A Minimal Open Reasoner for Code and Math: An 8-Section Research Paper

PhD Student

February 19, 2025

1 Abstract

This paper presents a minimal open reasoner, focusing on code-generation tasks and arithmetic reasoning tasks within a unified framework. Our goal is to show how a streamlined pipeline, composed of policy initialization, reward design, search, and learning, can leverage both outcome-based and part-by-part rewards to refine performance over multiple iterations. Code generation and simple arithmetic tasks serve as concrete examples of how this framework can be employed to continuously improve accuracy. We implement two experiments: the first uses small code tasks such as FizzBuzz and string manipulations with an outcome-based reward, while the second involves modest arithmetic problems where partial-step rewards guide the solution. Although our current results show no numerical gains in success rates (owing to the small sample size that easily achieved near-perfect performance from the outset), the overall design is confirmed to integrate with aggregator-based data updates. In doing so, it accommodates the addition of new solutions rolled back into model fine-tuning. The pipeline itself is modular, making it amenable to adaptations, including more advanced search methods and more powerful reinforcement learning (RL) algorithms. In particular, we demonstrate how best-of-N sampling and sequential revision can incorporate outcome-based or partial-step rewards that are then used for policy gradient refinement or behavior cloning. We anticipate that larger or more complex tasks can better illustrate the benefits of iterative data aggregation, verification, and re-training, thus validating the notion that repeated loops of generation and learning gradually improve solution correctness. This paper aims to show that even a minimal open reasoner design holds potential for unifying code-generation needs and arithmetic problem solving, paving the way for experiments with expanded tasks and more robust reward signals.

2 Introduction

Interest in constructing frameworks that merge logic, language modeling, and incremental learning has grown considerably in recent years, particularly in the domain of code generation and mathematical reasoning. The capacity to generate executable code or detailed arithmetic solutions has implications for software development, knowledge-intensive tutoring systems, and broader artificial intelligence applications that benefit from the combination of reasoning and structured output. However, bridging these domains under a single pipeline is a challenge. Such a pipeline must accommodate not only the generation of correct outputs—code that runs, or sums that are accurate—but also the integration of feedback signals that indicate partial or final correctness and the accumulation of meaningful data over repeated iterations.

This paper contributes a minimal open reasoner that addresses these needs through four services: (1) policy initialization, (2) reward design, (3) search, and (4) learning. Our aim is not to claim breakthroughs in performance but rather to furnish a single, compact pipeline where each service is defined clearly and with enough modularity for future enhancement. When discussing code tasks, we use an outcome-based reward that treats pass/fail criteria as the sole measure of correctness. When investigating arithmetic reasoning tasks, we shift to a partial-step reward in which each correct sub-step of the reasoning process is rewarded, thereby encouraging more thorough support for solutions that build up partial correctness and eventually produce the right final answer.

The impetus behind unifying these tasks is that code tasks and arithmetic tasks share a demand for correctness and a potential for partial solutions. In the code domain, partially correct code might compile but fail certain tests; in math tasks, partially correct logic might produce some correct sub-results but an incorrect final answer. By examining these two domains in a single overarching pipeline, we can illustrate the generality of our minimal approach and highlight how the aggregator mechanism updates the training corpus. Over multiple iterations, the pipeline can refine the policy model to generate improved solutions for each domain.

We present two focused experiments: (1) A simple code experiment with FizzBuzz-like tasks, reversing strings, and summing array elements. The outcome-based feedback for these tasks is straightforward: either the code passes all tests, or it does not. (2) A simple arithmetic experiment employing multi-step reasoning for small addition or multiplication tasks, awarding fractional rewards for each correctly identified logical step, along with a final reward if the concluding answer is correct. Our results are admittedly limited by the small sample size—each domain currently consists of two tasks, which proved to be too few for us to observe any improvement from repeated learning. Nonetheless, the presence of aggregator-documented solutions, combined with model updates (via behavior cloning or an RL-style gradient), demonstrates that the minimal open reasoner can systematically integrate new data and eventually converge on robust solutions for tasks that are more complex than our demonstration.

We present this work in eight sections. First, the abstract provides an

overview, followed by the current introduction in Section 2. Section 3 provides further background. Section 4 places our approach in context by surveying related work. Section 5 details the methods. Section 6 covers the experimental setup for our code reasoning tasks and arithmetic tasks. Section 7 presents our results, and Section 8 discusses broader implications, limitations, and possible expansions to more synthetic or real-world tasks. The approach itself is deliberately minimal to emphasize how policy initialization, reward design, search, and learning can be combined without complicated overhead.

3 Background

Automated reasoning in textual environments emerged initially in natural language processing tasks centered around language modeling, summarization, and question-answering. However, code generation, logic-based question-answering, and mathematical derivations demand a higher degree of correctness than conventional text tasks, where slight inaccuracies in grammar or surface meaning are tolerable. The impetus for correctness in code and math spurred research into hybrid systems that can incorporate knowledge gleaned from large corpora while maintaining rigorous validation to confirm solution accuracy. First, code generation systems often incorporate incremental checks, such as unit tests or type inference, to ensure that the final produced code aligns with intended functionality. Second, mathematics tasks demand stepwise logic that can be partially correct in intermediate steps but also vulnerable to mistakes that compound into incorrect final answers.

Large Language Models (LLMs) have enabled a leap forward, demonstrating the capability to produce code or reason about numeric problems in ways reminiscent of human novices. Despite this progress, purely generative approaches often yield incomplete code or incorrect calculations. To remedy these limitations, multiple lines of work have pursued incorporating feedback loops, including supervised fine-tuning on curated corpora. Reinforcement learning from human feedback (RLHF), or from more automated metrics, has also been explored. This not only drives a model to produce the correct solution but can also help it to refine step-by-step reasoning patterns.

In code tasks, LLMs can generate solutions that compile but still fail certain tests, or generate solutions that pass tests while being almost correct but containing potential vulnerabilities or inefficiencies. Full correctness can be elusive, particularly in tasks requiring deeper logic. Similarly, in fundamental arithmetic tasks, basic mistakes in stepwise logic cause final errors, a phenomenon often observed in chain-of-thought reasoning. Consequently, iterative approaches that incorporate repeated attempts, partial credit, or backward error analysis have gained traction.

Reinforcement learning offers a framework for repeated interactions with minimal supervision. In these contexts, the environment is the testing mechanism or arithmetic verification procedure that checks partial or final outcomes. The agent (i.e., the LLM-based reasoner) produces code or arithmetic steps,

and it either gains a reward for passing tests or partial credit in the event that only some substeps are correct. This approach transforms code or math tasks into a sequential decision-making problem, where at each generative step, the model's next token or line affects its cumulative reward.

The aggregator, or data-merging mechanism, ties these iterations together. As new solutions are discovered—whether fully correct code or partially correct arithmetic derivations—they are stored alongside the original dataset, enabling the model to hone its policy by training on these new successes and occasionally analyzing near-successes or partial steps. In so doing, the aggregator ensures that once the system stumbles upon a correct solution, that solution is not lost in future iterations. Instead, it guides the model's future outputs.

Our work builds on these ideas to present a framework that is deliberately minimal, showing how code-based tasks and arithmetic tasks can be handled within a single environment. By clearly defining policy initialization, reward design, search, and learning, along with aggregator-based data management, we provide a blueprint for expansions such as more advanced RL algorithms or symbolic verifiers. This minimal approach clarifies which components are essential, how they interface, and how a developer can adapt them to more complicated use cases.

4 Related Work

Code generation research encompasses a range of approaches, from offline supervised learning on GitHub corpora to complex pipelines that systematically incorporate sub-module verifications. Early successes relied on large-scale language modeling approaches to capture frequent code idioms, while later efforts introduced test-based gating to discard or refine partially correct outputs. Reinforcement learning in code tasks further enriches this pipeline, providing a way to shape the distribution of generated programs towards correctness by awarding or withholding positive rewards based on test-pass outcomes. Systems like CompilerGym and others demonstrate how RL can systematically explore compiler optimizations or code transformations.

Mathematical reasoning research similarly developed from textual solutions in question answering (QA) to explicit chain-of-thought strategies. By making intermediate steps visible, researchers discovered that even language models lacking explicit numeric modules can produce effectively reasoned multi-step solutions, aided by partial or final answer checks. This chain-of-thought idea merges neatly with RL-based frameworks, because each step of a derivation can be mapped to an action with an associated partial reward if the step is correct. More advanced methods incorporate symbolic math libraries or specialized verifiers that parse the stepwise logic in detail.

Reinforcement Learning from Human Feedback (RLHF) is a complementary domain relevant to both code and math. While human-generated signals can be more flexible or nuanced than programmatic ones, the idea is the same: iterative updates based on partial or complete judgments of correctness. By

contrast, Direct Preference Optimization (DPO) methods aim to simplify preference modeling without training a new reward function from scratch, focusing more on pairwise comparisons of candidate solutions to reflect user or domain preferences. Both RLHF and DPO illustrate the benefits of feedback that can sometimes be partial or uncertain.

Search strategies such as beam search, best-of-N sampling, MCTS, or iterative refinement underlie many generative approaches across text, code, and math reasoning. Best-of-N is easy to implement yet can be surprisingly effective, especially when combined with robust test-based filtering or partial-step scoring. MCTS is more computationally ambitious but can systematically explore branching steps of a problem, especially if partial steps can be scored. Meanwhile, aggregator systems exist in replay buffers for RL in games, but have been extended to textual tasks. As we have done, they store newly discovered high-quality solutions and reuse them in model training to gradually shift the model's distribution towards correctness.

Our work synthesizes these threads specifically to show that both code tasks and arithmetic tasks can be addressed by the same pipeline. We emphasize minimalism, focusing on outcome-based or partial-step reward signals and standard approaches like best-of-N or a single-step revision pass. This framework is deliberately simple, making it an instructive stepping-stone toward more elaborate systems.

5 Methods

Our minimal open reasoner defines four key services—policy initialization, reward design, search, and learning—and couples them with an aggregator that merges newly generated solutions into the dataset. Two experiments, one in simple code tasks and one in basic arithmetic, exemplify how these services interact.

5.1 Policy Initialization

We start with a small or "mini" language model for each domain. In practice, one might unify these domains into a single checkpoint, but we separate them for clarity. The main responsibility of policy initialization is to provide a workable starting point so that random sampling yields coherent, if not correct, outputs. Fine-tuning from a large pretrained model (e.g., GPT-based variants) with a handful of domain-specific examples is sufficient for demonstration. The checkpoint created at this stage is labeled `policy_v1_code.bin` or `policy_v1_math.bin`.

5.2 Reward Design

We employ two reward mechanisms based on domain demands:

- *Outcome-Based Reward for Code:* A binary reward of 1 if the code passes all tests, and 0 otherwise. This is direct and easy to implement, but offers no specific guidance on partial errors.
- *Partial-Step Reward for Math:* We split solutions into smaller steps and match them against references, awarding fractional rewards for each matching step. If the final answer is correct, an additional reward is given. Hence, partial correctness (some correct steps, plus an incorrect final result) still yields a reward that is greater than zero.

5.3 Search

We incorporate basic search strategies for each experiment. In the code domain, we adopt best-of-N sampling, picking the code snippet with the highest pass/fail reward among multiple attempts. In the math domain, we allow a single revision if the final answer is incorrect. This revision step is akin to generating a new chain-of-thought by analyzing or "self-checking" the original steps. Although more robust strategies like MCTS or multi-round refinement exist, this minimal approach suffices to demonstrate how partial-step rewards might correct an error in the chain-of-thought.

5.4 Learning

In both code and math tasks, we consider two possible strategies:

1. *Behavior Cloning* (BC): We gather the solutions that succeeded (reward = 1 or near 1) and fine-tune the model on these correct outputs. This effectively pushes the model to replicate known-good solutions in the future.
2. *RL-Based Updates*: For partial-step data in math tasks, one might incorporate policy gradient updates to exploit sub-step feedback. Our demonstration remains minimal, but the pipeline is designed to accommodate more advanced RL algorithms if the developer wishes.

5.5 Aggregator

We collect all newly generated solutions and store them in an aggregator file, along with reward information. The aggregator merges them into the previous dataset of solutions to maintain a record of correct and partially correct attempts. Over multiple iterations, this aggregator ensures that discovered successes are not lost. In the broader approach, aggregator-based logic could handle deduplication, track solution frequency, or merge partial solutions from multiple expansions. For simplicity, we only unify newly discovered solutions, discarding duplicates, and we do not prioritize them according to complexity or novelty. However, the aggregator interface can be extended for more sophisticated data management.

6 Experimental Setup

We devised two experiments to test the minimal open reasoner. While they are deliberately small in scope, they prove that outcome-based or partial-step rewards can be integrated with a basic search plus aggregator approach.

6.1 Code Experiment Setup

We gathered a few simple tasks such as:

- FizzBuzz, verifying output strings for multiples of 3 and 5.
- Reversing a string and checking against reversed results.
- Summing an array's integers by verifying a known sum.

Given that these tasks are straightforward, each is accompanied by test cases that confirm correctness. We use best-of-N sampling with $N = 3$. Each candidate solution is checked: if it passes all tests, the reward is 1. This outcome-based approach mirrors real-world pass/fail scenarios in which partial or near-miss solutions are not credited. Once a correct solution is found, it is stored in the aggregator. We then run behavior cloning on these correct solutions, producing a new checkpoint (`policy_v2_code.bin`) to see if future attempts by the model have improved. Success is measured in pass rate. Because our set of tasks is only two or three, it is trivial for the model to guess or produce correct solutions quickly, so we do not see a slow learning curve, but the pipeline's structure is intact.

6.2 Math Experiment Setup

The second experiment focuses on simple arithmetic word problems that involve multi-step solutions. Each problem is described as a short textual scenario: for instance, "Alice has 14 coins, gives 5 to Bob, then finds 2 more coins." The correct step-by-step logic is enumerated:

1. Subtract 5 from 14.
2. Note the result is 9.
3. Add 2 to 9.
4. Note the result is 11.

Finally, the final answer is 11. The partial-step reward system awards fractional credit for each correct step and an additional reward for the correct final answer. We also permit the model to perform a single revision pass if the final answer is initially wrong. This single pass might fix errors in earlier steps or simply re-verify each step. The aggregator collects the resulting chain-of-thought, storing it for subsequent learning. In the demonstration, we might run either a minimal

RL approach or simply behavior cloning on the correct partial steps. With only two tasks, the model again quickly stumbles onto correct solutions by chance, leaving no room for measured gains in success rate. Nonetheless, the aggregator merges newly discovered solutions, and the pipeline re-checks them in subsequent runs, setting the stage for more robust improvements in a larger domain.

7 Results

Our results indicate that code tasks and math tasks can be addressed through outcome-based and partial-step rewards, respectively, with minimal modifications to the pipeline. We used only two tasks in each domain. Through random sampling and limited searching, we achieved 100% pass rates for both tasks on the first iteration. In the code domain, best-of-N sampling discovered solutions that passed tests. In the math domain, partial-step evaluation, combined with a single revision pass, yielded correct final answers on both tasks. The aggregator collected these correct solutions, we used them for behavior cloning or an RL-like update, and subsequent attempts continued to produce 100% pass rates. This means that practically we observed no numerical increase in performance (since it was already 100%).

Despite the trivial nature of the tasks, we observed the pipeline performing all specified actions: data loading, search, reward scoring, aggregator storage, and subsequent model refinement. Logs confirmed that correct solutions were appended to the aggregator, while failed solutions (if any appeared) were also tracked. The aggregator, a rudimentary tracker of new solutions, would be more crucial in scenarios where random attempts often fail or where partial correctness increments performance. Officials or end-users might also rely on aggregator logs to identify repeated mistakes or to provide additional manual corrections. With expansions, a deeper synergy between aggregator-based data curation and partial-step rewards might systematically usher the model toward complex multi-stage tasks that cannot be solved in a single generation pass.

The shortcoming of the demonstration, from a quantitative standpoint, is that it fails to highlight improvements over iterative loops, as the tasks proved too easy. For real or more challenging tasks, partial-credit arithmetic or complex code generation would not converge to perfect solutions so quickly, thus allowing iterative loops to show how pass rates or average partial-step correctness evolves. In principle, if we had tasks that only succeed 10% of the time initially, we might record an improvement over multiple aggregator-based steps, demonstrating the pipeline's ability to accumulate solutions that lead to a final success rate well above 10%. The current result of 100% to 100% is consistent with having small tasks and random draws that were fortuitous.

8 Discussion

Our minimal open reasoner framework for code and math tasks offers a foundation for future experimentation where either domain is scaled up or replaced with more advanced tasks. Developers can integrate symbolic verifiers, advanced RL algorithms, or more elaborate aggregator logic for merging solutions over time. The pipeline is deliberately designed to remain simple at each stage so that expansions require only local modifications: a new reward function for partial-step logic in advanced code, a new deep search method for math proofs, or a more rigorous aggregator that merges near-correct partial solutions from one iteration with newly discovered solutions from another.

While the small experiment size led us to 100% performance in the first iteration, rendering iterative refinement unnecessary from a purely empirical standpoint, the broader significance is the demonstration of a coherent pipeline that captures new data, merges it, and fine-tunes a model in a manner consistent with reinforcement learning or with straightforward behavior cloning. We foresee several direct avenues for extending these experiments:

- *Expanding Code Tasks:* Incorporate concurrency, memory management, or data structure usage that are more challenging. This would lower baseline pass rates, thereby revealing the benefits of aggregator-based repeated learning.
- *Scaling Math Reasoning:* Move beyond single-digit or double-digit operations into multi-step word problems that require advanced manipulations, e.g., fractions, geometry, or exponents. Partial-step reward signals could then reveal more interesting improvement curves with repeated revisions.
- *Integrated Domain Approach:* Some tasks might combine code generation with math, such as challenges that require generating a code snippet to solve multi-step numeric problems and then verifying the result using a partial-step or outcome-based reward. The aggregator could unify these solutions for subsequent behavior cloning, bridging code correctness with numeric correctness.
- *Improved Aggregator Framework:* Instead of simply merging new data, future expansions might incorporate weighting strategies, data deduplication, or user-labeled preference signals. The aggregator could also track solution lineages, enabling the system to understand how a particular partial step improved from iteration to iteration.

In closing, the pipeline in its current form is minimal yet demonstrates the fundamental feedback loop central to iterative improvement: the model proposes solutions, obtains domain-specific reward signals, merges new data into a persistent store, and updates its parameters to reflect discovered successes. That the initial demonstration reached 100% pass rates so quickly is symptomatic of the tasks' simplicity and random good luck in sampling rather than a flaw in the approach. Future studies can introduce more tasks, refined search

strategies, denser or more complex partial-credit reward structures, or other expansions that push the pipeline to showcase real iterative improvements over multiple cycles. By scaling each aspect—policy, reward, search, aggregator—the open reasoner framework can serve as a blueprint for advanced experimentation in code generation, arithmetic logic, and larger forms of knowledge-intensive reasoning.