

Extended Abstract

Motivation. Large language models can solve many arithmetic reasoning problems with chain-of-thought prompting, but their intermediate computations are often ungrounded. A model can write a plausible derivation, make a small arithmetic error, and still present the result confidently. This project studies a tool-integrated alternative on the Countdown arithmetic task: the model may emit Python calls, receive executed observations, and decide whether to revise or answer. The central question is not only whether tool access improves correctness, but whether reinforcement learning can shape the model to use tools in a cost-aware way.

Method. I build on the default CS224R Countdown pipeline with Qwen2.5-0.5B. The baseline stages include supervised fine-tuning (SFT), implicit preference optimization (IPO), and online policy-gradient optimization with REINFORCE Leave-One-Out (RLOO). I then extend the pipeline with Python tool use. The model can emit `<python>` code blocks, the environment executes the code, and the resulting value is appended as an `<observation>` before generation continues. Tool-SFT is trained on synthesized tool-use traces, while Tool-RLOO further optimizes rollouts using a reward that combines final correctness with penalties for tool calls, observation tokens, failed calls, and observation misuse.

Implementation. I implemented the Countdown SFT, IPO, and RLOO training loops, then added a tool-integrated environment with parsing, Python execution, rollout continuation, environment-observation masking, and tool-aware evaluation. For Tool-SFT data, I generated staged recovery traces from `asinh15/countdown_tasks_3to4`: the model first tests candidate expressions, receives observations, and either revises or returns a verified answer. For Tool-RLOO, I reused the same query distribution but ignored the supervised completions; the model sampled fresh trajectories and received verifier-based rewards. All final evaluations use 50 held-out Countdown problems with 16 sampled completions per problem.

Results. Tool-RLOO achieves the strongest final performance when evaluated with up to eight Python tool rounds. It improves Pass@16 from 74.0% for Tool-SFT to 86.0%, and from 72.0% for the original SFT baseline to 86.0%. The improvement is not free: average Python calls increase from 3.26 for Tool-SFT to 4.63 for Tool-RLOO. Failed-call rates remain near zero, suggesting that the main cost is additional verification rather than invalid tool use. Original RLOO remains strong at Pass@1, but the tool-integrated policy provides a clearer accuracy-cost tradeoff because its external computation is explicitly measurable.

Discussion and Conclusion. The experiments support the hypothesis that tool-integrated reasoning can improve task performance when the model is allowed to verify and revise candidates. However, they also show that tool use must be reported with cost metrics. Tool-RLOO improves accuracy by spending more Python calls, so the right conclusion is not that it is strictly more efficient, but that it moves the policy along an accuracy-cost frontier. This project reframes arithmetic reasoning as a sequential decision problem: the policy chooses when to reason internally, when to call an external verifier, and when to stop.

Cost-Aware Tool-Integrated Reasoning for Countdown Arithmetic

Patrick Wang
Stanford University
patwang@stanford.edu

Abstract

This project studies cost-aware tool-integrated reinforcement learning for language-model reasoning. In the Countdown task, a model must construct an arithmetic expression that reaches a target number using each provided number exactly once. Standard language models solve this task entirely through text, making them vulnerable to ungrounded arithmetic errors. I extend the default CS224R SFT, IPO, and RLOO pipeline with Python tool calls: the model emits executable expressions, receives numerical observations, and can revise before answering. I synthesize tool-use SFT traces and then optimize a Tool-RLOO policy with a reward that balances final correctness against tool-call cost, observation-token cost, failed executions, and observation misuse. On 50 held-out problems with 16 samples each, Tool-RLOO reaches 86.0% Pass@16, outperforming SFT (72.0%), IPO (74.0%), RLOO (74.0%), and Tool-SFT (74.0%). This gain requires more tool calls, increasing average Python calls from 3.26 for Tool-SFT to 4.63 for Tool-RLOO, highlighting the accuracy-cost tradeoff of tool-integrated reasoning.

1 Introduction

Language models are increasingly used as reasoning agents, but many of their intermediate computations are generated as text rather than grounded in executable operations. This distinction matters for arithmetic and programmatic reasoning: a model can write a convincing derivation while silently making a numerical mistake. Tool-integrated reasoning addresses this by allowing a model to call external tools, observe their outputs, and condition later tokens on those observations. In this project, I focus on Python as a computation and verification tool for Countdown arithmetic.

The Countdown task asks a model to combine a small set of numbers with arithmetic operations to match a target. It is a useful setting for tool-integrated RL because final answers are automatically verifiable, Python observations are exact, and tool usage can be measured precisely. A no-tool policy must solve the entire problem internally. A tool policy can instead test candidate expressions, observe their values, and revise. This changes the problem from one-shot generation into a sequential decision process.

The main research question is: can reinforcement learning improve tool-integrated reasoning while making the cost of tool use explicit? Naive tool access can improve correctness by letting the model repeatedly try expressions, but this may increase latency and compute. In realistic agentic systems, tool calls can be expensive, slow, or unreliable. I therefore evaluate both accuracy and tool-use behavior, including average Python calls, failed-call rate, observation mismatch, and reward.

The project has two parts. First, I implemented the required SFT, IPO, and RLOO baselines for Countdown. Second, I built a tool-integrated extension with synthesized Tool-SFT traces and Tool-RLOO training. The final results show that Tool-RLOO achieves the highest Pass@16 when evaluated with sufficient tool rounds, but it does so by using more Python calls. This supports the framing of tool-integrated reasoning as an accuracy-cost tradeoff rather than a free accuracy improvement.

This report makes three main contributions. First, I implement a tool-integrated Countdown environment with executable Python calls and environment-provided observations. Second, I construct a staged Tool-SFT dataset that teaches verification and recovery behavior from the original Countdown problems. Third, I evaluate cost-aware Tool-RLOO and show that it improves final Pass@16 to 86.0% when given sufficient tool rounds, while making the increased tool-call cost explicit.

2 Related Work

Reasoning and acting with tools. ReAct interleaves natural-language reasoning with environment actions, showing that language models can solve tasks more effectively when they alternate between thinking and acting Yao et al. (2023). Toolformer shows that models can learn API usage patterns from self-supervised data, deciding when to call tools and how to incorporate results Schick et al. (2023). These works motivate the interface used here: the model emits structured tool-call tags, receives observations, and continues generation. This project differs by studying a small, automatically verifiable arithmetic setting where tool cost can be directly included in the reward.

Preference optimization and policy-gradient fine-tuning. Direct Preference Optimization and related implicit preference methods reinterpret policy log-likelihood ratios as rewards for preference learning Rafailov et al. (2023); Azar et al. (2023). The default project includes IPO as a pairwise preference optimization baseline. For online RL, REINFORCE-style updates remain attractive because they are simple and directly optimize sampled rollouts Williams (1992). RLOO reduces variance by comparing each sample against the average reward of other samples from the same prompt Ahmadian et al. (2024). I use this RLOO idea both in the no-tool baseline and in the tool-integrated extension.

Countdown as a verifiable RL task. Countdown provides a compact testbed for RL fine-tuning because each final expression can be checked exactly. The dataset asingh15/countdown_tasks_3to4 contains prompts, number sets, targets, and verifier-compatible ground truth Singh (2025). I use Qwen2.5-0.5B as the base model Qwen Team (2025), following the default project setup. The extension adds an executable Python environment, which turns the verifier from a final-only scorer into an intermediate feedback mechanism.

3 Method

3.1 Base Countdown Pipeline

The base model is first supervised fine-tuned on Countdown reasoning traces. Given a prompt x and response y , SFT minimizes token-level cross-entropy over response tokens while masking prompt and padding tokens. The SFT model is then used as the reference model for IPO and as the initialization for RLOO.

For IPO, the model receives pairs of chosen and rejected completions. Let y_w and y_l denote preferred and dispreferred responses. IPO optimizes a squared loss on the implicit reward margin, where the implicit reward is given by a log-probability ratio between the current policy and reference policy. This encourages the chosen response to have a larger implicit reward than the rejected response without training a separate reward model.

For no-tool RLOO, the policy samples multiple completions for each prompt. Each completion is scored by the Countdown verifier, and the policy-gradient advantage for one sample is computed relative to the mean reward of the other samples from the same prompt. This leave-one-out baseline reduces variance while preserving an online RL update.

3.2 Tool-Integrated Rollout Environment

Figure 1 summarizes the tool-integrated training pipeline. The model is first trained with Tool-SFT to learn the Python-call format, then optimized with Tool-RLOO using rollouts scored by the cost-aware reward.

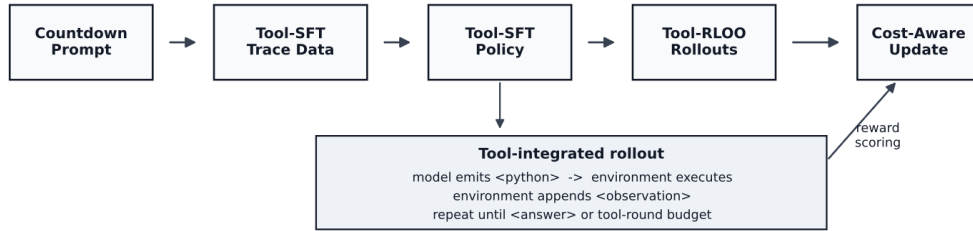


Figure 1: Tool-integrated training pipeline. Tool-SFT teaches the policy to emit Python verification calls; Tool-RLOO then optimizes sampled trajectories with a cost-aware reward.

The extension gives the policy access to Python through a lightweight environment. During generation, the model may emit:

`<python> c </python>`,

where c is Python code, typically a candidate arithmetic expression. The environment parses complete Python blocks, executes the code, and appends the resulting observation:

`<observation> o </observation>`.

The model then continues generation with this observation in context. Generation stops when the model emits `<answer>` or reaches the tool-round budget.

The environment treats observations as external tokens, not policy actions. During Tool-RLOO loss computation, model-generated text is included in the policy-gradient mask, while prompt and environment observation tokens are masked out. This is important because the policy should not receive credit or blame for text that was produced by the environment.

3.3 Tool-SFT Data Construction

The Tool-SFT dataset is synthesized from `asingh15/countdown_tasks_3to4`. For each problem, an exact arithmetic solver identifies valid expressions. I then generate staged tool-use traces in which the model tests candidate expressions with Python, receives observations, compares them to the target, and either revises or returns a verified answer. Each row contains a query, a completion, and a `ground_truth` object. Tool-SFT uses `query` and `completion`; Tool-RLOO later reuses `query` and `ground_truth`.

This staged format was important empirically. A later reasoning-rich synthetic dataset taught richer natural-language traces, but the model often mishandled observations, for example claiming a value matched the target when it did not. The staged recovery dataset produced lower observation mismatch and better tool reliability, so it became the starting point for final Tool-RLOO.

From original Countdown rows to tool traces. The original Countdown dataset provides a single prompt and verifier target. It does not contain intermediate tool calls or observations. To create Tool-SFT data, I convert each original problem into one or more staged supervised examples. The key idea is to expose the model to the interaction pattern it will later use at RL time: propose an expression, receive an executed observation, compare it to the target, and either revise or answer.

For example, an original row contains the prompt:

Original prompt: Using the numbers [44, 19, 35], create an equation that equals 98. Use each number exactly once and return the final answer in `<answer>` tags.

The generated Tool-SFT query prepends the tool protocol and asks the model to solve with Python verification. A first staged completion may intentionally test a wrong expression:

```

<think>
I will use Python as a tool to verify a candidate arithmetic expression.
</think>
<python>
((44 - 35) * 19)
</python>
  
```

Table 1: How one Countdown problem is converted into tool-integrated training data.

Field	Role in training
Original prompt	Natural-language Countdown task with numbers and target. Used as the semantic source problem.
Solver output	A verified expression used to ensure the problem has a correct answer and to generate recovery traces.
query	Chat-formatted prompt plus tool instructions and, for later stages, prior observations.
completion	Supervised model action: a <code><python></code> call, a revision after an observation, or a final <code><answer></code> .
ground_truth	Numbers and target used by the verifier reward during Tool-RLOO and evaluation.

The environment then executes the call and appends the true observation to the next query:

```
<observation>
171
</observation>
```

The next supervised completion teaches the model to reject the wrong candidate and try another:

```
<think>
The observation is 171. Since 171 != 98, it does not match the target.
I need to try a different expression.
</think>
<python>
((19 + 35) + 44)
</python>
```

After the environment returns 98, the final staged completion returns the verified answer:

```
<think>
The observation is 98. Since 98 == 98, it exactly equals the target.
</think>
<answer>((19 + 35) + 44)</answer>
```

This construction differs from ordinary SFT in two ways. First, the model is trained on the syntax of tool use, including when to stop after a `<python>` block. Second, the model is trained to ground its next action in the observation supplied by the environment. At RL time, the `completion` field is no longer used; Tool-RLOO samples fresh actions from the `query`, executes any Python calls, and scores the final transcript against the same `ground_truth`.

3.4 Cost-Aware Tool Reward

Tool-RLOO optimizes the reward

$$R = R_{\text{correct}} - \lambda_{\text{call}} N_{\text{calls}} - \lambda_{\text{token}} N_{\text{tool-tokens}} - \lambda_{\text{error}} N_{\text{failed-calls}} - \lambda_{\text{mismatch}} N_{\text{obs-mismatch}}. \quad (1)$$

Here R_{correct} is the final verifier reward, N_{calls} is the number of Python calls, $N_{\text{tool-tokens}}$ counts tokens in tool observations, $N_{\text{failed-calls}}$ counts invalid or failed Python executions, and $N_{\text{obs-mismatch}}$ flags cases where the latest successful observation does not match the target but the model still returns an answer. The final experiments use

$$\lambda_{\text{call}} = 0.02, \quad \lambda_{\text{token}} = 0.0, \quad \lambda_{\text{error}} = 0.1, \quad \lambda_{\text{mismatch}} = 0.3.$$

This reward is designed to encourage Python use when it improves correctness while discouraging unnecessary, invalid, or misinterpreted tool calls.

4 Experimental Setup

Models and data. All experiments use Qwen2.5-0.5B. The no-tool baselines use the default Countdown pipeline: SFT, IPO, and RLOO. The tool-integrated models use a custom staged Tool-SFT

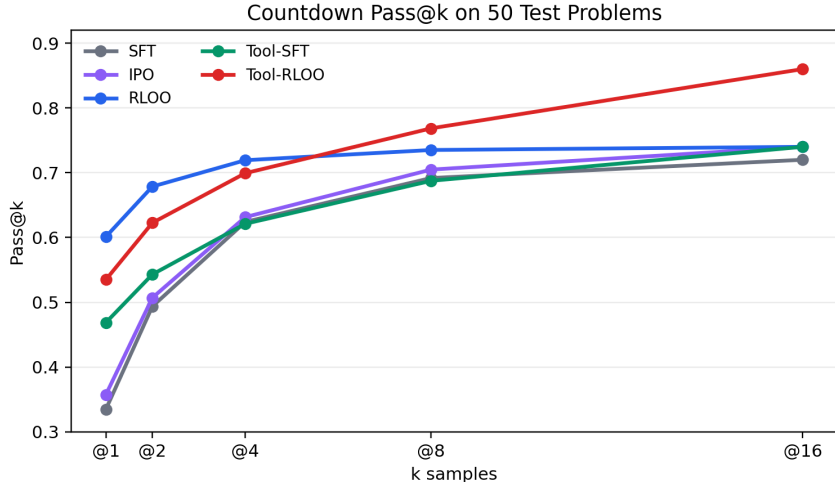


Figure 2: Pass@k comparison on 50 held-out Countdown problems with 16 samples per problem. Tool-RLOO uses up to eight Python tool rounds at evaluation.

Table 2: Countdown pass@k comparison. Tool-RLOO reaches the highest Pass@16, while original RLOO remains strongest among no-tool baselines.

Method	Pass@1	Pass@4	Pass@8	Pass@16
SFT	33.5%	62.4%	69.2%	72.0%
IPO	35.8%	63.1%	70.5%	74.0%
RLOO	60.1%	71.9%	73.5%	74.0%
Tool-SFT	46.9%	62.1%	68.8%	74.0%
Tool-RLOO	53.5%	69.9%	76.8%	86.0%

dataset with 10k training rows derived from the same Countdown problem distribution. Tool-RLOO is initialized from the staged Tool-SFT checkpoint and trains on the same query distribution, but it ignores the supervised completions and samples fresh rollouts.

Evaluation. All reported results use 50 held-out Countdown problems and 16 sampled completions per problem. I report corrected Pass@k for $k \in \{1, 4, 8, 16\}$. For tool models, I additionally report average Python calls, failed-call rate, observation mismatch rate, average reward, and observation-token cost. The final Tool-RLOO evaluation allows up to eight Python tool rounds per problem with a maximum generation length of 3072 tokens. This larger evaluation budget matters because many trajectories need more than five candidate checks before finding a valid expression.

Baselines. The main baselines are original SFT, IPO, and RLOO without tools. These models reason entirely in text and are scored only by the final verifier. The tool baselines are Tool-SFT, which imitates staged tool-use traces, and Tool-RLOO, which further optimizes sampled tool trajectories under the cost-aware reward.

5 Results

5.1 Quantitative Evaluation

Table 2 shows the main result. Original RLOO is a strong no-tool baseline, improving Pass@1 from 33.5% for SFT to 60.1%. However, its Pass@16 saturates at 74.0%. Tool-RLOO reaches the same broad level at low k , but improves substantially at high k , reaching 86.0% Pass@16. This indicates that the tool policy benefits from additional sampled trajectories and tool-verification rounds: at least one of the 16 samples often finds a verified solution.

Table 3: Tool-use metrics for tool-integrated models. Tool-RLOO improves accuracy by spending more Python calls, while failed-call rates remain near zero.

Method	Avg. Calls	Failed Rate	Obs. Mismatch	Avg. Reward
Tool-SFT	3.26	0.04%	4.88%	0.394
Tool-RLOO	4.63	0.08%	5.38%	0.431

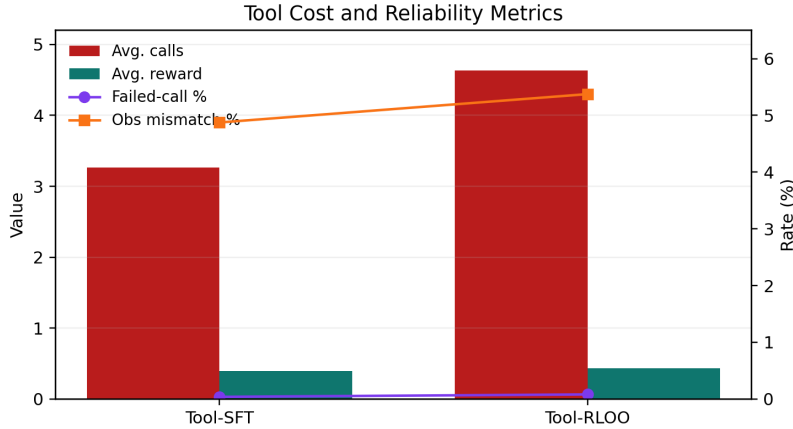


Figure 3: Tool cost and reliability metrics. The Tool-RLOO accuracy gain comes with higher average tool-call cost.

5.2 Tool Efficiency and Reliability

The tool metrics clarify the nature of the improvement. Tool-RLOO does not become more tool-efficient in the narrow sense of using fewer calls. Instead, it uses more Python calls per sample: 4.63 versus 3.26 for Tool-SFT. Failed calls remain rare for both models, so the added cost mostly reflects additional valid verification attempts. Average reward still improves, from 0.394 to 0.431, because the correctness gain outweighs the call penalty under the chosen reward coefficients. This is the central accuracy-cost tradeoff of the project.

Effect of rollout budget. A key empirical finding is that Tool-RLOO is sensitive to the allowed number of tool rounds. With a five-round evaluation budget, the same Tool-RLOO checkpoint reached only 74.0% Pass@16, matching Tool-SFT. Increasing the evaluation budget to eight rounds raised Pass@16 to 86.0%. This suggests that the learned policy can use additional verification opportunities productively, but that the improvement comes from a larger interaction budget rather than a lower-cost policy.

5.3 Qualitative Example

The following trajectory illustrates successful tool-integrated recovery. The model first tries two wrong expressions, uses observations to reject them, and then returns a verified expression.

Prompt:

Using the numbers [44, 19, 35], create an equation that equals 98.

Tool-RLOO completion:

```
<think>
I will use Python as a tool to verify the candidate arithmetic expression.
</think>
<python>
((44 - 35) * 19)
</python>
<observation>
```

```

171
</observation>
<think>
The observation is 171. Since 171 != 98, it does not match the target.
</think>
<python>
((44 * 35) * 19)
</python>
<observation>
29260
</observation>
<think>
The observation is 29260. Since 29260 != 98, it does not match the target.
</think>
<python>
((19 + 35) + 44)
</python>
<observation>
98
</observation>
<think>
The observation is 98. Since 98 == 98, it exactly equals the target.
</think>
<answer>((19 + 35) + 44)</answer>

Score: 1.0.

```

This example also shows why additional tool rounds can help. The policy needs multiple failed attempts before finding the correct expression. When evaluation was limited to five rounds, some trajectories were cut off before they could issue enough candidate checks; with eight rounds, Pass@16 increased to 86.0%.

6 Discussion

The strongest conclusion is that tool-integrated reasoning improves final problem-solving performance when the model is allowed enough interaction budget. Tool-RLOO outperforms all baselines at Pass@16, but it does not do so by reducing tool cost. Instead, it spends more Python calls to explore and verify more candidate expressions. This is still useful: the tool calls are valid and interpretable, and the accuracy gain is large. But it means the result should be described as a movement along an accuracy-cost frontier rather than a strict efficiency improvement.

The experiments also reveal that prompt and data format matter. A reasoning-rich synthetic Tool-SFT dataset produced natural traces, but the model frequently mishandled observations, sometimes claiming that a wrong observation matched the target. The staged recovery dataset was less linguistically rich but more reliable: it taught a simple pattern of testing, comparing, and revising. This suggests that tool-use data quality should prioritize observation grounding, not just fluent explanations.

There are several limitations. First, the tool is restricted to Python expression evaluation for arithmetic. This makes execution safe and evaluation clean, but it is simpler than real agentic tool use. Second, the final Tool-RLOO result depends on an increased test-time tool budget, so future work should study reward schedules that explicitly optimize the Pareto frontier between accuracy and calls. Third, all final evaluations use 50 held-out problems; larger evaluation sets would reduce variance and provide a more stable comparison among high- k metrics.

6.1 Limitations

The final evaluation uses 50 held-out Countdown problems, so high- k results may have nontrivial variance. The Python tool is also limited to arithmetic expression verification, which is cleaner and safer than open-ended code execution but less representative of broad tool-using agents. Finally, Tool-RLOO improves accuracy by increasing the number of Python calls. It should therefore be interpreted as improving the accuracy-cost tradeoff under the chosen budget, not as strictly reducing tool usage.

7 Conclusion

This project reframes arithmetic reasoning as a decision problem: the model must choose between internal reasoning and external computation. Python tool calls provide grounded intermediate feedback, and Tool-RLOO achieves the best final accuracy, reaching 86.0% Pass@16. However, this gain requires more Python calls, highlighting the accuracy-cost tradeoff in tool-integrated reasoning. The main takeaway is that tool integration can substantially improve final performance, but evaluation must report both accuracy and cost.

8 Team Contributions

- **Patrick Wang:** Completed the full project implementation and evaluation. This included implementing SFT, IPO, and RLOO components for the default Countdown pipeline; building the Python tool-call parser and execution environment; generating staged Tool-SFT data; implementing cost-aware Tool-RLOO; running Modal training and evaluation jobs; analyzing pass@k, tool-cost, and observation-mismatch metrics; and preparing the poster and final report.

Changes from Proposal The proposal initially considered a broader set of tools, including Python, retrieval, search, and shell-style environments. The final project narrowed the scope to Python verification on Countdown arithmetic. This made the environment automatically verifiable and allowed a cleaner study of the accuracy-cost tradeoff.

References

- Arash Ahmadian, Chris Cremer, Quentin Gallouedec, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Noah A. Smith, and Sara Hooker. 2024. Back to Basics: Revisiting REINFORCE Style Optimization for Learning from Human Feedback in LLMs. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.
- Mohammad Gheshlaghi Azar, Mark Rowland, Bilal Piot, Daniel Guo, Daniele Calandriello, Michal Valko, and Remi Munos. 2023. A General Theoretical Paradigm to Understand Learning from Human Preferences. *arXiv preprint arXiv:2310.12036* (2023).
- Qwen Team. 2025. Qwen2.5 Technical Report. <https://qwenlm.github.io/>.
- Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. 2023. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. In *Advances in Neural Information Processing Systems*.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems*.
- Anikait Singh. 2025. countdown_tasks_3to4. Hugging Face Dataset: https://huggingface.co/datasets/asinh15/countdown_tasks_3to4.
- Ronald J. Williams. 1992. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning* 8 (1992), 229–256.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations*.