} \sum_{t=1}^{T} \log P_\theta(y_t \mid x, y_{<t}) \tag{1}$$

Backward Data Training In the first stage, we leverage backward data training on the base model to instill tool-knowledge and enhance the model's ability to effectively use tools. Specifically, starting from initial parameters θ_0, we train the model on backward-generated data $\mathcal{D}_{\text{backward}}$:

$$\theta_1 = \arg\min_\theta \mathcal{L}_{\text{backward}}(\theta; \mathcal{D}_{\text{backward}}) \tag{2}$$

Forward Data Training The second stage involves training the model using tool awareness forward data. Here, we take the parameters θ_1 obtained from the first stage as initialization and train on a set of forward-generated data $\mathcal{D}_{\text{forward}}$:

$$\begin{aligned} \theta_2 &= \arg\min_\theta \mathcal{L}_{\text{forward}}(\theta; \mathcal{D}_{\text{forward}}) \\ &\text{with} \quad \theta \text{ initialized at } \theta_1. \end{aligned} \tag{3}$$

Multi-task Learning In the third stage, building on the model trained in the second stage, we apply rejection sampling and multi-task learning to further enhance adaptability. Let $\{\mathcal{D}_m\}_{m=1}^{M}$ represent multiple task datasets, each corresponding to a particular domain or problem type. We combine these datasets into a unified multi-task objective:

$$\begin{aligned} \mathcal{L}_{\text{multi-task}}(\theta) = \sum_{m=1}^{M} \alpha_m \Bigg(-\sum_{(x,y)\in\mathcal{D}_m} \sum_{t=1}^{T} \\ \log P_\theta(y_t \mid x, y_{<t}) \Bigg), \end{aligned} \tag{4}$$

where α_m are weighting factors for each task. With θ_2 as the initialization, the model is trained to minimize $\mathcal{L}_{\text{multi-task}}$, resulting in parameters θ_3:

$$\theta_3 = \arg\min_\theta \mathcal{L}_{\text{multi-task}}(\theta; \{\mathcal{D}_m\}_{m=1}^{M}) \tag{5}$$

5 Experiments

To evaluate the effectiveness of TAT, we conduct a series of experiments that assess the model's performance on a range of scientific reasoning tasks.

5.1 Experiment Settings

Evaluation Tasks We evaluate our method on three reasoning tasks: **mathematical**, **scientific**, and **tabular reasoning**. For mathematical reasoning, we use the MATH (Hendrycks et al., 2021) and GSM8K (Chen et al., 2022) datasets, which test capabilities ranging from basic to advanced competition-level problems. In scientific reasoning, we employ SciBench (Wang et al., 2023b), derived from college-level textbooks in chemistry, physics, and mathematics to assess scientific reasoning and computational skills. For tabular processing, we utilize the DocMath-Eval (Zhao et al., 2024) benchmark, focusing on subsets $\mathrm{DM}_{\mathrm{SimpShort}}$ and $\mathrm{DM}_{\mathrm{CompShort}}$, to evaluate numerical reasoning in financial documents while avoiding the complexity of long contexts or retrieval tasks. Further details are provided in Appendix A.2.1.

Baseline We compare TAT with three categories of baseline methods: **Closed-source LLMs**, which include OpenAI's GPT-4 (Achiam et al., 2023), Anthropic's Claude-2 (Anthropic, 2023), and Google's Gemini-1.5 (Team et al., 2024); **Open-source LLMs**, such as Mistral-7B-v0.3 (Nadhavajhala and Tong, 2024), GLM-4-8B (GLM et al., 2024), and LLama3-70B (AI, 2024); and **Tool-integrated learning methods**, including TORA (Gou et al., 2023), MAmmoTH2 (Yue et al., 2024), and MathCoder2 (Lu et al., 2024b). For general model baselines, we report performance with CoT prompting, primarily due to challenges in code generation. In contrast, tool-integrated learning methods incorporate reasoning mechanisms specifically tailored to tools, offering more effective solutions for tasks that require tools. Further details are provided in Appendix A.2.2.

Evaluation Metrics We assessed the model's performance using **Answer Accuracy**, **Code Executability Rate**, and a novel metric, **Tool Usage Score**. Answer Accuracy represents the model's overall problem-solving capability, while Code Executability Rate evaluates the success rate of code generation. The Tool Usage Score, detailed in the appendix A.2.3, is a composite metric that measures the model's proactive and effective use of tools in problem-solving. It combines three weighted sub-metrics: Tool Proactivity, Tool Utilization Success Rate, and Tool-Driven Accuracy.

5.2 Experiment Results

Answer Accuracy Table 1 compares answer accuracy across datasets. Our model performs competitively, excelling in Math (37.7%) and $\mathrm{DM}_{\mathrm{SimpShort}}$ (78%), matching or surpassing closed-source models like Claude2. It also outperforms open-source models across all datasets, demonstrating superior performance in various tasks. Furthermore, our model surpasses existing tool-integrated methods, achieving an impressive average accuracy of 53.1%. In more complex tasks like Math and SciBench, it outperforms most open-source baselines, highlighting its adaptability to challenging reasoning tasks. However, its lower accuracy on SciBench suggests scientific reasoning remains a challenge, likely requiring more domain-specific knowledge and advanced reasoning strategies.

Code Execution ratio Table 2 shows code executability across all datasets. High executability rates (90%+) are observed in simpler tasks like GSM8K, while complex tasks like Math yield lower rates (60%+), emphasizing the need for better Python tool integration. Our model outperforms baselines across all datasets, achieving the highest code executability rates by effectively handling task complexity. This highlights the importance of training models to understand and effectively utilize tools, especially in scenarios requiring advanced computational reasoning.

Tool Usage Score The Tool Usage Score in Table 2 reflects the extent to which each method utilizes tools in its reasoning process. Our method demonstrates selectively low tool usage scores in certain datasets (0.0 in $\mathrm{DM}_{\mathrm{SimpShort}}$ and $\mathrm{DM}_{\mathrm{CompShort}}$), while maintaining high tool usage in more complex datasets such as Math and SciBench, as shown in Figure 4. This suggests that our approach prioritizes tool efficiency, leveraging external tools only when necessary, rather than applying them indiscriminately. In contrast, TORA achieves consistently high tool usage scores (44.7 in $\mathrm{DM}_{\mathrm{SimpShort}}$, 30.6 in $\mathrm{DM}_{\mathrm{CompShort}}$) due to its reliance on SymPy-enhanced data. On the other hand, MAmmoTH2-8B and MathCoder2-8B exhibit significantly lower tool usage scores (e.g., 13.6 avg in MAmmoTH2-8B, 14.2 avg in MathCoder2-8B) because they lack

Models	GSM8K	Math	SciBench	$DM_{SimpShort}$	$DM_{CompShort}$	AVG
Closed-Source Models						
Claude2	85.2	32.5	13.9	74.5†	72.0†	55.6
GPT-4-Turbo	94.2	51.8	42.4	82.5	81.0	70.4
Gemini-1.5-Flash	86.2	54.9	46.6†	78.0	69.5	67.0
Open-Source Models						
Mistral-7B-v0.3	45.9	16.5	4.6	40.0	28.0	27.0
GLM-4-8B	84.0	30.4	7.9†	44.0	34.0	40.1
Llama3-70B	80.1	44.0	22.8†	73.5	63.5	56.8
Tool-integrated Learning Methods						
TORA-7B	68.8	40.1	4.1†	13.0†	10.0†	27.2
MAmmoTH2-8B	70.4	35.8	6.0†	3.0†	1.9†	23.4
MathCoder2-8B	69.9	38.4	2.6†	5.0†	3.5†	23.9
Ours	79.7	37.7	16.2	78.0	54.0	53.1

Table 1: Answer Accuracy across five evaluation sets for various models and tool-integrated methods. Results marked with † indicate re-implementations, while the remaining results are drawn directly from existing literature.

Models	GSM8K	Math	SciBench	$DM_{SimpShort}$	$DM_{CompShort}$	Average
TORA	99.5/21.9	84.4/52.9	72.9/45.8	59.0/44.7	78.0/30.6	78.8/39.2
MAmmoTH2-8B	91.6/0.0	65.9/31.3	76.4/36.6	65.5/0.0	52.5/0.0	70.4/13.6
MathCoder2-8B	98.3/0.0	64.1/35.7	85.7/35.2	92.0/0.0	94.5/0.0	86.9/14.2
Ours	99.2/29.9	87.8/48.1	89.1/32.6	100.0/0.0	100.0/0.0	95.2/22.1

Table 2: Code Executability Rate and Tool Usage Score across five evaluation sets for Vanilla tool learning method.

specific data augmentation strategies for tool-based reasoning. Overall, our model's ability to selectively and efficiently use tools highlights its awareness of when tool integration is beneficial, ultimately leading to more robust task performance across datasets.

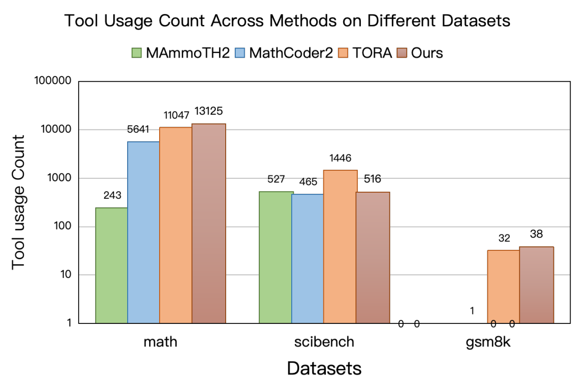


Figure 4: Comparison of tool usage on three datasets (Docmath is not shown due to its low tool frequency).

6 Further Analysis

6.1 Ablation of Multi-stage

Table 3 presents a multi-stage ablation study highlighting the contributions of each stage to model performance. Stage 1 (Tool Initialization) establishes tool usage capabilities, achieving high code executability (93.9% on GSM8K) but low task accuracy (41.1% on GSM8K), indicating reliance on execution without strong reasoning ability. Stage 2 (Reasoning Initialization and Tool Awareness) introduces two approaches: full tool use, which enhances execution and benefits complex tasks (35.6% on MATH), and awareness-based tool use, which selectively applies tools, improving simple task accuracy ($DM_{SimpShort}$: 81.9%) while reducing unnecessary tool reliance (GSM8K tool score: 77.1 $\rightarrow$ 29.2). This suggests that full tool use boosts execution, while awareness-based tool use enhances efficiency by integrating reasoning. Stage 3 (Multi-task Finetuning) integrates these improvements, achieving the highest overall accuracy (GSM8K: 79.7) with a well-balanced tool usage strategy. This structured training approach optimally combines

Models	GSM8K	Math	SciBench	$DM_{SimpShort}$	$DM_{CompShort}$	Average
Stage 1(tool initialization)	41.1/93.9/34.7	20.3/61.3/48.5	6.2/62.2/28.0	34.5/90.5/37.7	13.0/95.0/35.1	23.0/80.6/36.8
Stage 2(full tooluse)	74.7/99.2/77.1	35.6/84.5/56.4	17.1/90.2/34.9	66.0/99.5/37.4	22.0/99.5/0.0	43.1/94.6/41.2
Stage 2(awareness tooluse)	77.1/98.9/29.2	29.9/78.6/28.7	13.1/83.6/26.3	81.9/99.5/0.0	55.3/100.0/0.0	51.5/92.1/16.8
Stage 3(multi-task finetuning)	79.7/99.2/29.9	37.7/87.8/48.1	16.2/89.1/32.6	78.0/100.0/0.0	54.0/100.0/0.0	53.1/95.2/22.1

Table 3: Performance of the multi-stage training approach, including task accuracy, code executability rate, and tool usage score across different models and evaluation datasets.

Models	GSM8K	Math	SciBench	$DM_{SimpShort}$	$DM_{CompShort}$	Average
Tool-zeroshot	42.8/94.2/0.0	12.6/76.4/28.0	3.3/32.2/24.4	48.5/77.0/0.0	22.0/74.0/0.0	25.8/70.8/10.5
Tool-ICL	50.5/97.9/46.1	22.0/52.6/46.5	6.6/67.9/28.8	56.5/76.5/0.0	31.0/74.0/0.0	33.3/73.8/24.3
RAG	26.0/80.3/41.6	7.5/57.0/30.6	3.6/55.3/27.6	0.0/92.0/24.3	0.0/89.5/33.4	7.4/74.8/31.5
ours	79.7/99.2/29.9	37.7/87.8/48.1	16.2/89.1/32.6	78.0/100.0/0.0	54.0/100.0/0.0	53.1/95.2/22.1

Table 4: Performance of the different tool knowledge integration methods, including task accuracy, code executability rate, and tool usage score on evaluation datasets.

reasoning and tool utilization for robust performance across diverse tasks.

6.2 Comparison of Tool Knowledge Integration Methods

The comparison of few-shot ICL, RAG-based, and our fine-tuning approach highlights the advantage of integrating tool knowledge during training rather than relying on retrieval at inference. In our RAG implementation, the model retrieves relevant documentation to infer necessary functions, guiding tool usage dynamically. However, as shown in Table 4, RAG-based retrieval struggles with complex documentation, leading to accuracy degradation. This contrast emphasizes two key aspects of tool utilization: training-based approaches enable autonomous tool selection by directly associating queries with relevant tools, while inference-time retrieval aids generalization to unseen tool calls through contextual adaptation. By decoupling tool retrieval from execution, our fine-tuned model demonstrates superior adaptability in handling complex reasoning tasks, outperforming retrieval-based methods in both accuracy and efficiency.

6.3 Impact of Tool Usage Frequency

Our approach generates diverse reasoning paths via temperature sampling with an initialized model. We examine tool dependency across math, science, and table reasoning by analyzing 20 reasoning paths per domain, categorizing samples into five groups based on tool usage, and measuring task accuracy. As shown in Figure 5, math and table tasks peak at moderate tool usage, while SciBench shows a consistent positive correlation between tool use and accuracy, underscoring its reliance on tools. These findings highlight the need for calibrated tool dependency to optimize performance.

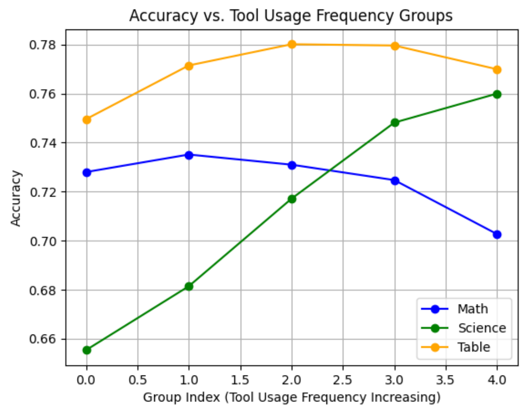


Figure 5: Analysis of task accuracy based on tool usage frequency across three domains.

7 Discussion and Conclusion

Our work highlights the importance of selective tool use and optimized Python libraries in enhancing LLMs' scientific reasoning. The TAT framework fosters tool awareness and refines reasoning process, enabling LLMs to tackle complex scientific problems effectively. Our experiments reveal tool usage patterns, demonstrating TAT's impact on tool application.

Future research will focus on expanding data scale and domain coverage, improving data quality, bridging distribution gaps, and refining sampling and filtering strategies to enhance the generalization and diversity of tool-assisted reasoning.

Limitations

Our method currently encounters several limitations:

(1) Tool Initialization stage Data Mismatch: The data used in the tool initialization stage differs from the data in the later stages, potentially affecting the effectiveness of training. We aim to optimize the reverse problem generator to better match real-world task distributions.

(2) Dependency on Strong Model Supervision in Second Stage: The second stage relies on strong model supervision signals, which can be costly. We plan to develop a more cost-effective solution by using open-source models to generate data for this stage.

(3) Suboptimal Tool Selection Preferences: Current tool selection relies on rule-based heuristics and confidence-based distillation, which have limitations. Exploring more effective modeling of tool selection preferences is a promising future direction.

(4) Limited Data Generation and Sampling: Data generation and sampling currently rely solely on the training dataset's problem samples. To enhance model performance and scalability, we plan to expand the sampling data sources to include a broader range of real-world scientific problems.

(5) Limited Domain Applicability: While the TAT framework enhances Python-based scientific reasoning, its scope is limited to mathematics, science, and table reasoning. Future work should extend its application to domains like coding, finance, and engineering, integrating diverse tools to improve adaptability and robustness.

Acknowledgements

We want to thank all the anonymous reviewers for their valuable comments. This work was supported by the National Science Foundation of China (NSFC No. 62206194), the Natural Science Foundation of Jiangsu Province, China (Grant No. BK20220488), and the Young Elite Scientists Sponsorship Program by CAST (2023QNRC001).

References

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

Meta AI. 2024. Llama 3 model card. `https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md`. GitHub Repository.

Anthropic. 2023. Claude 2. `https://claude.ai`. Version 2.0.

Daman Arora, Himanshu Gaurav Singh, et al. 2023. Have llms advanced enough? a challenging problem solving benchmark for large language models. *arXiv preprint arXiv:2305.15074*.

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

Zhi-Yuan Chen, Shiqi Shen, Guangyao Shen, Gong Zhi, Xu Chen, and Yankai Lin. 2024. Towards tool use alignment of large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 1382–1400.

Zhiyu Chen, Wenhu Chen, Charese Smiley, Sameena Shah, Iana Borova, Dylan Langdon, Reema Moussa, Matt Beane, Ting-Hao Huang, Bryan Routledge, et al. 2021. Finqa: A dataset of numerical reasoning over financial data. *arXiv preprint arXiv:2109.00122*.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359.

Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, Hao Yu, Hongning Wang, Jiadai Sun, Jiajie Zhang, Jiale Cheng, Jiayi Gui, Jie Tang, Jing Zhang, Juanzi Li, Lei Zhao, Lindong Wu, Lucen Zhong, Mingdao Liu, Minlie Huang, Peng Zhang, Qinkai Zheng, Rui Lu, Shuaiqi Duan, Shudan Zhang, Shulin Cao, Shuxun Yang, Weng Lam Tam, Wenyi Zhao, Xiao Liu, Xiao Xia, Xiaohan Zhang, Xiaotao Gu, Xin Lv, Xinghan Liu, Xinyi Liu, Xinyue Yang, Xixuan Song, Xunkai Zhang, Yifan An, Yifan Xu, Yilin Niu, Yuantao Yang, Yueyan Li, Yushi Bai, Yuxiao Dong, Zehan Qi, Zhaoyu Wang, Zhen Yang, Zhengxiao Du, Zhenyu Hou, and Zihan Wang. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *Preprint*, arXiv:2406.12793.

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

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

Varun Kumar, Leonard Gleyzer, Adar Kahana, Khemraj Shukla, and George Em Karniadakis. 2023. Mycrunchgpt: A chatgpt assisted framework for scientific machine learning. *arXiv preprint arXiv:2306.15551*.

Pan Lu, Swaroop Mishra, Tanglin Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. 2022. Learn to explain: Multimodal reasoning via thought chains for science question answering. *Advances in Neural Information Processing Systems*, 35:2507–2521.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2024a. Chameleon: Plug-and-play compositional reasoning with large language models. *Advances in Neural Information Processing Systems*, 36.

Zimu Lu, Aojun Zhou, Ke Wang, Houxing Ren, Weikang Shi, Junting Pan, Mingjie Zhan, and Hongsheng Li. 2024b. Mathcoder2: Better math reasoning from continued pretraining on model-translated mathematical code. *arXiv preprint arXiv:2410.08196*.

Pingchuan Ma, Tsun-Hsuan Wang, Minghao Guo, Zhiqing Sun, Joshua B Tenenbaum, Daniela Rus, Chuang Gan, and Wojciech Matusik. 2024. Llm and simulation as bilevel optimizers: A new paradigm to advance physical scientific discovery. *arXiv preprint arXiv:2405.09783*.

Sanjay Nadhavajhala and Yingbei Tong. 2024. Rubra-mistral-7b-instruct-v0.3.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2023. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE.

Nitin Liladhar Rane, Abhijeet Tawde, Saurabh P Choudhary, and Jayesh Rane. 2023. Contribution and performance of chatgpt and other large language models (llm) for scientific and research advancements: a double-edged sword. *International Research Journal of Modernization in Engineering Technology and Science*, 5(10):875–899.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. 2023. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Ke Wang, Houxing Ren, Aojun Zhou, Zimu Lu, Sichun Luo, Weikang Shi, Renrui Zhang, Linqi Song, Mingjie Zhan, and Hongsheng Li. 2023a. Mathcoder: Seamless code integration in llms for enhanced mathematical reasoning. *arXiv preprint arXiv:2310.03731*.

Xiaoxuan Wang, Ziniu Hu, Pan Lu, Yanqiao Zhu, Jieyu Zhang, Satyen Subramaniam, Arjun R Loomba, Shichang Zhang, Yizhou Sun, and Wei Wang. 2023b. Scibench: Evaluating college-level scientific problem-solving abilities of large language models. *arXiv preprint arXiv:2307.10635*.

Zhiruo Wang, Zhoujun Cheng, Hao Zhu, Daniel Fried, and Graham Neubig. 2024. What are tools anyway? a survey from the language model perspective. *arXiv preprint arXiv:2403.15452*.

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

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R Fung, Hao Peng, and Heng Ji. 2023. Craft: Customizing llms by creating and retrieving from specialized toolsets. *arXiv preprint arXiv:2309.17428*.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Ren Kan, Dongsheng Li, and Deqing Yang. 2024. Easytool: Enhancing llm-based agents with concise tool instruction. *arXiv preprint arXiv:2401.06201*.

Xiang Yue, Tuney Zheng, Ge Zhang, and Wenhu Chen. 2024. Mammoth2: Scaling instructions from the web. *arXiv preprint arXiv:2405.03548*.

Yilun Zhao, Yitao Long, Hongjun Liu, Ryo Kamoi, Linyong Nan, Lyuhao Chen, Yixin Liu, Xiangru Tang, Rui Zhang, and Arman Cohan. 2024. Docmath-eval: Evaluating math reasoning capabilities of llms in understanding long and specialized documents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16103–16120.

Fengbin Zhu, Wenqiang Lei, Youcheng Huang, Chao Wang, Shuo Zhang, Jiancheng Lv, Fuli Feng, and Tat-Seng Chua. 2021. TAT-QA: A question answering benchmark on a hybrid of tabular and textual content in finance. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3277–3287, Online. Association for Computational Linguistics.

A Appendix

A.1 Training Details

A.1.1 Training Data Generation

Table 7 presents the detailed training data statistics and Table 8 and Table 9 presents the examples of training data.

Stage 1: Backward This stage focuses on tool selection and initialization, equipping the model with the ability to identify, understand, and explain the use of tools. The process is divided into the following steps:

Tool Information Collection: We begin by identifying the relevant Python toolkits for the target domain through an in-depth analysis of problem types and their Python-based solutions in existing datasets. To collect these tools:

- Web Scraping: Using web scraping techniques, we extract comprehensive information for each tool, including functionality descriptions, required parameters, and usage details. This ensures a broad and inclusive dataset covering all necessary tools.
- Toolset Repository Creation: The collected information is saved into a repository, forming the basis for subsequent processing and generation tasks.

Tool Information Filtering and Processing: To ensure the collected data is structured, concise, and directly applicable to reasoning tasks:

- Documentation Refinement: We use large language models (LLMs) to process the raw documentation. For each Python function or tool: 1) Labeling: Functional labels (e.g., "Algebra," "Geometry," "Number Theory") are assigned based on the tool's purpose and usage context.2) Description Simplification: Complex and verbose documentation is rewritten into clear and concise summaries. This step ensures easy interpretability and efficient use of the tool information.
- Irrelevant API Removal: Non-functional or unrelated APIs are filtered out, resulting in a clean, task-specific tool library.

Code Case Generation: To demonstrate the practical application of the tools, we generate example code snippets that illustrate their usage. The output of this step is a large collection of diverse, executable code snippets that effectively showcase each tool's functionality

- Few-Shot Seed Pool: A small pool of handcrafted examples is initially created to seed the generation process.
- Iterative Expansion: Using LLMs in a few-shot setup, we iteratively expand this pool by generating additional code snippets. Similarity-based filtering ensures diversity and avoids redundancy during each iteration.
- In-Context Learning (ICL): Randomly selected examples from the seed pool are used as in-context examples to enhance the generation quality in subsequent iterations.

Query and Code Plan Generation: To integrate the generated code snippets into a reasoning framework:

- Single-function Sampling: We focus on single-function usage examples and exclude those with non-numerical outputs to maintain relevance for reasoning tasks.
- Query Generation: Using LLMs(like Qwen2.5-7B-instruct model), the sampled functions are executed, and their results are analyzed to generate potential user queries. These queries simulate real-world problem statements that can be addressed using the functions.
- Natural Language Explanations: For each sampled case, the LLM generates a natural language explanation detailing: 1) The reasoning process behind the tool's usage. 2) How the tool addresses the problem and produces the observed output.
- Code Plan Creation: The explanations are combined with the code execution paths to form structured code plan samples, creating cohesive datasets that link tool outputs with problem-solving contexts.

Automated QA Pair Creation: The generated code snippets, explanations, and structured code plans are then leveraged to create high-quality scientific QA pairs:

- Code Validation: Each generated sample is validated to ensure correctness and compatibility across diverse tools, minimizing noise in the dataset.
- Scalability Across Domains: The backward data generation process is designed to scale effectively across multiple domains by leveraging tool diversity and automated processes.

This backward data generation approach automates the creation of labeled data with minimal human intervention, ensuring high-quality and consistent training datasets while reducing dependency on manual annotation.

Stage 2: Forward The second stage focuses on generating tool-aware forward data by addressing real-world problems in the target domain. This process leverages large language models (LLMs) to produce high-quality solutions through in-context learning, facilitating reasoning aligned with domain-specific challenges. The implementation details are outlined below:

- **Problem Collection and Preparation:** We utilize datasets such as MATH (Hendrycks et al., 2021) and GSM8K (Chen et al., 2022), which include problems paired with their corresponding CoT solutions for **mathematical reasoning**. Relevant problem sets are collected from FinQA (Chen et al., 2021) and TAT-QA (Zhu et al., 2021) in **tabular reasoning**. In **scientific reasoning** domains with limited large-scale datasets, such as scientific problem solving, we employ SciBench (Wang et al., 2023b) few-shot examples as seed data to prompt GPT-4omini to generate new scientific problems. Solutions are generated using three reasoning strategies: CoT, PoT, and CodePlan. To ensure robustness, a majority voting mechanism is applied, where the most frequent result across methods is selected as the final solution.
- **Tooluse Labeling:** Each problem is systematically labeled based on its dependency on tools, ensuring precise guidance for the model's decision-making in tool invocation.
  - **Rule-Based Labeling:** We compare the performance of CoT and tool-integrated reasoning on multiple sampling solutions using LLaMA-3-8B Instruction model. If tool-enhanced reasoning achieves a higher accuracy rate—measured by solution correctness—the problem is labeled as "tool needed."
  - **LLM-as-Judge Labeling:** We design a structured prompt (Table 6) that instructs LLaMA-3-8B to analyze each problem and determine the necessity of tool usage, providing a natural language explanation to justify its decision.
- **Solution Generation via In-Context Learning:** Using GPT-4omini, we generate step-by-step solutions for CoT reasoning. For problems requiring tool assistance, few-shot examples are used to prompt LLM to form structured code plans: 1) CoT Decomposition: For mathematical datasets (Math/GSM8K), we design prompts that guide the model to break down the standard CoT solution into fine-grained steps and generate corresponding code snippets for each step. 2) Structured Output:The generated reasoning paths are combined with executable code snippets to create cohesive code plan datasets. This approach emphasizes precision by integrating logical reasoning with tool execution.
- **Code Execution and Validation:** The CoT solution is directly extracted, and all generated code is executed to ensure correctness. Only samples with verified outputs are retained, filtering out flawed reasoning paths to ensure the dataset comprises high-quality problem-solving examples.

Stage 3: Multi-Task Fine-Tuning In this stage, we can theoretically access a vast array of open-source problems from the internet to enhance dataset diversity and expand training volume, ultimately improving model performance. However, to optimize computational efficiency and time, we primarily reuse the domain-specific datasets collected in Stage 2, leveraging rejection sampling to filter and enhance the solution set. Specifically, we use the training sets from MATH (Hendrycks et al., 2021) and GSM8K (Chen et al., 2022) for mathematical reasoning, incorporate LLM-generated scientific problems as described in Stage 2, and utilize training data from FinQA (Chen et al., 2021) and TAT-QA (Zhu et al., 2021) for tabular reasoning.

To generate a diverse set of candidate reasoning paths, we employ temperature sampling with a temperature of 0.9 and top_p of 0.8, setting the maximum output length to 2048 tokens. Temperature sampling introduces controlled randomness, encouraging the generation of varied reasoning trajectories while maintaining response quality. Correct responses are sampled, with a focus on challenging queries. Specifically, we apply our Stage 2 model to all collected problems, generating an average of 20 samples per question. Each sampled response undergoes execution and evaluation based on both correctness and tool utilization. Finally, only verified correct responses are retained after filtering, ensuring a high-quality dataset for fine-tuning.

A.1.2 Training Settings

We utilize Meta-Llama-3-8B-Instruct (AI, 2024) as the backbone to train our model. We employ alignment framework[1], DeepSpeed (Rasley et al., 2020) library, Zero Redundancy Optimizer (ZeRO) (Rajbhandari et al., 2020) Stage 3, FlashAttention (Dao et al., 2022), and the bfloat16 (BF16) and tfloat32 (TF32) mix computation precision on 8 NVIDIA A100 GPUs.

The training process is divided into three stages, with each stage configured to address specific tasks and optimize performance:(1) Stage 1 - Tool Initialization: In this stage, we use a dataset of 255K samples to initialize the tools. The model trains for 1 epoch, with a batch size of 3 and a gradient accumulation step of 8. The initial learning rate is set to 2e-5 and the warm-up ratio is 0.03 to facilitate stable convergence during early training.(2) Stage 2 - Reasoning Initialization: In this stage, the model undergoes 4 epochs of training using a dataset of 30K samples for inference initialization. Similar to Stage 1, the batch size is 3, with gradient accumulation steps of 8 and a starting learning rate of 7e-6. The warm-up ratio remains at 0.03.(3) Stage 3 - Sampling stage: In the final stage, the model is trained on a dataset of 300K samples. To mitigate catastrophic forgetting, the learning rate is reduced to 5e-6, and the model trains for 1 epoch. Gradient accumulation steps are again set to 8, with a batch size of 3.

A.1.3 Three-Stage Training Instruction Design

To prevent catastrophic forgetting in multi-task learning, we designed a three-stage training instruction framework that ensures task continuity. The first two stages serve as preparatory tasks, focusing on tool selection and initialization (Stage 1) and integrating tool usage through multi-step reasoning (Stage 2). These stages establish the core knowledge of the model. The third stage emphasizes multi-task learning, with instructions tailored to specific tasks but still linked to the first two stages, ensuring the model effectively applies previously learned knowledge. This design guarantees that the model retains its core capabilities while adapting to new tasks, promoting consistency and effective knowledge transfer between tasks. The design of the instructions for the different task stages is shown in the table 10

[1]`https://github.com/huggingface/alignment-handbook`

A.2 Evaluation Details

A.2.1 Evaluation Datasets

Table 11 summarizes the statistical details and examples of the five datasets used in the evaluation experiments, illustrating the size of the evaluation data and the domains involved in the tasks. These datasets cover a range of difficulty levels (from high school to college-level scientific problems) and diverse fields (including mathematics, physics, chemistry, and finance). Specifically, these datasets ensure a thorough assessment of the model's performance across various reasoning scenarios and highlight its generalization capabilities in diverse scientific tasks.

Dataset	#Samples	Fields
Math	5000	Pre-Algebra, Inter-Algebra, Algebra, Probability, Number Theory, Calculus, Geometry
GSM8K	1319	Pre-Algebra
SciBench	580	Chemistry, Physics, Mathematics
DM$_{\text{SimpShort}}$	200	Financial
DM$_{\text{CompShort}}$	200	Financial

Table 11: Statistics of the datasets used in the evaluation experiments, including test question counts, scenarios, and covered fields.

A.2.2 Evaluation Baselines

We select powerful closed-source models such as GPT-4 (Achiam et al., 2023), Claude-2 (Anthropic, 2023), and Gemini-1.5 (Team et al., 2024) as baselines to represent the current state-of-the-art model capabilities. In addition, we choose Mistral-7B (Nadhavajhala and Tong, 2024), GLM-4 (GLM et al., 2024), and LLaMA3-70B (AI, 2024) as open-source model baselines to reflect the general-

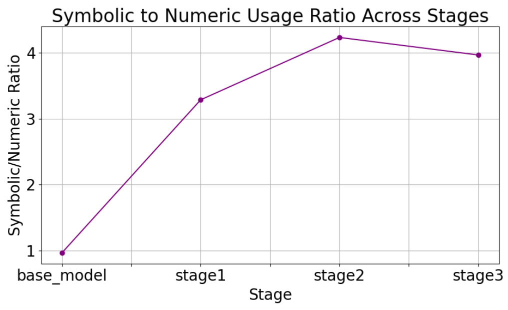


Figure 6: Shift towards symbolic computation: increased usage of symbolic functions as the model progresses in mathematical reasoning task.

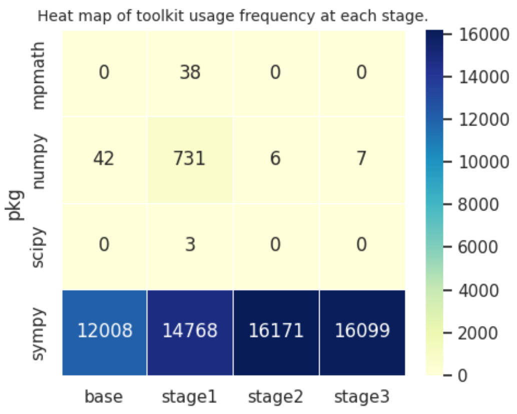


Figure 7: Visualization of the tool usage frequency at each stage on math dataset using a heatmap.

purpose reasoning ability of LLMs within the open-source community. Given the inherent challenges that general models face when using tools, particularly in Python code generation, we employ CoT prompting to generate solutions.

In addition to general-purpose models, we also evaluate several tool-integrated learning methods as comparison baselines to our approach which including:

- ToRA (Gou et al., 2023): By combining natural language reasoning and external computational tools, this approach uses imitation learning to improve the performance of LLMs on mathematical problems.
- MAmmoTH2 (Yue et al., 2024): Based on web-scraped corpora, this model builds instruction datasets to enhance the model's scientific reasoning ability.
- MathCoder2 (Lu et al., 2024b): A technique that employs high-quality mathematical pretraining datasets to generate tool-based reasoning data, enhancing mathematical problem-solving.

While these tool-integrated approaches focus on generating high-quality reasoning data to improve the model's reasoning ability, they do not explicitly address tool awareness or the optimized utilization of Python libraries in solving complex problems, which is a core focus of our work.

A.2.3 Evaluation Metric

Tool Usage Score The Tool Usage Score is a composite metric designed to assess the proactivity and effectiveness of a model's tool usage in problem-solving scenarios. It is derived from three weighted sub-metrics: (1)Tool Proactivity: The frequency at which the model actively chooses to utilize tools for solving problems, expressed as the proportion of all tasks (N) where tools were used (NT):Proactivity= $\frac{NT}{N}$(2) Tool Utilization Success Rate: This evaluates the correctness and efficiency of tool use, calculated as the ratio of successful tool executions (STE) to total tool invocation attempts (TEA).Utilization= $\frac{STE}{TEA}$(3) This metric measures the proportion of correct answers achieved with the assistance of tools, combining the model's reliance on tools and their effectiveness in solving tasks: ToolDriven Acc= $\frac{CAT}{N} * \frac{CAT}{CA}$, where CAT is the number of correct answers achieved using tools, CA is the total number of correct answers, and N is the total number of task.

To facilitate comparisons, the Tool Usage Score is calculated as a weighted average of these three sub-metrics:Tool Usage Score = $w_1 \cdot$ Proactivity $+$ $w_2 \cdot$Utilization$+ w_3 \cdot$ToolDriven Acc,,we assigned $w_1 = 0.3, w_2 = 0.3, w_3 = 0.4$ for computation in our experiments.

A.2.4 Additional Tool Analysis

This section provides a detailed analysis of tool usage across different training stages. As indicated in Figure 7, Sympy is the primary tool, with increasing usage highlighting its importance for complex tasks. This correlates with improved performance in GSM8K and Math tasks (Table 1), where symbolic computation is key. In contrast, mpmath and scipy are minimally used, suggesting tasks don't heavily rely on precise numerical computations. Numpy sees a notable spike in stage 1, likely due to tasks requiring efficient array operations.

By analyzing the distribution of symbolic (e.g., solve, simplify) and numerical (e.g., sqrt, cos, tan) computation functions, we observe a shift in the model's computational focus, indicating a transition toward symbolic capabilities for addressing advanced mathematical problems.

By comparing the distribution and proportion of symbolic computation functions (e.g., solve, simplify in sympy) with numerical computation functions (e.g., sqrt, cos, tan in numpy or math), we observe a transition in the model's computational capabilities. The proportion curves of function calls across different stages, shown in Figure 6 demonstrate a clear shift from numerical to symbolic computation. This trend indicates the model's growing ability to handle complex mathematical expressions and equations, suggesting a gradual shift towards symbolic capabilities for advanced mathematical challenges.

Prompt for Tool-Integrated Zeroshot

Integrate step-by-step reasoning and Python code to solve math problems using the following guidelines:
- Analyze the question and write functions to solve the problem; the function should not take any arguments.
- The format of the solution should be the following:

Begin with question analysis: step-by-step reasoning here
Let's write Python code to solve:

~~~
```python
def solution():
    # Define variables name and value based on the given context
    ...
    # Do math calculation to get the result
    ...
    # return result
    return result
result = solution()
print(result)
```
~~~

Prompt for Tool-Integrated ICL

Integrate step-by-step reasoning and Python code to solve math problems using the following guidelines:
- Analyze the question and write functions to solve the problem; the function should not take any arguments.
- Present the final result in LaTeX using a '`\boxed{}`'without any units.
- Utilize the '`pi`' symbol and '`Rational`' from Sympy for π and fractions, and simplify all fractions and square roots without converting them to decimal values.

Here are some examples you may refer to:

`Question:` Convert the point $(0, -3\sqrt{3}, 3)$ in rectangular coordinates to spherical coordinates. Enter your answer in the form (ρ, θ, ϕ), where $\rho > 0$, $0 \leq \theta < 2\pi$, and $0 \leq \phi \leq \pi$.

`Solution:`

`To convert the point` (x, y, z) `in rectangular coordinates to spherical coordinates` (ρ, θ, ϕ), `we use the following formulas:` rho = sqrt(x**2 + y**2 + z**2) theta = atan2(y, x) phi = acos(z/rho) Let's write a function:

~~~
```python
from sympy import sqrt, atan2, acos, pi
def rectangular_to_spherical():
    x, y, z = 0, -3*sqrt(3), 3
    rho = sqrt(x**2 + y**2 + z**2)
    theta = atan2(y, x)
    phi = acos(z/rho)
    return rho, theta, phi
spherical_coordinates = rectangular_to_spherical()
print(spherical_coordinates)
```output
(6, -pi/2, pi/3)
~~~
```

The required form of the answer is $(\rho, \theta, \phi)$, where $\rho > 0$, $0 \leq \theta < 2\pi$, and $0 \leq \phi \leq \pi$. Since our initial output has $\theta = -\pi/2$, which doesn't meet the condition for $\theta$, we add $2\pi$ to adjust its value.

Hence, the spherical coordinates for the point $(0, -3\sqrt{3}, 3)$ are $\boxed{(6, \frac{3\pi}{2}, \frac{\pi}{3})}$.

(Other examples omitted...)

Table 5: Prompt for Tool-Integrated Zeroshot and ICL settings
```

Prompt for Tooluse Labeling
You are a scientific expert and required to assess whether the given problem requires tool assistance based on golden solution. Please analyze the problem and its provided solution based on the following criteria: 1. High-complexity Calculations or Large-scale Data Operations: If manual derivation could lead to errors or would take excessive time, tool assistance may be needed. 2. Advanced Functions/Algorithms: If the solution involves complex functions (e.g., Algebra, Number Theory, Calculus) or advanced algorithms (e.g., Geometry, Probability, Graph Theory), tools can improve efficiency. 3. Visualization: If the solution benefits from visualizations for result verification or analysis, tool assistance (e.g., graphing, plotting) may be helpful. Please analyze the given problem solution, determine if tool assistance (including calculators or code) is required, and provide a brief explanation. Finally, respond in the following format: "Answer: Yes/No Explanation: <Your Explanation>." Problem: Solution:

Table 6: Prompt for Tooluse Labeling

Dataset	Samples	Fields	Problem Source
Stage 1	255k	sympy;scipy;numpy;mpmath	Self-generated
Stage 2	30k	reasoning	MATH/GSM8K/TATQA train and generated scientific problems
Stage 3	300k	Chemistry; Physics; Mathematics;Table	MATH/GSM8K/TATQA train and generated scientific problems

Table 7: Statistics of the generated train datasets.

Stage 1: Backward Data Example for Tool Initial

Question: Calculate the definite integral of the function $f(x) = x^2$ from x = 0 to x = 1.
CodePlan Solution:
### Step 1: Define the variable x and the function $f(x) = x^2$ using SymPy. Compute the definite integral of the function from x = 0 to x = 1 using the integrate function. Print the result to obtain the solution.

~~~
```python
from sympy import integrate, symbols
# Define the variable and function
x = symbols(\'x\')
expr = x**2
# Perform definite integration from 0 to 1
result = integrate(expr, (x, 0, 1))
print("The answer is: {}".format(result))
```
~~~

Stage 2: Forward data Example for multi-step reasoning Initial

Question:what was the mathematical range for all four groups in 12/2010 , assuming investments of $100 initially in 2008?
CodePlan Solution:
### Step 1: Extract the values for each group at the end of the specified period (12/2010).We need to extract the cumulative total return values for Valero common stock, S&P 500, old peer group, and new peer group from the table for the date 12/2010.

~~~
```python
# Values for each group at 12/2010
valero_2010 = 35.01
sp500_2010 = 91.67
old_peer_2010 = 88.41
new_peer_2010 = 72.84
~~~

### Step 2: Calculate the range for each group.The range is calculated as the difference between the maximum and minimum values of the four groups at the specified date (12/2010). We will use NumPy to facilitate this calculation.

~~~
```python
import numpy as np
# Create an array of the values
values = np.array([valero_2010, sp500_2010, old_peer_2010, new_peer_2010])
# Calculate the range\nrange_value = np.max(values) - np.min(values)
```
~~~

### Step 3: Print the calculated range value.

~~~
```python
# Print the result
print(f'The answer is: {range_value}')
```
~~~

Table 8: Examples of the generated train datasets (Part 1).

Stage 3: Task specific data Example for multi-task finetuning

Question: A hot metal cube of mass (m = 1.5kg) is dropped into a calorimeter containing (0.2kg) of water at an initial temperature of (20°C). The cube has an initial temperature of (100°C). Assuming no heat is lost to the surroundings, calculate the final equilibrium temperature of the system. The specific heat capacity of water is (c_w=4186 J/(kg·°C)) and the specific heat capacity of the metal is (c_m= 500 J/(kg·°C)). The unit of the answer is °C.

CodePlan Solution:

### Step 1: Import necessary libraries and Define the constants and initial conditions

~~~~
```python
import numpy as np
m = 1.5  # mass of the metal in kg
m_w = 0.2  # mass of the water in kg
c_w = 4186  # specific heat capacity of water in J/(kg·°C)
c_m = 500  # specific heat capacity of metal in J/(kg·°C)
# Initial temperatures
T_initial_m = 100  # initial temperature of the metal in °C
T_initial_w = 20  # initial temperature of the water in °C
```
~~~~

### Step 2: Set up the equation for heat transfer

~~~~
```python
m * c_m * (T_initial_m - T_f) = m_w * c_w * (T_f - T_initial_w)
```
~~~~

### Step 3: Rearrange the equation to solve for T_f

~~~~
```python
from sympy import symbols, Eq, solve
T_f = symbols('T_f')
heat_equation = Eq(m * c_m * (T_initial_m - T_f),
m_w * c_w * (T_f - T_initial_w))
T_final = solve(heat_equation, T_f)[0]
```
~~~~

### Step 5: Output the result

~~~~
```python
result = round(T_final, 1)
print(f'The answer is: {result} unit(°C)')\n
```
~~~~

Table 9: Examples of the generated train datasets (Part 2).

Stage	**Instruction**
Stage 1	You are a Python programming expert skilled in solving problems with appropriate libraries and functions. For the given task, identify and apply the most suitable Python tools, providing a clear explanation of the reasoning and implementation steps.
Stage 2	You are a reasoning expert with strong programming skills, specializing in step-by-step problem-solving. You excel at integrating natural language reasoning with Python code and using python libraries for optimized calculations. Your goal is to generate a step-by-step code plan that combines reasoning with Python code implementations, each wrapped in triple backticks, to deliver a precise and verifiable solution.
Math Task for Stage 3	You are an expert in Mathematics and Python programming, with deep expertise in algebra, calculus, geometry, number theory, and probability. You excel at leveraging Python libraries and integrating step-by-step mathematical reasoning with Python code for efficient and precise calculations. Your task is to generate a comprehensive and accurate code plan for the given mathematical problem, combining step-by-step reasoning with Python code implementations for each step.
Scientific Task for Stage 3	You are an expert in scientific reasoning and Python programming, with advanced knowledge in Physics, Chemistry, and Mathematics. You excel at leveraging Python libraries and integrating step-by-step scientific reasoning with Python code for complex computations. Your task is to generate a comprehensive and accurate code plan for the given scientific problem, uniting step-by-step reasoning with Python code implementations for each step.
Table Task for Stage 3	You are an expert in financial analysis and Python programming, skilled in interpreting finance-specific documents involving text and tables. You excel at leveraging Python libraries and integrating step-by-step table reasoning with Python code for accurate numerical computations. Your task is to generate a comprehensive and accurate code plan for the given table reasoning problem, blending step-by-step reasoning with Python code implementations for each step.

Table 10: Instructions for multi-Stage Training. The first two stages of initialization tasks contain only one instruction, while the third stage of multi-task fine-tuning designs domain-specific instructions for different domain data.