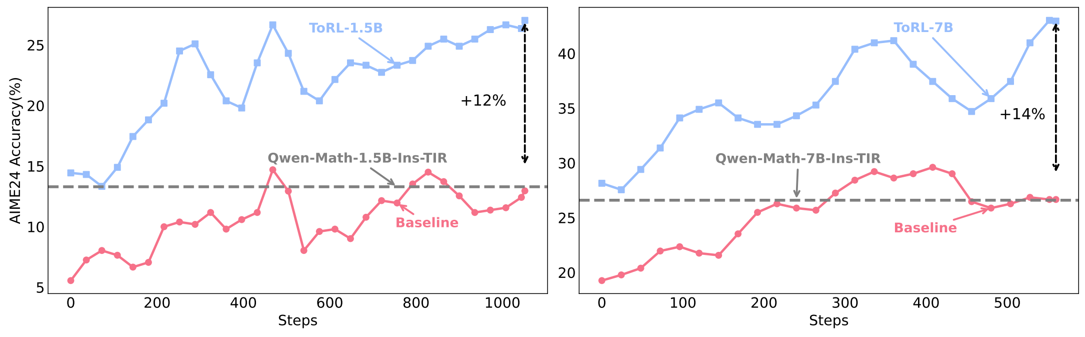
# TORL: Scaling Tool-Integrated RL

**Xuefeng Li**[*] **Haoyang Zou**[*] **Pengfei Liu**[†]

SJTU, SII, GAIR

## Abstract

We introduce **TORL** (Tool-Integrated Reinforcement Learning), a framework designed to train large language models (LLMs) to autonomously utilize computational tools through direct reinforcement learning (RL) scaling from base models. Unlike approaches based on supervised fine-tuning, TORL enables models to discover optimal strategies for tool utilization via unrestricted exploration. Experiments with Qwen2.5-Math base models demonstrate substantial improvements, with TORL-7B achieving **43.3%** accuracy on AIME24—surpassing RL without tool integration by **14%** and the best existing Tool-Integrated Reasoning (TIR) model by **17%**. Further analysis reveals several emergent behaviors—such as strategic tool invocation, self-regulation of ineffective code generation, and dynamic adaptation between computational and analytical reasoning. These capabilities **emerge without explicit instruction**, solely through reward-driven learning. We open-source our implementation, datasets, and models at https://github.com/GAIR-NLP/ToRL.



Question: Two parabolas are the graphs of the equations $y = 2x^2$ . . . Please give the value of k + m.

Response: To solve this problem, we need to find the points of intersection between . . .

~~~
```python
k_m_sum = k + m; print(k_m_sum)
```
```output 16 ```
~~~

. . . $k + m = -1 + (-8) = -9$. . . . However, the output of the code is 16, . . . Let's recheck the code and the steps:

~~~
```python
...
# Find the corresponding y-coordinates
intersection_points = [(x_val, eq1.subs(x, x_val)) for x_val in intersection_x]...
print((intersection_points, k_m_sum))
```
```output ([(-2, 18), (8, 38)], 16) ```
~~~

Thus, the value of $k + m$ is $\boxed{16}$.

Figure 1: Top Figure: Performance comparison of TORL versus baseline models(16-step moving). Both plots show AIME24 Accuracy (%) against training steps across 1.5B and 7B models. In both cases, TORL (Ours) significantly outperforms the baseline without tool and Qwen-2.5-Math-Instruct-TIR, achieving up to 12%(1.5B) and 14%(7B) higher. Bottom Figure: Emergent cognitive behavior during training. TORL first cross-validates the tool's output with reasoning results. Upon detecting inconsistencies, it engages in reflection and further verification through tool.

[*] Co-first authors
[†] Corresponding author

## 1 Introduction

Recent advances in large language models (LLMs) have demonstrated remarkable reasoning capabilities through reinforcement learning (RL). Models such as o1, R1 and Kimi exhibit emergent abilities including backtracking, self-correction, and reflection when trained with large-scale RL (Team, 2024; Guo et al., 2025; Team et al., 2025). Several open-source efforts have shown similar promising results (Qin et al., 2024; Huang et al., 2024; Zeng et al., 2025; Luo et al., 2025).

Traditional reasoning in language models has relied on pure natural language approaches such as Chain-of-Thought (CoT) (Wei et al., 2022). While effective for many tasks, these methods falter when facing complex calculations, equation solving, or processes requiring precise computation (Gao et al., 2023; Chen et al., 2022). Despite research into code generation models, few approaches successfully bridge the gap between LLMs' reasoning capabilities and computational tools' execution power through reinforcement learning. Tool-Integrated Reasoning (TIR), explored prior to recent RL scaling breakthroughs, enables models to invoke external tools by writing code, executing it through interpreters, and iteratively generating reasoning informed by code outputs. Implementations like ToRA, MathCoder, and Qwen2.5-Math-Instruct-TIR demonstrate TIR's effectiveness in enhancing mathematical problem-solving (Gou et al., 2023; Wang et al., 2023; Shao et al., 2024; Liao et al., 2024; Yang et al., 2024).

Despite these advances, existing TIR approaches face critical limitations. Most studies distill trajectories from stronger models and perform Supervised Fine-Tuning (SFT), restricting models to predetermined tool usage patterns and limiting exploration of optimal strategies. While some work, like Qwen-Math, applies RL to SFT-trained models, limited implementation transparency obscures understanding of tool integration within RL frameworks. To address these challenges, we introduce ToRL (Tool-Integrated Reinforcement Learning), a framework that scales reinforcement learning directly from base models without the constraints of prior supervised fine-tuning. Unlike previous approaches that incrementally improve SFT-trained models, ToRL enables RL training from scratch, allowing models to discover optimal tool utilization strategies through extensive exploration. This scaling approach yields qualitatively different behaviors than methods that build upon predetermined patterns.

Our experiments with Qwen2.5-Math base models at both 1.5B and 7B scales demonstrate substantial performance improvements over both traditional RL without code interpreters and Qwen2.5-Math-Instruct in TIR environments (Yang et al., 2024). Notably, our 7B model achieves a 43.3% accuracy on AIME problems, comparable to some 32B models trained with reinforcement learning (Hu et al., 2025).

Through extensive analysis, we uncover several key insights about tool-integrated learning dynamics:

- **Code usage evolution:** As training progresses, the proportion of problems the model solves using code increases steadily, accompanied by a growing percentage of syntactically correct and executable code. This demonstrates the model's autonomous acquisition of effective tool utilization strategies.
- **Self-regulation of ineffective code:** Without explicit instruction, models learn to identify and reduce the generation of ineffective code patterns, suggesting an emerging form of metacognition about tool utility.
- **Tool call frequency trade-offs:** Increasing the maximum allowed tool calls per problem significantly enhances performance but introduces severe computational overhead, revealing a critical efficiency-effectiveness trade-off in tool-integrated learning.

These findings reveal how ToRL fosters diverse cognitive behaviors, including obtaining feedback from code execution, cross-checking between computational and analytical approaches, and adaptively selecting reasoning strategies based on problem characteristics. The emergence of these behaviors through reward-driven learning, rather than imitation of human-designed patterns, suggests tool-integrated reinforcement learning represents a promising direction for enhancing LLMs' reasoning capabilities.

To support further research, we open-source our implementation and trained models, enabling the community to build upon and extend ToRL for advancing tool-augmented language models.

## 2 Methodology

### 2.1 Dataset Construction

We constructed our dataset from Olympic-level mathematical competitions questions sourced from Numina-MATH (Li et al., 2024), MATH (Hendrycks et al., 2021), and DeepScaleR (Luo et al., 2025). The initial filtering removed proof-based problems and questions with ambiguous verification criteria, yielding 75,149 verifiable questions. We then applied LIMR (Li et al., 2025), a reinforcement learning data distillation technique, to extract high-quality samples with balanced difficulty distribution. This process resulted in a final dataset of 28,740 questions, which serves as the foundation for all subsequent experiments.

**Problem**

Determine how many zeros immediately follow the decimal point in the decimal representation of $\frac{1}{(6\cdot 10)^{10}}$.

**(a) CoT example**

To determine how many zeros immediately follow the decimal point in the decimal representation of $\frac{1}{(6\cdot 10)^{10}}$, we first simplify the expression.

Given expression: $\frac{1}{(6\cdot 10)^{10}}$

Simplifying the denominator:

$$(6 \cdot 10)^{10} = (60)^{10} = (6^{10} \cdot 10^{10})$$

So, the expression becomes:

$$\frac{1}{6^{10} \cdot 10^{10}}$$

This can be further simplified as:

$$\frac{1}{6^{10}} \cdot \frac{1}{10^{10}}$$

The term $\frac{1}{10^{10}}$ is the one that directly affects the number of zeros immediately following the decimal point.

This is because $\frac{1}{10^{10}}$ is equivalent to moving the decimal point ten places to the left of 1, resulting in a number with nine zeros immediately following the decimal point (and a 1 in the 10th position).

The term $\frac{1}{6^{10}}$ will affect the digits that follow the nine zeros, but it does not change the number of zeros immediately following the decimal point.

Therefore, the number of zeros immediately following the decimal point in the decimal representation of $\frac{1}{(6\cdot 10)^{10}}$ is $\boxed{9}$.

**(b) TIR example**

To determine how many zeros immediately follow the decimal point in the decimal representation of $\frac{1}{(6\cdot 10)^{10}}$, we can break down the problem into the following steps:

1. Calculate the value of $6 \cdot 10$.
2. Raise the result to the power of 10.
3. Find the decimal representation of the reciprocal of this value.
4. Count the number of zeros immediately following the decimal point.

Let's go through these steps using Python code.

~~~
```python
base = 6 * 10
power_value = base ** 10
reciprocal = 1 / power_value
reciprocal_str = f"{reciprocal}"
decimal_part = reciprocal_str.split(".")[1] # get the part after the decimal point
zero_count = 0
for char in decimal_part:
    if char == "0": zero_count += 1
    else: break
print(zero_count)
```
~~~

~~~
```output
17
```
~~~

The number of zeros immediately following the decimal point in the decimal representation of $\frac{1}{(6\cdot 10)^{10}}$ is $\boxed{17}$.

Figure 2: Example of CoT and TIR solution of the problem. TIR enables the model to write code and call an interpreter to obtain the output of the executed code, and then perform further reasoning based on the execution results.

## 2.2 Tool Intergrated Reasoning

Tool Integrated Reasoning (TIR) (Gou et al., 2023; Wang et al., 2023; Shao et al., 2024; Liao et al., 2024; Yang et al., 2024) enables large language models (LLMs) to overcome their computational limitations by incorporating executable code into their reasoning process. Unlike traditional prompting approaches, TIR significantly improves accuracy on numerical computation and logical tasks through an iterative process of reasoning and code execution.

Formally, given a language model $M$, A code interpreter $I$ which executes code written by $M$ to obtain the results of tool calls, and an input question $Q$, TIR constructs a reasoning trajectory $s_k$ at step $k$ as:

$$s_k = r_1, c_1, o_1, \ldots, r_k, c_k, o_k \tag{1}$$

where $r_i$ represents natural language reasoning, $c_i$ denotes generated code, and $o_i$ is the execution result of $c_i$. The iterative generation process follows:

$$(r_k, c_k) = M(Q \oplus s_{k-1}) \tag{2}$$
$$o_k = I(c_k) \tag{3}$$
$$s_k = s_{k-1} \oplus r_k \oplus c_k \oplus o_k \tag{4}$$

This cycle continues until the model produces a final answer, with each step informed by previous code execution results. The dynamic adjustment of reasoning paths based on computational feedback allows the model to verify intermediate steps and correct errors during complex problem-solving.

## 2.3 ToRL: Tool Integrated Reinforcement Learning

We propose Tool Integrated Reinforcement Learning (ToRL), a framework that combines TIR with reinforcement learning directly from base language models without prior supervised fine-tuning. While existing approaches typically apply RL on models already aligned through SFT, our method demonstrates that tool-augmented reasoning capabilities can emerge effectively from base models through properly designed reinforcement learning.

In conventional RL for reasoning, models generate Chain-of-Thought (CoT) trajectories and receive rewards based on answer correctness. However, CoT often fails in scenarios requiring precise computation or exhaustive enumeration. Our approach replaces these pure language trajectories with tool-integrated reasoning paths, where a Code Interpreter module is integrated directly into the RL environment interaction loop.

### 2.3.1 TIR Rollout Framework

To enable the model to automatically output reasoning with code blocks, we use the prompt in Figure 3. During the model's rollout process, when a code termination identifier (`` ```output ``) is detected, the system pauses text generation, extracts the latest code block for execution, and inserts the structured execution result into the context in the format `` ```output\nOBSERVATION\n```\n ``, where `OBSERVATION` is the execution result. The system then continues to generate subsequent natural language reasoning until the model either provides a final answer or produces a new code block. Notably, we deliberately return error messages to the LLM when code execution fails, as we hypothesize that these error diagnostics enhance the model's capacity to generate syntactically and semantically correct code in subsequent iterations.

**Prompt**

A conversation between User and Assistant. The user asks a question, and the Assistant solves it.\nUser: Please integrate natural language reasoning with programs to solve the problem above, and put your final answer within \boxed{}.\nprompt\nAssistant:

Figure 3: Prompt template for TORL, and prompt will be replaced with specific question in training.

We observed that the integration of tool during the rollout process introduces significant GPU idle time, with speed of rollout process inversely proportional to the frequency of tool calls. To maintain training speed within a reasonable range, we introduced a hyperparameter, $C$, which is the maximum number of tool calls the model can make during a single response generation. Once this threshold is exceeded, the system ignores further code execution requests, forcing the model to switch to pure-text reasoning mode.

### 2.3.2 Design Choices of Code Interpreter

1. **Tool Call Frequency Control.** We observed that tool integration during rollout introduces significant GPU idle time, with rollout speed inversely proportional to tool call frequency. To maintain reasonable training efficiency, we introduced a hyperparameter $C$, representing the maximum number of tool calls allowed per response generation. Once this threshold is exceeded, the system ignores further code execution requests, forcing the model to switch to pure-text reasoning mode.

2. **Execution Environment Selection.** To balance training efficiency and effectiveness, we sought a stable, accurate, and responsive code interpreter implementation. We initially tested the Python executor from qwen-agent[1], which offers extremely low latency. However, its execution environment lacks isolation from the training system, potentially allowing execution errors (e.g., segmentation faults from illegal memory access) to compromise the entire training process. We ultimately selected Sandbox Fusion[2], which provides an isolated execution environment. Despite slightly higher latency, it delivers superior stability for sustained training operations.

3. **Error Message Processing.** We implemented specific error handling optimizations to enhance training effectiveness. When Sandbox Fusion encounters execution errors, it generates verbose tracebacks containing irrelevant file path information. To reduce context length and preserve only relevant error information, we extract only the final line of error messages (e.g., `NameError: name 'a' is not defined`).

4. **Sandbox Output Masking.** During loss computation, we mask out the `OBSERVATION` outputs from the sandbox environment, significantly improving training stability by preventing the model from attempting to memorize specific execution outputs rather than learning generalizable reasoning patterns.

### 2.3.3 Reward Design

We implemented a rule-based reward function where correct answers receive a reward of 1, incorrect answers receive -1. In addition, the code interpreter naturally provides feedback on code executability. Based on the correlation between successful code execution and problem-solving accuracy, we introduced an **execution-based penalty**: responses containing non-executable code incur a -0.5 reward reduction.

[1]https://github.com/QwenLM/Qwen-Agent
[2]https://bytedance.github.io/SandboxFusion/

Table 1: Answer Correctness Reward

| Answer Correctness | Reward Value |
|---|---|
| Correct | 1 |
| Incorrect | -1 |

Table 2: Code Executability Reward

| Code Executability | Reward Value |
|---|---|
| Executable | 0 |
| Unexecutable | -0.5 |

## 3 Experiment

### 3.1 Experimental Setup

**Training** All RL experiments are conducted using the veRL (Sheng et al., 2024) framework, with Sandbox Fusion as the code interpreter. We employ the GRPO (Shao et al., 2024) algorithm, setting the rollout batch size to 128 and generating 16 samples per problem. To enhance model exploration, all experiments omit the KL loss and set the temperature to 1. We choose the Qwen-2.5-Math (Yang et al., 2024) series models as the base models for RL. To maximize efficiency, the default number of calls, $C$, is set to **1**. Moreover, only the **Answer Correctness Reward** is retained in the default experiments, without incorporating the **Code Executability Reward**, which will be discussed in § 3.3.

**Evaluation** For evaluation, we use greedy decoding (temperature = 0) across all models. We evaluate our results on several challenging mathematical benchmarks: (1) AIME24, (2) AIME25, (3) MATH500 (Hendrycks et al., 2021), (4) OlympiadBench (He et al., 2024), and (5) AMC23.

### 3.2 Main Results

Table 3: Comparison of different models testing accuracy on mathematical benchmarks. Qwen2.5-Math-1.5B-Instruct-TIR represents the testing of Qwen2.5-Math-1.5B-Instruct in the TIR environment. For a fair comparison, we set the maximum number of tool calls to 1. The best results are presented in **bold**.

| Model | SFT/RL | Tool | AIME24 | AIME25 | MATH500 | Olympiad | AMC23 | Avg |
|---|---|---|---|---|---|---|---|---|
| | | | *Models based on Qwen2.5-Math-1.5B-Base* | | | | | |
| Qwen2.5-Math-1.5B-Instruct | RL | ✗ | 10.0 | 10.0 | 66.0 | 31.0 | 62.5 | 35.9 |
| Qwen2.5-Math-1.5B-Instruct-TIR | RL | ✓ | 13.3 | 13.3 | 73.8 | 41.3 | 55.0 | 41.3 |
| ToRL-1.5B(Ours) | RL | ✓ | **26.7**$_{+13.3}$ | **26.7**$_{+13.3}$ | **77.8**$_{+3.0}$ | **44.0**$_{+2.7}$ | **67.5**$_{+5.0}$ | **48.5**$_{+7.2}$ |
| | | | *Models based on Qwen2.5-Math-7B-Base* | | | | | |
| Qwen2.5-Math-7B-Instruct | RL | ✗ | 10.0 | 16.7 | 74.8 | 32.4 | 65.0 | 39.8 |
| Qwen2.5-Math-7B-Instruct-TIR | RL | ✓ | 26.7 | 16.7 | 78.8 | 45.0 | 70.0 | 47.4 |
| SimpleRL-Zero | RL | ✗ | 33.3 | 6.7 | 77.2 | 37.6 | 62.5 | 43.5 |
| rStar-Math-7B | SFT | ✗ | 26.7 | - | 78.4 | 47.1 | 47.5 | - |
| Eurus-2-7B-PRIME | RL | ✗ | 26.7 | 13.3 | 79.2 | 42.1 | 57.4 | 43.1 |
| ToRL-7B(Ours) | RL | ✓ | **43.3**$_{+10.0}$ | **30.0**$_{+13.3}$ | **82.2**$_{+3.0}$ | **49.9**$_{+2.8}$ | **75.0**$_{+5.0}$ | **62.1**$_{+14.7}$ |

Table 3 presents the performance comparison of different models on mathematical benchmarks. ToRL consistently outperforms baseline models across all tested benchmarks. For the 1.5B parameter models, ToRL-1.5B achieves an average accuracy of **48.5%**, surpassing both Qwen2.5-Math-1.5B-Instruct (35.9%) and Qwen2.5-Math-1.5B-Instruct-TIR (41.3%). The improvement is even more pronounced in the 7B parameter models, where ToRL-7B reaches 62.1% average accuracy, significantly higher than other open-source models with the same base model—representing an impressive **14.7%** absolute improvement.

Figure 4 illustrates the training dynamics across five different mathematical benchmarks. ToRL-7B shows consistent improvement over training steps and maintains a clear advantage over both the baseline without tool integration and Qwen-2.5-Math-7B-Instruct-TIR. This performance gap is particularly notable in challenging benchmarks like AIME24, AIME25, and OlympiadBench, where ToRL-7B achieves **43.3%**, **30.0%**, and **49.9%** accuracy respectively.

### 3.3 Analysis

To gain deeper insights into how RL facilitates the model's tool invocation for reasoning assistance, we quantitatively analyze the model's tool usage and code-writing behaviors in § 3.3.1 and showcase illustrative examples in § 3.3.3. Furthermore, we change critical parameters related to ToRL in the RL framework to examine their effects on both model performance and training efficiency in § 3.3.2.

## 3.3 Analysis

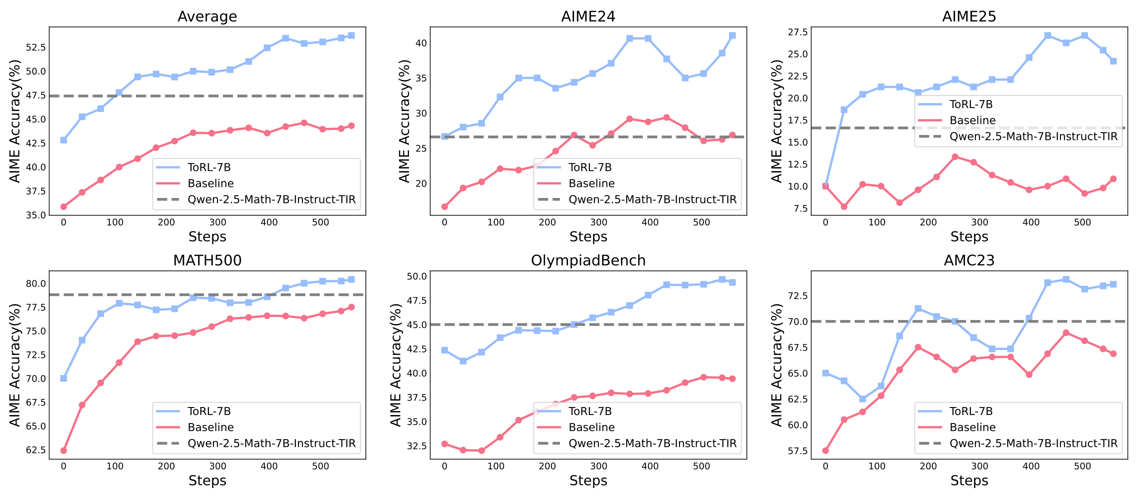
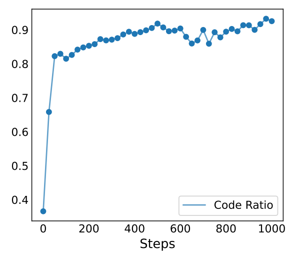
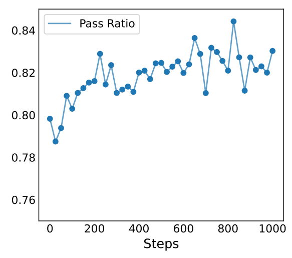


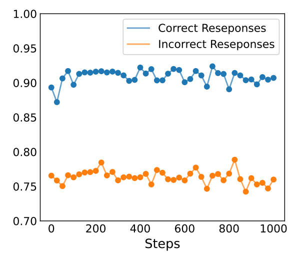
Figure 4: Performance comparison across mathematical benchmarks. The plots show accuracy (%) against training steps for the 7B model evaluated on different benchmarks. Across all benchmarks, ToRL-7B (blue) consistently outperforms both the baseline without tool integration (red) and Qwen-2.5-Math-7B-Instruct-TIR (dashed gray).



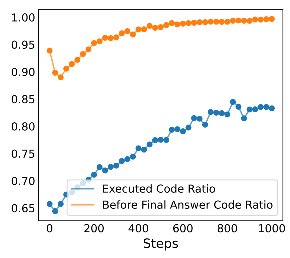
(a) Code Ratio



(b) Pass Ratio



(c) Correct/Incorrect Pass Ratio



(d) Effective Pass Ratio

Figure 5: Tool usage dynamics during training. (a) Code Ratio: proportion of responses that contain code, (b) Pass Ratio: proportion of executed code that runs without errors, (c) Pass Ratio in Correct/Incorrect Responses: comparing execution success rates for correct vs. incorrect responses, and (d) Effective Code Ratio.

### 3.3.1 Part I: Code behaviors during training

Figure 5 provides insights into tool usage patterns during training, including:

**Code Ratio** Figure 5a illustrates the proportion of model-generated responses containing code within the first 100 steps, which increased from 40% to 80%, demonstrating steady improvement throughout training.

**Pass Ratio** Figure 5b presents the proportion of successfully executed code, highlighting a continuous upward trend that reflects the model's enhanced coding capability.

**Pass Ratio of Correct to Incorrect Responses** Meanwhile, Figure 5c depicts the Pass Ratio of Correct to Incorrect Responses, revealing a correlation between code execution errors and final answer accuracy, with correct responses exhibiting higher code pass rates.

**Effective Code Ratio** Furthermore, Figure 5d examines changes in the proportion of effective code, focusing on two key metrics:

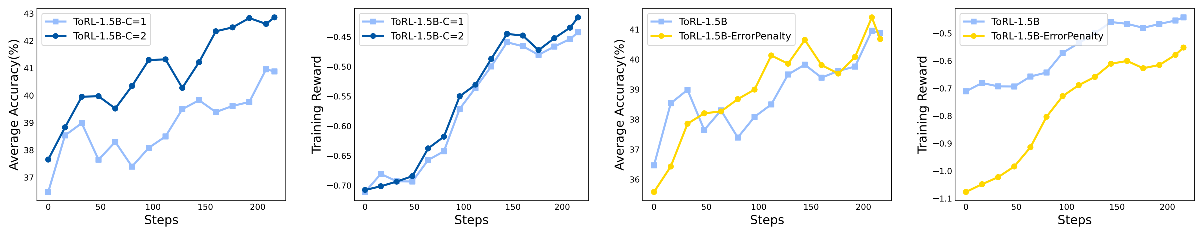
- Executed code ratio, which represents the proportion of generated code successfully executed by the interpreter, excluding any code omitted due to due to exceeding the maximum number of tool calls ($C$).
- Before final answer code ratio, which measures the proportion of code generated before the model provides its final answer, disregarding code used solely for correctness verification post-answer generation.

Both metrics increased over time, indicating that the effectiveness of model-generated code improved as training progressed.

**Takeaway-I**

As the number of training steps increases, the proportion of problems the model solves using code, as well as the proportion of code that can be executed correctly, continues to grow. Meanwhile, during the training process, the model identifies and reduces the generation of ineffective code.

### 3.3.2 Part II: Impact of Key Settings



(a) (b) (c) (d)

Figure 6: ToRL-1.5B-C=2 indicates that the hyperparameter $C$ is set to 2. ToRL-1.5B-ErrorPenalty represents the incorporation of the Code Executability Reward into the reward function, penalizing incorrectly executed code.

In this section, we explore the impact of key ToRL settings on final performance and behavior.

First, we investigate the effect of increasing $C$, the maximum number of tools that can be called in a single response generation during rollout. Specifically, we examine the impact of setting $C$ to 1 and 2 on model performance. As shown in Figures 6a and 6b, setting $C$ to 2 significantly improves performance, increasing average accuracy by approximately 2%. However, as indicated in Table 4, increasing $C$ substantially reduces training speed, necessitating a trade-off between performance and efficiency.

Furthermore, we analyze the impact of incorporating **Code Executability Reward** into the reward shaping. Figures 6c and 6d demonstrate that this reward design does not enhance model performance. We hypothesize that introducing penalties for execution errors may incentivize the model to generate overly simplistic code to minimize errors, which could, in turn, hinder its ability to solve problems correctly.

| $C$ | Average Step Time(s) |
|---|---|
| 0 | 118 |
| 1 | 237 |
| 2 | 288 |

Table 4: The average time per single step under different values of $C$(measured on an 8*A800 GPU system).

**Takeaway-II**

Increasing the $C$ can significantly enhance model performance but also severely reduces training efficiency. Incorporating **Code Executability Reward** does not improve performance.

### 3.3.3 Part III: Cognitive Behavior via RL Scaling

We present some interesting phenomena that occurred in the later stages of model training, which can help us further understand in detail the model's cognitive behavior in solving problems using tools.

Table 5 shows an example of the model adjusting its reasoning based on execution feedback from the code interpreter. The model first wrote the code, but improper handling of `results` caused an indexing error. After receiving the feedback `TypeError: 'int' object is not subscriptable`, it promptly adjusted and produced executable code, eventually inferring the correct answer.

Table 6 demonstrates the model's cognitive behavior of reflection. The model initially solved the problem with natural language reasoning, then verified it through tool but found inconsistencies. Consequently, the model further makes corrections, ultimately generate the correct answer.

**Takeaway-III**

ToRL gives rise to a variety of cognitive behaviors, including obtaining feedback from code execution results, cross-checking with code and natural language.

Table 5: The code initially generate by the model encountered an execution error, then it is corrected by model and is successfully executed.

Question: Given the polynomial $f(x) = x^6 - 5x^5 + 6x^4 + x^2 + 0.3x + 2$, use Horner's method to calculate $f(-2)$ and find the value of $v_1$.

Response: To solve the given problem using Horner's method, we can ...

~~~
```python
def horner(coeffs, x):
...
 return result, results[1]
```
```output
TypeError: 'int' object is not subscriptable
```
~~~

To fix this, we need to ensure that we properly store and access the ...

~~~
```python
def horner(coeffs, x):
 ...
 return result, results
```
```output
v_1 = -7
```
~~~

Thus, the value of $f(-2)$ is $\boxed{325.4}$ and the value of $v_1$ is $\boxed{-7}$.

Table 6: The model first derives an incorrect result based on natural language reasoning, then discovers the error during code verification and makes corrections

Question: After labeling 12 balls with numbers 1, 2, 3, ... 12 ...
What are the numbers on the other three balls taken by Person C?

Response: To solve the problem, we need to find the numbers on the ...
...
The numbers on the other three balls taken by Person C are $2, 10, 13$,
reduced to valid numbers from 2 to 12, so the only valid combination is $2, 10, 13$.
...
Here is the Python code to verify the solution:

~~~
```python
# Define the total sum of the numbers from 1 to 12
total_sum = sum(range(1, 13))
# Each person's sum is total_sum / 3
person_sum = total_sum // 3
...
# Output the balls taken by Person C
print(balls_C_remaining)
```output
[3, 10, 12]
```
~~~

...
The numbers on the other three balls taken by Person C are $\boxed{3, 10, 12}$.

## 4 Conclusion

We propose ToRL, which enables LLMs to integrate tools in reasoning through reinforcement learning, surpassing predefined tool usage constraints. Our results show significant performance gains and emergent reasoning, highlighting ToRL's potential for advancing LLMs in complex reasoning.

## References

[1] 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*.

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

[3] 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*.

[4] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

[5] Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, et al. 2024. Olympiadbench: A challenging benchmark for promoting agi with olympiad-level bilingual multimodal scientific problems. *arXiv preprint arXiv:2402.14008*.

[6] 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*.

[7] Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, and Heung-Yeung Shum Xiangyu Zhang. 2025. Open-reasoner-zero: An open source approach to scaling reinforcement learning on the base model. `https://github.com/Open-Reasoner-Zero/Open-Reasoner-Zero`.

[8] Zhen Huang, Haoyang Zou, Xuefeng Li, Yixiu Liu, Yuxiang Zheng, Ethan Chern, Shijie Xia, Yiwei Qin, Weizhe Yuan, and Pengfei Liu. 2024. O1 replication journey–part 2: Surpassing o1-preview through simple distillation, big progress or bitter lesson? *arXiv preprint arXiv:2411.16489*.

[9] Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Huang, Kashif Rasul, Longhui Yu, Albert Q Jiang, Ziju Shen, et al. 2024. Numinamath: The largest public dataset in ai4maths with 860k pairs of competition math problems and solutions. *Hugging Face repository*, 13:9.

[10] Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025. Limr: Less is more for rl scaling. *arXiv preprint arXiv:2502.11886*.

[11] Minpeng Liao, Wei Luo, Chengxi Li, Jing Wu, and Kai Fan. 2024. Mario: Math reasoning with code interpreter output–a reproducible pipeline. *arXiv preprint arXiv:2401.08190*.

[12] Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Y. Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Tianjun Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. 2025. Deepscaler: Surpassing o1-preview with a 1.5b model by scaling rl. `https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8` Notion Blog.

[13] Yiwei Qin, Xuefeng Li, Haoyang Zou, Yixiu Liu, Shijie Xia, Zhen Huang, Yixin Ye, Weizhe Yuan, Hector Liu, Yuanzhi Li, et al. 2024. O1 replication journey: A strategic progress report–part 1. *arXiv preprint arXiv:2410.18982*.

[14] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

[15] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*.

[16] Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, Chuning Tang, Congcong Wang, Dehao Zhang, Enming Yuan, Enzhe Lu, Fengxiang Tang, Flood Sung, Guangda Wei, Guokun Lai, Haiqing Guo, Han Zhu, Hao Ding, Hao Hu, Hao Yang, Hao Zhang, Haotian Yao, Haotian Zhao, Haoyu Lu, Haoze Li, Haozhen Yu, Hongcheng Gao, Huabin Zheng, Huan Yuan, Jia Chen, Jianhang Guo, Jianlin Su, Jianzhou Wang, Jie Zhao, Jin Zhang, Jingyuan Liu, Junjie Yan, Junyan Wu, Lidong Shi, Ling Ye, Longhui Yu, Mengnan Dong, Neo Zhang, Ningchen Ma, Qiwei Pan, Qucheng Gong, Shaowei Liu, Shengling Ma, Shupeng Wei, Sihan Cao, Siying Huang, Tao Jiang, Weihao Gao, Weimin Xiong, Weiran He, Weixiao Huang, Wenhao Wu, Wenyang He, Xianghui Wei, Xianqing Jia, Xingzhe

Wu, Xinran Xu, Xinxing Zu, Xinyu Zhou, Xuehai Pan, Y. Charles, Yang Li, Yangyang Hu, Yangyang Liu, Yanru Chen, Yejie Wang, Yibo Liu, Yidao Qin, Yifeng Liu, Ying Yang, Yiping Bao, Yulun Du, Yuxin Wu, Yuzhi Wang, Zaida Zhou, Zhaoji Wang, Zhaowei Li, Zhen Zhu, Zheng Zhang, Zhexu Wang, Zhilin Yang, Zhiqi Huang, Zihao Huang, Ziyao Xu, and Zonghan Yang. 2025. Kimi k1.5: Scaling reinforcement learning with llms.

[17] OpenO1 Team. 2024. Openo1. *Github*.

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

[19] 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.

[20] An Yang, Beichen Zhang, Binyuan Hui, Bofei Gao, Bowen Yu, Chengpeng Li, Dayiheng Liu, Jianhong Tu, Jingren Zhou, Junyang Lin, et al. 2024. Qwen2. 5-math technical report: Toward mathematical expert model via self-improvement. *arXiv preprint arXiv:2409.12122*.

[21] Weihao Zeng, Yuzhen Huang, Wei Liu, Keqing He, Qian Liu, Zejun Ma, and Junxian He. 2025. 7b model and 8k examples: Emerging reasoning with reinforcement learning is both effective and efficient. `https://hkust-nlp.notion.site/simplerl-reason`. Notion Blog.