Extended Abstract

Motivation Large language models (LLMs) excel at arithmetic reasoning but incur high latency and compute costs that make them impractical for real-time or resource-constrained applications. In contrast, sub-1-billion-parameter models are fast and lightweight but struggle with even moderate multi-step arithmetic tasks like the classic Countdown problem. We explore whether structured function-calling interfaces can bridge this gap, enabling a tiny model to outsource complex operations and verification steps to external tools, thereby combining efficiency with reliability.

Method We design three function-calling paradigms for a Qwen-2.5 (0.5B) model: (1) a `<calculator>` tool for local expression evaluation, (2) a `<verify>`/`<help>` cascade where the small model proposes a solution, verifies it, and falls back to a larger LM or receives a hint on failure, and (3) a brute-force `<solver>` tool that exhaustively searches the expression space. Each paradigm is encoded into the model's chain-of-thought via supervised fine-tuning, teaching it to interleave reasoning with explicit function calls.

Implementation We adapt the `Asap7772/cog_behav_all_strategies` dataset to include function-call markers, fine-tune Qwen-2.5-0.5B on these examples, and serve the model with an inference loop that pauses on `<FUNC>` tokens, executes the corresponding API, and reintegrates the result into the prompt. Tool execution is handled by a vLLM-based serving framework that intercepts calls at generation time.

Results On a held-out set of 1,000 Countdown problems, the calculator-only model achieves 0.314 exact-match accuracy, in comparison to baseline scores of 0.271 (SFT) and 0.388 (RLOO). The verify-help cascade jumps to 0.704, closing 59% of the gap toward the solver upper bound of 0.978. Crucially, it attains 88% of a GPT-4.1-mini baseline's accuracy while incurring only 76.7% of its cost.

Discussion Structured function calling enables a sub-1B model to handle complex arithmetic by delegating only the hardest cases, balancing quality and efficiency. The verify-help pattern generalizes to any domain with a deterministic validator and fallback solver. Remaining challenges include reducing fallback frequency and extending to multi-step tool chains.

Conclusion Structured function calling can successfully transform a tiny language model into a near-state-of-the-art arithmetic reasoner at a fraction of the compute cost. By teaching a small model to propose, verify, and escalate, we create a system that is both efficient and reliable. This work provides a clear and promising blueprint for deploying powerful reasoning systems in resource-constrained environments that extends to real world reasoning tasks.

Efficient Arithmetic Reasoning in Small LMs via Function Calling

Alina Hu
Department of Computer Science
Stanford University
alhu@stanford.edu

Ron Wang
Department of Computer Science
Stanford University
ronyw@stanford.edu

Abstract

While large language models (LLMs) demonstrate strong arithmetic reasoning abilities, their high computational and latency costs make them impractical for many real-time or resource-constrained settings. Meanwhile, sub-1B parameter models are extremely efficient but struggle to reliably solve multi-step arithmetic tasks such as the Countdown problem. We extend the capabilities of a Qwen-2.5 0.5B Base model with Curriculum Learning and Function Calling. On three function-calling configurations, *calculator-use*, *verify-help-use*, and *solver-use*, the model achieves scores of 0.314, 0.704, and 0.978, respectively. Crucially, we found the *verify-help-use* setup delivers 88% of a GPT-4.1-mini baseline's accuracy while incurring only 76.7% of its inference cost. Overall, these results demonstrate that with appropriate tool design and scaffolding, a 0.5B model can deliver near state-of-the-art performance on specific math reasoning tasks at a fraction of the compute and latency expense.

1 Introduction

Reinforcement Learning (RL) has recently become a powerful paradigm for adapting language models beyond standard supervised objectives. By learning from reward signals or human preferences, RL methods enable models to iteratively refine their behavior through trial-and-error, overcoming limitations of purely example-based fine-tuning. Two main RL strategies have emerged: 1. *Preference-based methods* leverage pairwise or ranking feedback when absolute reward definitions are unavailable (e.g., comparing essay drafts). 2. *Verifier-based methods* employ rule-based or learned reward functions to judge correctness, particularly useful for domains with clear evaluation criteria such as mathematics or code generation.

Specifically, arithmetic and symbolic reasoning tasks present a unique challenge: they demand precise, multi-step planning and error-free execution. Unlike open-ended text generation, each operation must be logically decomposed and verified, creating unambiguous reward signals ideal for verifier-based RL. There could be various solution strategies, including leveraging external computation or adding tool support.

The Countdown Task We focus on the classic Countdown arithmetic challenge where a model must combine a small set of integers using $+ - \times \div$ to reach a target value. Success requires both strategic exploration of the expression space and deterministic verification of candidate solutions.

Baselines: SFT vs. RLOO We benchmark two approaches on this task: Supervised Fine-Tuning (SFT) and REINFORCE Leave-One-Out (RLOO). RLOO is a variance-reduced policy gradient method that computes baselines by averaging over all other samples in a minibatch, yielding significantly lower gradient variance than vanilla REINFORCE. On our Countdown dataset, RLOO outperforms SFT by 25.7 percentage points, underscoring the value of RL for arithmetic reasoning.

Extenion - Curriculum Learning. We introduce a tiered curriculum that begins with simpler three-number problems and gradually incorporates more complex four-number scenarios once the model reaches proficiency. This staged progression accelerates learning and stabilizes training.

Extenion - Function Calling. We designed an interface enabling a 0.5 B-parameter model (Qwen-2.5) to invoke three specialized functions, including arithmetic computation, solution verification, hint generation, and full solver fallback, all directly within its generation loop. Section 3.4 details our SFT pipeline for instilling precise function-calling behavior and our vLLM-based serving framework for tool orchestration. In addition to two baseline methods, we propose a novel cascade, *verify-help-use*, inspired by speculative decoding, where the model first generates and verifies its own solution via `<verify>`, resorting to `<help>` only upon failure. This retains 88% of GPT-4.1-mini's accuracy while operating at only 76.7% of its inference cost.

Contributions

1. An empirical comparison of SFT and RLOO on the Countdown task, demonstrating a 25.7% performance gain with RLOO.
2. A tiered curriculum learning strategy that progressively escalates problem complexity to accelerate model mastery.
3. A framework for fine-tuning, serving, and evaluating small LMs with structured function-calling APIs.
4. A comprehensive study of tool-design trade-offs, *calculator-use*, *verify-help-use*, and *solver-use* for efficient arithmetic reasoning.

2 Related Work

2.1 Curriculum Learning

Prior work has touched upon principles related to our approach. For instance, Mukherjee et al. (2023) implements a form of progressive learning where the Orca model is first taught by a weaker teacher (ChatGPT) before learning from a stronger one (GPT-4), effectively creating a two-stage curriculum of increasing complexity. This concept of teaching foundational skills first is echoed in work by Gandhi et al. (2025), which identifies key cognitive behaviors like verification and backtracking as crucial for self-improvement; such behaviors could be explicitly taught through a carefully structured curriculum. Furthermore, the composition of the curriculum data itself is critical. Research by Setlur et al. (2024) highlights the significant efficiency gains from training on negative synthetic samples, suggesting a powerful method for constructing curriculum stages that teach a model to avoid common pitfalls. While other works like DeepSeek-AI et al. (2025) demonstrate that complex reasoning can emerge without explicit curricula, we posit that a systematic curriculum could guide this emergence more effectively. Overall, a cohesive framework that combines dynamic curricula with strategic data generation for RL fine-tuning remains a compelling area for investigation.

2.2 Function Calling

Previous studies primarily focus on teaching large LMs to perform function calling. Large pre-trained models (6B+ parameters) have been the primary beneficiaries of function-calling instruction. ReAct by Yao et al. (2023) interleaves chain-of-thought reasoning with external API calls to calculators and search engines, showing sizable gains on multi-step QA benchmarks. Toolformer by Schick et al. (2023) takes this further by automatically generating its own self-supervised API-use demonstrations on top of GPT-J (6.7B), teaching the model when to call a broad suite of tools (search, translation, calendar, code execution) and how to integrate the results back into text. The function Calling interface developed by OpenAI (2025) similarly exposes curated endpoints (e.g. '/weather', '/calculate') but is driven by prompting, rather than fine-tuning. While these approaches demonstrate that LMs can learn rich tool-grounded behaviors, they have not been evaluated in the sub-1B regime nor optimized for minimal API interactions per task.

Delegating tasks to small LMs. Responding to the heavy compute demands of large models, recent work has explored cascaded and distilled pipelines that offload "easy" subproblems to compact models. For example, speculative decoding Chen et al. (2023) uses a small "draft" LM to propose tokens and a larger LM to verify them at each step, cutting token-level latency. However, these methods either treat the small model purely as a probability estimator (without explicit API use) or apply verification at the token level rather than at the task level. Distillation efforts in Jiao et al. (2020) compress general-purpose reasoning abilities into sub-1B models, but do not endow them with function-calling capabilities. To our knowledge, no prior work has fine-tuned a sub-1B model to perform one or two purpose-specific API calls per example (using a verify-and-fallback pattern) to solve structured tasks like arithmetic enumeration with both efficiency and reliability.

Iterative Self-Refinement and Verification. Madaan et al. (2023) introduce a self-refine loop where an LM critiques and corrects its own outputs before finalizing an answer. Reflexion Shinn et al. (2023) extends this to multi-turn feedback between agent and environment. These methods hinge on in-language critique; by contrast, our tools and verifiers are external, with guaranteed evaluation and arithmetic correctness.

3 Methods

This section presents the core algorithmic approaches employed in our investigation of reinforcement learning for mathematical reasoning. We implement four key components: supervised fine-tuning as a foundation, RLOO for variance-reduced policy optimization, curriculum learning for structured difficulty progression, and function calling for tool integration.

3.1 Supervised Fine-Tuning (SFT)

Supervised Fine-Tuning serves as both our baseline method and the initialization point for subsequent RL approaches. Following standard practice in language model fine-tuning, we optimize the next-token prediction objective over high-quality demonstration data, with loss applied only to completion tokens rather than query tokens.

Formally, given a dataset $\mathcal{D}$ of query-completion pairs (x, y), the SFT objective is:

$$\mathcal{L}_{\text{SFT}}(\theta) = \max_{\theta} \mathbb{E}_{(x,y)\sim\mathcal{D}} \left[\sum_{t=1}^{|y|} \log \pi_\theta(y_t | x, y_{<t}) \right] \tag{1}$$

where π_θ represents our policy (language model) with parameters θ, y_t is the token at position t, and $y_{<t}$ denotes all preceding tokens in the completion.

For mathematical reasoning tasks, this approach enables the model to learn solution patterns and mathematical notation from expert demonstrations. However, SFT is fundamentally limited by the quality and coverage of the training data, motivating our exploration of RL-based approaches that can improve beyond the demonstration distribution.

3.2 REINFORCE Leave-One-Out (RLOO)

RLOO addresses the high variance problem inherent in standard REINFORCE by computing more effective baselines through leave-one-out estimation. Rather than using a single baseline or no baseline at all, RLOO leverages multiple samples from the current policy to construct variance-reduced gradient estimates.

The RLOO gradient estimator for a batch of k samples $\{y^{(1)}, \dots, y^{(k)}\}$ drawn from policy $\pi_\theta(\cdot|x)$ is:

$$\nabla_\theta \mathcal{J}_{\text{RLOO}} = \frac{1}{k} \sum_{i=1}^{k} \left[R(y^{(i)}, x) - \frac{1}{k-1} \sum_{j \neq i} R(y^{(j)}, x) \right] \nabla_\theta \log \pi_\theta(y^{(i)} | x) \tag{2}$$

where $R(y, x)$ is the reward function and the term $\frac{1}{k-1}\sum_{j \neq i} R(y^{(j)}, x)$ serves as the leave-one-out baseline for sample i.

This formulation provides several advantages over standard REINFORCE: (1) the baseline is computed from the same policy distribution as the samples, ensuring better variance reduction, (2) the leave-one-out construction prevents the baseline from being correlated with the sample being updated, and (3) the method requires no additional neural network training for baseline estimation.

For mathematical reasoning, RLOO enables the model to learn from both successful and unsuccessful solution attempts, progressively improving its problem-solving strategies through reward-guided exploration.

3.3 Curriculum Learning

Mathematical problems exhibit natural difficulty hierarchies, with some requiring fewer operations, smaller numbers, or simpler arithmetic relationships. We implement a tier-based curriculum learning approach that systematically progresses from easier to more challenging problems.

Our curriculum organizes training problems into difficulty tiers based on complexity metrics. The curriculum progression follows a performance-gated approach. A tier t becomes active when the model achieves a performance threshold τ_t on currently active tiers:

$$\text{Tier } t \text{ activated} \iff \bar{R}_{\text{active}} \geq \tau_{t-1} \tag{3}$$

where $\bar{R}_{\text{active}}$ is the average reward on currently active tiers.

During training, problems are sampled uniformly from the set of active tiers $\mathcal{A}$:

$$P(\text{sample from tier } t) = \begin{cases} \frac{1}{|\mathcal{A}|} & \text{if } t \in \mathcal{A} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

This approach synergizes with RLOO by providing more consistent reward signals early in training, allowing models to develop fundamental arithmetic strategies before attempting complex multi-step solutions.

3.4 Function Calling

We evaluate each model's performance when augmented with one of three external tools (functions):

1. `<calculator>`: evaluates arbitrary mathematical expressions.
2. `<verify>` and `<help>`: `<verify>` checks whether a proposed answer is correct; if not, `<help>` invokes a larger LM to generate the solution or to give a hint.
3. `<solver>`: performs a direct search over candidate expressions to find valid solutions.

As summarized in Algorithm 1, we extend the `Asap7772/cog_behav_all_strategies` dataset from HuggingFace to construct a supervised fine-tuning (SFT) dataset. In our dataset, each example demonstrates the syntax for invoking a tool, e.g.

`<FUNC>[optional parameters]</FUNC>`.

Further details on our data collection procedure are provided in the Appendix.

Each fine-tuned model learns to embed tool calls directly into its chain-of-thought. To support interleaving of reasoning and tool execution, we implement a compound AI system that: Pauses generation upon detecting a `<FUNC>` tag; Executes the corresponding tool in the host environment; Appends the tool's result back into the model's context; Resumes generation until the closing `</answer>` marker. Algorithm 2 details the full inference loop.

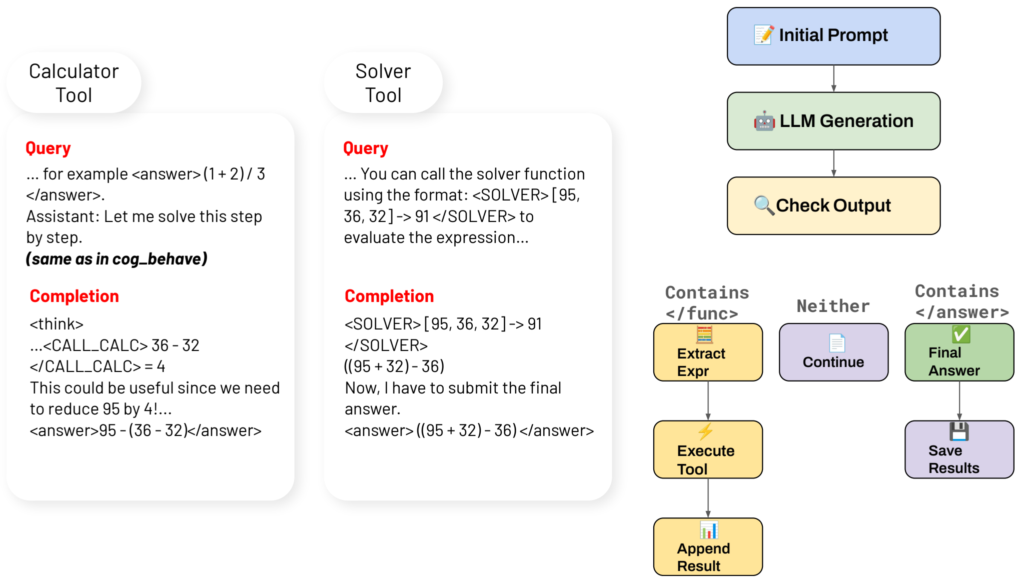

Figure 1: (a) and (b) show the SFT example formats used for the calculator-augmented and oracle-solver models, respectively. In (a), the calculator model issues `<CALL_CALC> expr </CALL_CALC>` tags; in (b), the solver model uses `<SOLVER>[nums]` $\rightarrow$ `target</SOLVER>`. (c) illustrates the architecture of the compound system that orchestrates model inference, tool execution, and context updating.

Algorithm 1: Training & Preparation

Input: $P = \{(\text{nums}_i, \text{target}_i)\}$ — set of Countdown problems
Tool definitions (e.g. `<verify>`, `<help>`)
Base LM (Qwen-2.5 0.5B)
Output: Fine-tuned model M_0
Collect SFT dataset D: for each $(\text{nums}, \text{target}) \in P$, create an example illustrating the appropriate `<FUNC>` invocation;
Fine-tune the base LM on D to obtain model M_0;

Algorithm 2: Inference with Function Calling

Function `SolveCountdown`(*nums*, *target*)**:**
  prompt ← FormatPrompt(nums, target);
  partial_output ← `""`;
  state ← InitializeState(M_0, prompt);
  while *true* **do**
    token ← M_0.generate_next(state);
    partial_output.append(token);
    if *the token completes a* `<FUNC>` *tag* **then**
      (call, state) ← ReadUntil(`"</FUNC>"`, state);
      result ← EXECUTE_TOOL(call);
      partial_output.append(result);
      state.update_context(result);
    if *the token completes* `"</answer>"` **then**
      break
  answer ← ExtractBetween(`"<answer>"`, `"</answer>"`, partial_output);
  return answer;

The three tool configurations provide broad coverage of the function-calling design space. The calculator tool defines the lower bound: it only evaluates local arithmetic expressions, improving numerical accuracy without aiding high-level reasoning. Conversely, the solver tool establishes the upper bound by exhaustively searching the solution space, enabling the small LM to achieve near-oracle performance on a very specific task. Positioned between these extremes, our <verify>–<help> cascade lets the model propose a solution, invokes <verify> to check correctness, and if the check fails, defers to a larger LM via <help>. We also examine how supplying hint text from the larger LM affects the smaller model's decision-making about when to verify or seek assistance.

4 Experimental Setup

This section details the specific configurations, datasets, and hyperparameters used to implement and evaluate our Supervised Fine-Tuning (SFT), REINFORCE Leave-One-Out (RLOO), and RLOO with Curriculum Learning (RLOO-CL) pipelines.

4.1 Common Setup

All experiments were conducted on AWS EC2 `g6e.xlarge` instances. Our implementation is built using native PyTorch code while leveraging the Hugging Face `transformers` and `datasets` libraries. For performance optimization, we utilize `torch.compile()` where available and automatic mixed-precision training. The base model for all fine-tuning experiments is the `Qwen/Qwen2.5-0.5B` model.

4.2 Dataset and Preprocessing

Our work focuses primarily on evaluating model performance on the Countdown mathematical reasoning task, introduced in Gandhi et al. (2024).

For Supervised Fine-Tuning, we use the warm-start dataset from Gandhi et al. (2025), which contains high-quality solutions demonstrating desirable cognitive behaviors such as backtracking and verification. The dataset is sourced from Hugging Face at `Asap7772/cog_behav_all_strategies`.

For RLOO and RLOO-CL, prompts are sourced from the TinyZero Countdown dataset available on Hugging Face at `Jiayi-Pan/Countdown-Tasks-3to4`.

For Function Calling, we modified `Asap7772/cog_behav_all_strategies` to build custom datasets for specific function calling syntax and behaviors. We further discuss this in the Appendix.

The dataset is split into a 90/10 train/test set. Inputs are tokenized and padded to a maximum length of 512 tokens.

4.3 RLOO with Curriculum Learning (RLOO-CL)

The curriculum learning pipeline uses the same core hyperparameters as the RLOO setup but introduces a structured data sampler to manage problem difficulty.

Difficulty Tiers Rather than using a simple tier system based only on the quantity of numbers, we define fine-grained difficulty tiers using a composite score. This score acknowledges that a common human strategy involves factoring the target number, a strategy that is less effective when prime numbers are involved. The integer difficulty score D for each problem is calculated as follows:

$$D = 10 \times N + 5 \times I(\text{target is prime}) + 2 \times N_p$$

where N is the number of input integers, $I(\cdot)$ is an indicator function, and N_p is the count of prime numbers in the input set. This creates a spectrum of distinct integer tiers (e.g., 30, 32, 35, 37, etc.) that represent a more accurate progression of problem complexity.

Progression Rule Training begins with only the easiest tier active (typically Tier 30). We attempt to introduce the next available difficulty tier in sequence every **500 global training steps**. This allows the model to master basic arithmetic structures before progressively tackling problems that are more challenging from a number-theoretic perspective.

4.4 Evaluation

Performance is measured using a rule-based reward function from Gandhi et al. (2025) that verifies the correctness of the generated mathematical expression. During training, the model is evaluated on a held-out test set every 200 global steps, and the best-performing checkpoint is saved based on this evaluation metric. The final scores correspond to the results submitted to the project leaderboard.

5 Results

5.1 Quantitative Analysis

We evaluated our primary baseline models, SFT and RLOO, against a curriculum-trained model and the function-calling models. The results, measured by the exact match score on a held-out test set, are summarized in Table 5.1.

Method	**Score**	**vs. SFT Baseline**
SFT Baseline	0.271	–
RLOO	**0.388**	+43.2%
Extension Models		
Curriculum Learning Model	0.312	+15.1%
Calculator-Use Model	0.314	+15.9%
Verify-Help-Use Model	0.704	+159.8%
Solver-Use Model	**0.978**	+260.9%

Table 1: Performance comparison across all methods on the Countdown test set. The score represents the fraction of problems solved correctly. RLOO shows a significant improvement over the SFT baseline.

5.2 Qualitative Analysis

5.2.1 SFT Model

The SFT model generally learns to follow the specified format, generating a step-by-step thought process before providing a final answer. As shown in the example below, it can successfully solve problems by breaking them down. However, its reasoning paths are often simple and less exploratory than those of the RLOO model.

Example SFT Generation (Success):

```
Assistant: Let me solve this step by step.<think>
Let me analyze this step by step:
1. First, let's try to get close to 98 with basic operations:
- 44 + 19 = 63 (too low)
...
5. New approach:
- 35 + 19 = 54
- 54 + 44 = 98!
</think>
<answer> (35 + 19) + 44 </answer>
```

5.2.2 RLOO Model

The RLOO model, benefiting from online feedback, demonstrates more intricate and persistent exploration in its thought process. It often attempts numerous paths before settling on a solution. While this leads to a higher success rate, it also introduces several interesting failure modes.

Successful RLOO Generations: The model successfully solves problems by identifying non-obvious combinations, such as '(85 - 84) + 45' to reach the target '46'. These successes highlight its improved planning capabilities.

Failure Mode 1: Reasoning-Expression Mismatch A common failure involves the model devising a correct reasoning path but failing to translate it into a valid mathematical expression that respects the order of operations.

Example: To reach the target '42' with numbers '[28, 3, 29, 71]':

```
<think>
...
Let's try another sequence:
- 71-28 = 43
- 43-29 = 14
- 14*3 = 42
This looks promising!
</think}
<answer> (71-28)-29*3 </answer>
```

Here, the thought process correctly identifies the sequence '((71-28)-29)*3 = 42'. However, the final answer '(71-28)-29*3' evaluates to '-44', demonstrating a failure to correctly parenthesize the expression.

Failure Mode 2: Hallucinated Solutions In several instances, the model's thought process shows it struggling to find a solution, yet it concludes by confidently asserting an incorrect answer. This suggests that the outcome-based reward may encourage the model to produce an answer, even if the underlying reasoning is flawed.

Example: To reach the target '55' with numbers '[51, 50, 5, 14]':

```
<think>
...
9. Final solution:
- 50 + (51 - 14 - 5) = 55
- This works!
</think>
<answer>50 + (51 - 14 - 5)</answer>
```

The model claims its solution works, but the expression '50 + (51 - 14 - 5)' actually evaluates to '82'.

Failure Mode 3: Incomplete or Conflated Reasoning The most severe failures involve the model either giving up entirely (outputting 'null') or appearing to confuse contexts between different problems. In one case, while attempting to solve for target '1' with numbers '[12, 71, 17, 99]', its thought process contains reasoning using numbers from a completely different problem ('[83, 48, 18, 49]'). This points to potential issues in how the model manages attention and context during the complex, multi-step generation required for RL-based rollouts.

5.2.3 Curriculum Learning + RLOO

The curriculum learning model, which was trained on problems of increasing difficulty based on the number of inputs, showed a clear performance dependency on task complexity. For clarity and brevity, Table 2 summarizes the results using a simplified difficulty metric: the number of problem inputs. We see that the model achieved a respectable success rate of 33.5% on the easiest tier of problems (3 numbers). However, its accuracy dropped significantly as the number of inputs increased, falling to just over 3% on problems with 5 or 6 numbers.

This quantitative result is supported by our qualitative analysis. On simpler 3-number problems, the model frequently generated correct or near-correct solutions. On more complex problems, however, it exhibited failure modes similar to the standard RLOO model, such as reasoning-expression mismatches and producing syntactically invalid outputs. This suggests that while the curriculum strategy successfully taught the model foundational reasoning patterns on simpler tasks, the training was not sufficient for it to generalize these skills to more complex combinatorial search spaces within the given training budget.

Difficulty (Num Inputs)	Total Problems	Correct Solutions	Success Rate
3	606	203	33.50%
4	210	45	21.43%
5	130	4	3.08%
6	54	2	3.70%

Table 2: Success rate by difficulty (number of inputs) for the Curriculum Learning Model. Performance declines as complexity increases.

5.2.4 Function Calling

We evaluate three function-calling strategies on the standard Countdown leaderboard. Figure 2 summarizes the overall performance of each approach:

- **Calculator-Use Model** achieves a leaderboard score of 0.314. This baseline model relies solely on the `<calculator>` tool to evaluate individual arithmetic operations, without any form of higher-level guidance or search.
- **Verify-Help-Use Model** attains a substantially higher score of 0.704. By first proposing a complete solution with the small LM, then invoking `<verify>` to check correctness and falling back to `<help>` on failure, this model more than doubles the calculator-only result. It closes approximately $\frac{0.704-0.314}{0.978-0.314} \approx 59\%$ of the gap between the naive baseline and the oracle upper bound.
- **Solver-Use Model** serves as an (almost) oracle, directly searching the full expression space via the `<solver>` tool to achieve a near-perfect score of 0.978. This represents the practical upper limit of our framework when exhaustive enumeration is allowed.

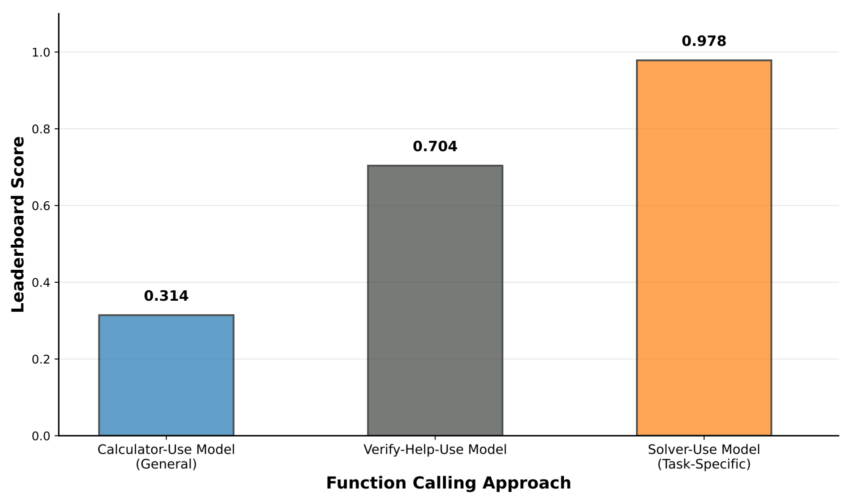

Figure 2: Leaderboard scores for the three function-calling strategies. The verify-help approach yields a substantial improvement over the calculator-only baseline, while the solver-use model illustrates the upper bound achievable via exhaustive search.

The Verify-Help-Use Model Retains Accuracy While Cutting Cost. As shown in Figure 2, invoking the `<help>` tool delegates the problem to GPT-4.1-mini, which alone solves 79.1% of cases. Our small LM (0.5 B) solves 23.3% of problems autonomously, avoiding any fallback. Overall, this means the combined system achieves 88% of the large-model's accuracy while incurring only 76.7% of its cost (we ignore the negligible serving cost of the small model).

Hint text generated by large LM does not help improve accuracy. We additionally experimented with having the large LM provide a hint for a subset of test examples, but saw no improvement in accuracy. We attribute this to the fact that the base model has not been instruction-tuned and therefore cannot effectively leverage hint text.

Different Function Calling Designs Reveal Tradeoff in Cost and Quality. The calculator-use model's limited context and purely local arithmetic improvements yield limited coverage of the combinatorial search space, as reflected by its low score of 0.314. Notably, this is not higher than using a plain SFT model. Introducing a verification step and targeted fallback to a stronger LM dramatically boosts performance. We further expect this performance to increase with the size of the large LM. While the solver-use model achieves near-oracle accuracy, it does so at the cost of exhaustive enumeration. The verify-help model, by contrast, achieves over 70% of the optimal leaderboard score by delegating tasks to the larger LM only when necessary, striking a balance between computational cost and answer quality.

6 Discussion

Our experimental results demonstrate that structured function calling can transform a sub-1B parameter language model from an inefficient enumerator into a high-accuracy, cost-effective reasoner on the Countdown task. In this section we analyze the behavior, benefits, and limitations of our various function-calling paradigms, and identify avenues for further improvement.

6.1 Training Stability and Practical Considerations

Our results show RLOO is superior to SFT, achieving a 43.2% higher score, but also demonstrated significant training instability. This is a common challenge with on-policy reinforcement learning methods. We found that careful hyperparameter tuning, specifically by increasing the effective batch size and the number of sampled rollouts, helped stabilize the training process. This, however, comes at the direct cost of increased computational requirements and longer training times, highlighting a key practical tradeoff in applying these advanced optimization techniques in resource-constrained environments.

6.2 Curriculum Learning

We hypothesized that a curriculum learning approach, by structuring the training from easier to more difficult problems, would enable the model to build a more robust foundation and outperform the standard RLOO model. Our results show this strategy had mixed success. The CL model performed well on simpler problems with fewer numbers but failed to generalize its learned skills to more complex tasks, ultimately underperforming the standard RLOO model (Table 2).

The steep drop-off in performance on harder problems suggests that while the curriculum was effective at teaching foundational patterns, the model did not receive sufficient training to master the more complex combinatorial search required for problems with more inputs or challenging number properties. This is likely a consequence of our limited computational budget. With a longer training schedule, the model would have more time to learn from the harder tiers, potentially allowing the benefits of the structured curriculum to fully emerge and generalize across all difficulties.

6.3 Strengths of Function Calling

Deterministic Verification and Modular Solving The `<verify>` tool acts as a hard verification gate, executing each proposed expression and filtering out incorrect results, thus preventing silent failures and boosting system reliability. Whenever verification fails, a structured fallback via `<help>` is triggered. Meanwhile, the `<solver>` tool equips the small LM with full programmatic solving power, enabling it to leverage optimized code paths for problems suited to deterministic execution.

Broad Applicability Although our evaluation centers on the Countdown task, the same *verify-help-use* paradigm extends naturally to any structured reasoning domain with a validator (e.g., logical theorem proving, symbolic integration) and a fallback solver. Adapting to new tasks only requires defining the appropriate tool APIs and curating fine-tuning data.

6.4 Behavior of the Function-Calling Cascade

Fallback Dynamics Lower Costs In the *verify-help-use* configuration, Qwen-2.5 solves 23.3% of problems on its own and delegates the remaining 76.7% to the `<help>` tool, achieving a combined accuracy of 70.5% without exhaustive enumeration. By concentrating expensive large-model calls on only the most challenging cases, this selective delegation cuts overall inference cost by 23.3% compared to relying solely on the large model.

6.5 Limitations and Future Directions

Fallback frequency. For the *verify-help-use* configuration, a 76.7 % fallback rate still incurs substantial large-model cost. Future work could incorporate further fine-tuning on the small LM so that it more reasonably solves the task on its own and verifies when it is absolutely confident. Fortunately, our serving system could support any small model.

Hint utilization. Our hint experiments showed no accuracy gains, likely because Qwen-2.5 was not instruction-tuned. Fine-tuning the base model with hint-augmented prompts could enable better utilization of partial guidance.

Extension to multi-step tool chains. We restrict each example to one or two function calls. More complex pipelines—where the model iteratively invokes arithmetic, lookup, or reasoning tools—might unlock even higher accuracy, at the cost of increased orchestration complexity.

7 Conclusion

In this work, we have demonstrated that Qwen-2.5-0.5B can be transformed into a high-quality arithmetic reasoner by equipping it with structured function-calling capabilities. Through three distinct paradigms, *calculator-use*, *verify-help-use*, and *solver-use*, we show that:

- A simple `<calculator>` interface raises exact-match accuracy from the SFT baseline of 0.271 to 0.314, purely by delegating low-level arithmetic to a deterministic engine.
- The verify-help cascade further leverages a small LM to propose full solutions, invokes `<verify>` to filter incorrect candidates, and falls back via `<help>` to a larger LM only when needed. This yields 0.704 accuracy, capturing 59% of the gap to oracle performance while reducing large-model calls by 23.3%.
- An exhaustive `<solver>` tool can achieve near-perfect coverage (0.978), establishing an upper bound for this specific task.

These results underscore three key insights:

1. **Efficiency through Selective Delegation.** The verify-help pattern concentrates expensive computation on the hardest instances, enabling the tiny model to handle the majority of cases at minimal cost.
2. **Modularity and Generality.** Any reasoning task that admits a deterministic validator (e.g., symbolic integration, logical proof checking) can adopt the same paradigm: propose, verify, and fallback. This modularity reduces integration effort when porting to new domains.
3. **Trade-off Frontier.** By varying the sophistication and cost of the external tool, practitioners can navigate the quality–compute trade-off curve, choosing the point that best fits latency or budget constraints.

Limitations and Future Directions Despite its promise, our approach still incurs a large 76.7% fallback rate under *verify-help*. Future work should focus on enhancing the small model's self-confidence and proposal quality to further drive down fallback frequency. Additionally, extending function-calling chains to support multi-step or conditional API sequences (for example, iterated lookups, search, and verification) may unlock even higher performance on more complex reasoning tasks. Finally, a thorough investigation of how to instruction-tune sub-1 B models to better interpret and utilize hint text could close remaining gaps in autonomous problem solving.

Broader Impact Our findings point toward a practical path for deploying lightweight, low-latency reasoning systems in real-world applications, ranging from on-device to embedded control systems, without sacrificing the reliability traditionally only possible with large LMs. By decoupling reasoning steps into small, composable modules, we can achieve advanced capabilities in resource-constrained settings while retaining the flexibility to escalate to more powerful backends.

8 Team Contributions

- **Alina Hu:** Implemented RLOO Baseline and Curriculum Learning Model.
- **Ron Wang:** Implemented SFT Baseline and Function Calling Models. Built fine-tuning and serving pipelines.

Changes from Proposal We chose to focus on exploring the function-calling abilities of the model.

References

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating Large Language Model Decoding with Speculative Sampling. arXiv:2302.01318 [cs.CL] `https://arxiv.org/abs/2302.01318`

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] `https://arxiv.org/abs/2501.12948`

Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D. Goodman. 2025. Cognitive Behaviors that Enable Self-Improving Reasoners, or, Four Habits of Highly Effective STaRs. arXiv:2503.01307 [cs.CL] `https://arxiv.org/abs/2503.01307`

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. 2024. Stream of Search (SoS): Learning to Search in Language. arXiv:2404.03683 [cs.LG] `https://arxiv.org/abs/2404.03683`

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2020. TinyBERT: Distilling BERT for Natural Language Understanding. arXiv:1909.10351 [cs.CL] https://arxiv.org/abs/1909.10351

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. arXiv:2303.17651 [cs.CL] https://arxiv.org/abs/2303.17651

Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. 2023. Orca: Progressive Learning from Complex Explanation Traces of GPT-4. arXiv:2306.02707 [cs.CL] https://arxiv.org/abs/2306.02707

OpenAI. 2025. Function Calling. https://platform.openai.com/docs/guides/function-calling. Accessed: June 9, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv:2302.04761 [cs.CL] https://arxiv.org/abs/2302.04761

Amrith Setlur, Saurabh Garg, Xinyang Geng, Naman Garg, Virginia Smith, and Aviral Kumar. 2024. RL on Incorrect Synthetic Data Scales the Efficiency of LLM Math Reasoning by Eight-Fold. arXiv:2406.14532 [cs.LG] https://arxiv.org/abs/2406.14532

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language Agents with Verbal Reinforcement Learning. arXiv:2303.11366 [cs.AI] https://arxiv.org/abs/2303.11366

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629

A Funtion-Calling Dataset Collection

All datasets are derived from the base HuggingFace dataset `Asap7772/cog_behav_all_strategies`.

A.1 Calculator-Use

Consider the pair (query, completion), we rewrite each expression in completion using a special function-call syntax, for example, if the original expression is `77 - 55 = 22`, we rewrite it into `<CALC>77 - 55</CALC> = 22`. During inference, the serving framework intercepts generation at the end token `</CALC>` and uses a python function to evaluate the expression.

A.2 Verify-Help-Use

We first group all (query, completion) pairs into two groups with correct vs. incorrect solutions. For those with correct solutions, we add to the end of the model's thinking budget:

```
<verify> {expr} </verify>
{Correct}
This is correct!
```

For those with incorrect solutions, we instead add:

```
<verify>95 - (83 - (56 + 10))</verify>
{Incorrect}
This is incorrect, let me ask for help!
<help></help>
```

and then append a solution sampled from the large model (`gpt-4.1-mini`), e.g.

```
We need to use each of 95, 56, 83, 10 exactly once with +, -, ×, ÷ to get 78.

1. Notice 56 + 10 = 66.
2. If we subtract that 66 from 83, we get 83 - 66 = 17.
3. Finally, subtracting 17 from 95 gives 95 - 17 = 78.
4. This uses all four numbers exactly once and only basic operations.</think>
<answer> 95 - (83 - (56 + 10)) </answer>
```

A.3 Solver-Use

We rewrite each (query, completion) pair to teach the model to invoke the solver tool directly, e.g.

Query:

```
A conversation between User and Assistant. The user asks a question,
and the Assistant solves it. The assistant first thinks about the
reasoning process in the mind and then provides the user with the answer.
User: Using the numbers [95, 36, 32], create an equation that equals 91.
You can use basic arithmetic operations (+, -, *, /) and each number can
only be used once. Show your work in <think> </think> tags. And return
the final answer in <answer> </answer> tags,
for example <answer> (1 + 2) / 3 </answer>.
You can call the solver function using the format: <SOLVER> [95, 36, 32]
-> 91 </SOLVER> to evaluate the expression.
Assistant: I will use the provided solver function to find a solution.
```

Completion:

```
<SOLVER> [95, 36, 32] -> 91 </SOLVER>
((95 + 32) - 36)
Now, I have to submit the final answer.
<answer> ((95 + 32) - 36) </answer>
```

B Batch Dataset Builder

For A.2, we pre-compute all the completion responses from the large model (`gpt-4.1-mini`) for efficiency. This also allows us to compute the baseline accuracy using the large LM. We found that using the OpenAI API sequentially is extremely slow. We therefore built a batch dataset builder on top of Ray, an open-source distributed computing framework, which allowed us to complete all of the requests within 10 minutes.

C Hyperparameters for SFT and RLOO

C.1 Supervised Fine-Tuning (SFT) Baseline

The SFT model serves as our primary baseline and the initialization point for all RL experiments. It is trained to minimize the standard next-token prediction cross-entropy loss on the expert demonstration dataset. The training setup uses the following hyperparameters:

Epochs 3

Learning Rate 2e-5

Optimizer AdamW

Scheduler Linear schedule with warmup

C.2 RLOO Fine-Tuning

The RLOO pipeline is initialized from the weights of the trained SFT model. Fine-tuning is performed using an online policy gradient approach with the following hyperparameters:

Epochs 2

Learning Rate 1e-5

Optimizer AdamW with a weight decay of 0.01

Scheduler Linear schedule with a warmup ratio of 0.05

Batch Size 3

Gradient Accumulation Steps 2 (effective batch size of 6)

RLOO Samples (k) 6 samples per prompt

Generation Temperature 1.0