Research Report: A Multi-Phase Reasoning Framework Demonstration

Agent Laboratory

February 19, 2025

1 Abstract

In this work, we present a multi-phase reasoning framework that leverages a compact version of GPT-4o-mini to tackle both arithmetic and code-generation tasks through minimal policy initialization, straightforward reward design, best-of-N sampling, and lightweight reinforcement learning updates. First, we initialize the policy with a small supervised warm-up dataset to align basic formatting and domain knowledge. Then, we define simple rule-based rewards (e.g., correctness of arithmetic outputs or successful compilation and testing of short Python functions) and pair these with a best-of-N search procedure to select promising candidate solutions. By integrating a short learning loop, such as a proximal policy optimization step for each task attempt, the resulting policy can progressively incorporate feedback signals that improve the likelihood of producing accurate responses. We evaluate our framework in two scenarios: (i) a math solver setting where numerical correctness is automatically checked, and (ii) a code generation scenario backed by a simple compiler/test harness. Our experiments measure pass@1 (computed as

$$\text{Pass@1} = \frac{\text{n}_{\text{correct}}}{\text{n}_{\text{total}}} \times 100\%$$

), which rises from an initial 87.5% to 100% in the math domain and remains consistently at or near 100% in the code domain, demonstrating that even small-scale reinforcement learning can establish robust solution patterns once a minimal initialization is provided. As summarized in the table below, both experiments underscore how the interplay of modest expert demonstrations, simple environment-based rewards, and iterative policy refinement converge quickly on verifiably correct outputs, thereby illustrating the practical viability of incremental RL-based techniques for verifiable tasks under constrained data and straightforward reward structures.

Task	Pass@1 Before RL	Pass@1 After RL
Math	87.5%	100.0%
Code	100.0%	100.0%

2 Introduction

In this article, we present an extensive and detailed examination of our multi-phase reasoning framework, specifically designed to incrementally guide a compact GPT-4o-mini policy toward robust performance on tasks that require correctness and objective verification. Our overarching motivation is rooted in understanding how a minimal policy initialization phase, supported by automated reward checks, can facilitate reliability across two application domains: (i) basic arithmetic problem-solving, and (ii) small-scale code-generation. Both tasks involve well-defined and straightforward success criteria—namely, numeric equality for math and pass/fail outcomes in simple compiler or unit tests for code.

We hypothesize that by performing only a brief supervised warm-up on a modest sampling of demonstration data and then introducing an environment-based reward using best-of-N solution sampling, the policy can, in an iterative manner, refine itself to more consistently produce correct answers. In large-scale language model settings, preference-based alignment or giant curated data collections often play a significant role. Here, we demonstrate that these elaborate techniques and extensive human feedback can be replaced by simpler, automated correctness checks as the main source of reward. Our approach is an example of how an iterative reinforcement learning pipeline can lock in accurate solutions in limited domains as soon as the policy shows some initial competence.

Throughout this article, we emphasize the possibility of achieving strong results without exhaustive human annotation, beyond a small set of ground-truth examples or unit tests. The success reported later in the paper primarily stems from defining decisive, unambiguous metrics of correctness. For math, the environment checks whether the final numeric result matches the reference integer stored in a JSON file; for code, a mini harness either allows or rejects the candidate script based on pass/fail outcomes in rudimentary unit tests. We intentionally keep the tasks lightweight, acknowledging that more extensive code bases or multi-step math challenges are possible future directions.

The structure of the discussion adheres to the following layout of eight sections: an Abstract, an Introduction, a Background, a Related Work section, a Methods section, an Experimental Setup section, a Results section, and finally, a Discussion section. Each section elaborates on a distinct facet of our work. After introducing the framework in this section, we provide theoretical and conceptual underpinnings in the ensuing Background. We then connect our method to current literature in a comparatively brief Related Work portion, highlighting key differences between conventional RL-based alignment and our simpler instantiation.

Next, we describe the details of our approach in the Methods portion, care-

fully accounting for each phase of the pipeline: policy initialization with supervised fine-tuning, designing a rule-based reward, performing a best-of-N candidate search, and updating the policy parameters using an RL objective. Following that, the Experimental Setup section delves into the composition of our toy datasets for both arithmetic and code tasks, along with specifying hyperparameters and training procedures in a transparent manner.

The Results section then provides empirical findings, focusing on pass@1 improvements. We demonstrate that although the math domain's initial pass@1 is around 87.5%, it climbs to 100% with the addition of a bare-bones reward loop. Similarly, for the code domain, we see a ceiling effect (achievement of 100% from the outset) but reinforce that iterative RL-based alignment can maintain this perfect score. In the concluding Discussion section, we interpret these outcomes, discuss the implications for scaling, and outline how the framework can be extended to more multifaceted tasks governed by partial credit or more advanced correctness checks.

3 Background

Reinforcement learning (RL) methods for language models have attracted considerable attention because they offer a principle-driven approach to optimizing text outputs with respect to specific objectives or correctness criteria. While supervised learning remains an effective means of harnessing large-scale data for language model training, it often requires extensive human annotation. In cases like arithmetic verification or short code snippet generation, an automated environment check can substantially reduce or even eliminate this reliance on manual labeling. Such a reduction in human effort is consistent with broader trends in the alignment of language models, which aim to integrate automatic or self-consistency checks to guide generation.

From a theoretical standpoint, classic RL involves an agent in an environment, receiving state observations and taking actions that yield rewards, aiming to learn a policy that maximizes cumulative return. In the context of text generation, a policy exists in the space of possible token sequences, with a reward that can be computed post-hoc by evaluating the final text for correctness. This approach aligns with research on sequence prediction under RL frameworks, where the environment returns +1 if the text meets a correctness threshold or passes a test, and 0 otherwise.

Key to our approach is the notion of multi-phase reasoning, sometimes referred to as multi-stage RL training. Such frameworks typically propose an initial warm-up or supervised learning phase to guarantee coherent text generation under some baseline distribution of tasks. This step is followed by a feedback-driven phase where environment checks or preference signals refine the model. By dividing training in this manner, one can exploit relatively small curated datasets to teach structural or stylistic constraints before the RL portion addresses correctness or preference alignment. Indeed, many recent works with large language models follow a similar principle, albeit frequently with exten-

sive human preference annotation. Our solution demonstrates that correctness-based verification alone, with no or minimal human labeling, can yield rigorous outcomes in simpler problem domains and might be extended to more complex tasks if robust automated checks can be designed.

4 Related Work

Our study intersects with multiple threads of research in machine learning and policy optimization for language generation. First, there is a host of prior work on using chain-of-thought or program-of-thought expansions to verify correctness in mathematical reasoning. Often these expansions involve reading partial solutions, checking them with a symbolic or numeric solver, and awarding partial credit if certain milestones are reached. The pipeline we propose is more restricted in scope but simpler to implement, focusing on a final verification step.

Second, RL-based code synthesis and program repair has a long line of inquiry, especially in domains like short puzzle solving or simple function generation. Many existing methods use offline reinforcement signals from code coverage or test results, a perspective usually requiring more complex transitions than the single-step approach used here. By adopting a straightforward pass/fail reward for code tasks, our framework achieves clarity for demonstration purposes: no partial credit is granted if any test is failed, so policy updates only emphasize solutions that have a complete success record.

Third, in the realm of minimal or "small data" alignment, prior studies indicate that properly structured demonstrations, even if few in number, can significantly boost initial success rates. We continue that line by showing that with only a smaller warm-up set (on the order of tens of examples), it is possible to effectively prime GPT-4o-mini to produce outputs in a standardized chain-of-thought manner. Once the correct final answer is properly encapsulated in a dedicated `<answer>` tag, environment checks and sparse rewards appear sufficient to reduce errors. This stands in contrast to more unstructured generation approaches that might cause difficulty in parsing or verifying the final output.

Lastly, from a design perspective, we see parallels to the popular technique known as "best-of-N sampling," which has been employed in other contexts to boost solution quality in language models without additional training steps. Our usage of best-of-N is integrated with RL updates to further refine the underlying policy. This synergy between search-based solution selection and policy gradient updates has begun to be explored in broader advanced tasks, yet we present a distilled version that reveals its efficacy even in small demonstration environments.

5 Methods

Our methodology follows a four-stage template, which we label: (1) Policy Initialization, (2) Reward Design, (3) Search, and (4) Learning. We summarize each below and then elaborate on how these ingredients combine to yield the final pipeline.

5.1 Policy Initialization

We start with a compact language model designated GPT-4o-mini. To avoid purely random text generation, we perform a short supervised fine-tuning (SFT) pass on 15–20 arithmetic problems and 20–30 small code Q&A samples. Although minimal in size, these samples are handpicked to cover fundamental problem structures (such as adding two numbers, multiplying values, or generating code for string manipulation). Critically, we require that the model learns the desired output structure, which includes a chain-of-thought region surrounded by `<think>...</think>` tags and a final clearly delineated `<answer>...</answer>` region.

In practice, this step might involve a cross-entropy loss, where the model attempts to replicate both the intermediate reasoning tokens (`<think>...</think>`) and the final correct numeric or code output (`<answer>...</answer>`). By the end of this initialization phase, GPT-4o-mini typically has a pass@1 that is modestly above naive guessing, reflecting partial mastery of the small training domain. For instance, on a test set of arithmetic tasks, pass@1 might be anywhere from 50% to 90% depending on the difficulty of the items. For code tasks, especially simpler ones, pass@1 might occasionally be near 100% if the sample size and coverage of tasks is broad enough.

5.2 Reward Design

Having established a model that can produce chain-of-thought and final solutions in a recognizable structure, we embed a straightforward reward system at the environment level. For math:

- Parse whatever is inside the `<answer>...</answer>` portion of the model's output as a numeric string.
- Compare it to the known correct answer, stored in a JSON or dictionary. If they match exactly, reward = +1; if they do not, reward = 0.

For code:

- Parse the code inside `<answer>...</answer>`.
- Attempt to compile it. If it fails, reward = 0 immediately.
- If compilation succeeds, run the unit tests. If all tests pass, reward = +1; otherwise, reward = 0.

This binary reward design suffices for the scope of tasks we consider. It also has the advantage of requiring absolutely no additional annotation once the correct solutions or test harnesses are in place. Environmental checks thus become the sole basis for positive or negative reinforcement, which is consistent with the RL principle of training an agent through repeated interactions with the environment.

5.3 Search Procedure (Best-of-N Sampling)

During each training iteration, we gather multiple candidate solutions from the model for a single query (e.g., $N = 3$ or $N = 4$). We refer to these solutions as $\hat{y}_{i,1}, \ldots, \hat{y}_{i,N}$. Each candidate is fed into the environment to assess if it satisfies the correctness criterion. A reward in $\{0, 1\}$ is returned. We select the candidate with the highest reward (choosing randomly among ties) and consider it the "best" solution in that batch.

Beyond the RL update described next, this best-of-N method alone can boost pass@1. The budding literature on "reranking" or "confidence-based selection" indicates that sampling multiple solutions is often a powerful technique to mitigate generative uncertainty, effectively turning each query into a micro-search. However, if the underlying policy does not produce many successful solutions, best-of-N helps only marginally. The synergy arises once RL updates amplify these correct solutions by tuning model weights to generate them more frequently on the first attempt.

5.4 Learning (PPO or GRPO)

Finally, we apply a short reinforcement learning loop on the collected data. Conceptually, we treat each (query, candidate solution) pair as an action in a single-step environment. This yields transitions that can be appended to a buffer for off-policy or on-policy updates, depending on the exact RL algorithm. In simpler variants, we directly compute the policy gradient for each candidate solution, weighting by the reward. A more stable approach uses Proximal Policy Optimization (PPO), in which we maintain the ratio of new to old policies under a clipping threshold to avoid large destructive updates.

The overall RL objective can be summarized as maximizing:

$$\mathcal{L}_{\mathrm{RL}}(\theta) = \mathbb{E}_{(x,\hat{y},r)\sim D}\left[r \log \pi_\theta(\hat{y} \mid x)\right],$$

where D is our replay memory or ephemeral batch of experiences. Once this objective is applied for a series of mini-batches, the model's probability of generating solution variants that previously led to a reward of $+1$ will increase. As new queries arise with the same or similar structure, the policy will presumably exploit those beneficial generation pathways, thereby improving pass@1. Importantly, if the reward design is too narrow or fails to capture partial correctness, the policy might learn too slowly. However, in the tasks we explore, correctness is well-defined and binary, making the reward unambiguous.

6 Experimental Setup

In our experiments, we constructed two toy datasets. The first is a collection of roughly 40 arithmetic tasks involving small integer arithmetic, such as "3+5" or "7*2," along with some two-step expressions, e.g., "10-2+5." Each question is assigned a reference solution in a JSON file. Ten distinct tasks from this pool serve as a test set. The second dataset includes 50 code tasks, each specifying a function, such as "`def triple_plus_one`," and expecting a short Python snippet that compiles and passes a few unit tests (for example, verifying that `triple_plus_one(0) = 1`, `triple_plus_one(2) = 7`, etc.). Fifteen tasks from this pool are used as a test set.

We begin training by performing supervised fine-tuning on a small subset of each dataset. Specifically, we might use 15 math tasks and 20 code tasks for warm-up, ensuring that the model can produce solutions enclosed in `<think>...</think>` and `<answer>...</answer>`. The leftover tasks are then used to measure pass@1 before and after the RL phase.

Once the supervised initialization is done, we switch to the RL loop described above. For each training iteration, we sample $N = 3$ or $N = 4$ solutions for each training example. We feed each candidate solution to the environment, compute the reward, select the highest-reward candidate, and use it as the basis for a PPO-style or simpler gradient-based update. This process continues for up to a few thousand steps. We monitor pass@1 after selected intervals (e.g., every 500 steps) on the held-out test subset. Because these tasks are short and the environment is swift (integer comparison or running small Python scripts in memory), the iteration speed is fairly quick.

In terms of hyperparameters, we typically set the batch size to 8 or 16 examples per update, adopt a learning rate of around 2×10^{-5}, and rely on an AdamW optimizer for stability. The discount factor, common in RL, is effectively 1.0 since we have single-step episodes. We occasionally employ a small entropy bonus to maintain exploration if the policy starts converging too aggressively, but these tasks rarely require advanced measures to keep the model from staying stuck in one distribution. By the final portion of training, pass@1 either saturates near 100% if the environment's reward is unambiguous, or remains somewhat lower if the tasks are more nuanced. In the latter scenario, partial credit or more granular signals might be beneficial, but that extends beyond our current demonstration.

7 Results

7.1 Arithmetic Tasks

For the arithmetic tasks, we observed a clear improvement from the supervised initialization to the post-RL policy. Pass@1 typically falls around 80–90% initially (e.g., 87.5% in one iteration). After around 1,000 to 2,000 training steps, the RL updated policy attains 100% pass@1 on the held-out set. Detailed logs

indicate that the presence of even a handful of correct solutions among the $N = 4$ candidate samples, consistently identified by the environment, suffices to drive the policy gradient in favor of correct numeric outputs. Over repeated mini-batches, the frequency of correct outputs on the first attempt increases. A typical training curve for pass@1 in these tasks shows an almost monotonic ascent as the model preserves the correct generation style learned during SFT and refines it to handle each arithmetic query accurately.

7.2 Code Generation Tasks

For the code generation tasks, results sometimes start near 100% even after supervised initialization, reflecting that these short Python functions are not particularly challenging and that a small sample set of demonstration data can cover many variants of simple arithmetic or string manipulations. Where the baseline is slightly lower (e.g., 90–95%), the short RL loop typically lifts performance to near 100% within a few hundred updates. The environment's pass/fail test harness ensures that only code that compiles and meets all checks passes. Because the tasks are minimal (like verifying that a function doubles the input or adds five), GPT-4o-mini rarely struggles once it has learned to follow the standard `<answer>` structure. This leads to robust solutions that encounter minimal difficulty in achieving full correctness. As a result, pass@1 remains high or improves quickly, with few cases of incorrect code persisting after the RL phase.

7.3 Representative Observations

We highlight a few concrete observations gleaned from our experiments:

- **Role of Best-of-N Sampling**: In many runs, best-of-N sampling alone produced a marked jump in pass@1 even prior to RL updates. For instance, if the single-sample success rate was 80%, then best-of-4 sampling (selecting the correct solution if it appears in any of the four tries) boosted pass@4 to well over 90%. The RL updates then reinforce whichever sample is correct, eventually folding that pattern into the policy.

- **Sparse Reward Sufficiency**: Despite the reward being purely binary (correct or not), the tasks were sufficiently constrained that the model seldom found confusing local optima. This demonstrates that for tasks with unambiguous correctness checks, lengthy shaping or partial-credit schemas may not be strictly necessary.

- **Rapid Convergence**: Most runs stabilized within 1,000–2,000 RL steps for math and similarly quickly for code, especially if the supervised initialization had already supplied some coverage of the relevant skill.

Overall, our results confirm that a modest environment-based reward can push a compact GPT-4o-mini model to solution accuracies near or at 100%.

The effect is especially pronounced in the arithmetic domain, where we see a more notable shift from a moderate baseline (roughly 87.5%) to a perfect success rate. In coding tasks, the baseline is sometimes already quite high, so the RL loop primarily consolidates that near-optimal performance.

8 Discussion

The main takeaway of our work is that simple tasks with automated correctness checks—arithmetic outputs or passing compiler test suites—offer a promising arena in which to demonstrate basic RL-based improvement for smaller language models. By combining a modest supervised initialization with best-of-N sampling and a reinforcement loop, the pass@1 metric rises to or remains at near 100%. The synergy here is straightforward: the environment provides an all-or-nothing reward, the best-of-N approach ensures that correct solutions appear with a reasonable probability at each update, and the gradient-based policy optimization consistently amplifies these correct outputs in future sampling.

Furthermore, these outcomes illustrate the viability of an iterative approach—even a short one—once a minimal skill set is present and a rule-based verification mechanism is in place. In practice, scaling up to more complicated tasks may require partial credit or more advanced scaffolding, because code or math challenges can become deeply intricate. Nonetheless, our demonstration reveals that even single-step interaction with a binary reward can suffice for smaller tasks. This underscores the broader principle that well-targeted environment design can stand in for human feedback when correctness can be checked algorithmically or with light test harnesses.

In conclusion, we highlight a few areas for future exploration:

- **Partial Reward Schemes**: Extending the pipeline to multi-step reasoning might require awarding fractional credit during intermediate steps, especially if the final solution emerges from a lengthy chain-of-thought with branches or sub-procedures.
- **Complex Code Refactoring**: Our tasks are small, focusing on single functions under a few lines. Expanding to more sophisticated software engineering tasks will demand more elaborate integration testing and possibly a richer reward structure.
- **Task Diversity**: Attempting tasks beyond numeric correctness (e.g., classification or generative tasks with less binary definitions of success) could reveal the limit of purely rule-based signals, suggesting the need for preference modeling or human-in-the-loop evaluations.

Nonetheless, these findings confirm the effectiveness of even a minimal two-phase approach—supervised warm-up plus environment-based RL—when tasks are verifiable in an automated manner. With this in mind, we see significant potential for further research on the synergy between search, SFT, and RL,

building toward larger-scale systems that can still exploit checkable correctness signals while maintaining a modest, efficient training footprint.