LEAP: Lean Environment-Feedback via Adaptive Pruning for Code RL in GPU Kernel Generation

Tankun Li, Zhi Chen, Yaohua Tang

Moore Threads AI

tangyaohua28@gmail.com

ABSTRACT

Post-training large language models (LLMs) via reinforcement learning (RL) has significantly advanced code generation capabilities. To bypass the heavy memory footprint of critic networks, current state-of-the-art frameworks leverage critic-free paradigms like Group Relative Policy Optimization (GRPO) tied to rule-based verification sandboxes. However, applying these frameworks to low-level systems programming, such as CUDA kernel generation—presents severe challenges: binary pass/fail rewards introduce severe signal sparsity, while multi-turn environmental feedback loops suffer from prohibitive compilation latencies and reward dilution across trajectories. In this work, we introduce **LEAP** (**L**ean **E**nvironment-Feedback via **A**daptive **P**runing), a scalable and computationally efficient multi-turn RL framework optimized for low-level hardware accelerator alignment. LEAP features **Difficulty-Conditioned Pruning (DCP)**, a dynamic gating mechanism that adaptively cuts off simple and overly catastrophic tasks from multi-turn expansion, focusing resource-heavy compilation and hardware exploration exclusively on high-value, complex tasks. To fully operationalize these paths without manual hyperparameter engineering, we propose a **Rank-Based Reward** formulation. By deriving scale-free relative advantages from pairwise tournament outcomes within the GRPO rollout group, our method inherently penalizes token inefficiency on simple prompts while maximizing learning gradients on challenging distributions. Empirical evaluations show that LEAP achieves superior first-turn proficiency and robust multi-turn debugging resilience while converging faster than unpruned multi-turn baselines, establishing a practical paradigm for low-level code RL.

1 INTRODUCTION

Large language models (LLMs) are central to modern software development, driving extensive research into LLM-based coding assistance. To enhance performance, recent work focuses on large-scale datasets for pre-training Lozhkov et al. (2024) and supervised fine-tuning (SFT) Wei et al. (2023), evaluated against rigorous benchmarks Liu et al. (2023); Jimenez et al. (2024). Consequently, current state-of-the-art models deliver exceptional coding proficiency across diverse engineering tasks Yang et al. (2025a); Zeng et al. (2026).

Post-training models via Reinforcement Learning (RL) introduces memory bottlenecks. While Proximal Policy Optimization (PPO) Schulman et al. (2017) relies on a memory-intensive critic model, Group Relative Policy Optimization (GRPO) Guo et al. (2025) eliminates this overhead by calculating relative advantages among multiple sampled rollouts per prompt. Because code correctness is deterministically verifiable via sandboxed unit testing, rule-based GRPO variants optimizing for binary rewards have become the mainstream approach to advancing code generation capabilities Yu et al. (2026); Zheng et al. (2025); Cheng et al. (2026b). Despite its potential, rule-based RL suffers from reward sparsity, where binary feedback penalizes minor code errors as severely as catastrophic failures, prompting dense feedback mechanisms like token-level Cheng et al. (2026a) or step-level Hou et al. (2025) rewards that remain difficult to scale. Alternatively, leveraging stronger LLMs as judges offers softer, rubric-based grading Gunjal et al. (2025); Viswanathan et al. (2026), though this approach incurs significant computational costs and relies heavily on prompt engineering.

To establish a dense learning signal without external judging overhead, we turn to an alternative paradigm: multi-turn training with environmental feedback. Instead of merely penalizing failed samples, a multi-turn formulation leverages these feedback logs to provide richer iterative signals, closely mirroring how human engineers debug low-level systems code. While recent frameworks have explored multi-turn training on simplified academic benchmarks Ekbote et al. (2025); Jain et al. (2025a); Gehring et al. (2024), two major issues hinder their practical deployment:

Reward Dilution Across Trajectories: Typical multi-turn training evaluates the reward based solely on the final outcome and propagates it across the entire trajectory without differentiating per-turn performance. Consequently, a

rollout requiring multiple debugging iterations receives the same reward as a rollout that succeeds on the first attempt. This biases the model toward the final result, greatly degrading its early-turn pass rate.

Prohibitive Compilation and Hardware Latency: Multi-turn training is inherently time-consuming. Prior works typically formalize multi-turn generation as a tree-search over simple, interpreted languages (e.g., Python) that are incredibly fast to verify. In real-life production tasks such as from-scratch CUDA kernel generation, however, the overhead of sequential compilation, host-to-device memory allocation, and kernel execution introduces massive latency. Running dense, unpruned multi-turn rollouts under these conditions drastically slows down training iterations and makes large-scale code RL practically unviable.

In this work, we introduce **LEAP** (**L**ean **E**nvironment-Feedback via **A**daptive **P**runing), a scalable and efficient multi-turn RL framework designed specifically to address these hardware-level bottlenecks within a rule-based GRPO paradigm. Our framework is grounded in a simple intuition: harder coding tasks require richer learning signals, whereas simple tasks already provide sufficient feedback. We dynamically classify kernel difficulty based on the aggregate rollout pass rate. Simple problems, which exhibit high pass rates, naturally derive ample optimization signals from successful rollouts. Conversely, difficult problems suffer from a lack of successful trajectories. Leveraging this insight, LEAP employs a branching mechanism termed **Difficulty-Conditioned Pruning (DCP)**. DCP adaptively prunes simple problems from multi-turn expansion, bypassing redundant compilation cycles and focusing resource-heavy hardware exploration exclusively on complex tasks.

This adaptive pruning strategy yields three key advantages:

Computational Efficiency: By restricting multi-turn branching via DCP, LEAP drastically lowers training time, saving substantial compute resources and GPU cycles otherwise wasted on redundant compilation and sandbox execution of trivial tasks.

Balanced Optimization: The model learns to optimize early-turn accuracy on simpler kernels while gaining dense, multi-turn error-correction signals on complex ones, thereby preserving early-turn proficiency without sacrificing late-turn debugging capabilities.

Performance Retention: We empirically demonstrate that our pruning strategy maintains—and in some cases surpasses—the performance of dense, unpruned multi-turn baseline frameworks.

To fully operationalize these efficiency gains, LEAP introduces a non-parametric, rank-based advantage formulation that generalizes across multi-turn trajectories. Rather than manually tuning heuristic, scalar-based "magic numbers" for intermediary turns, our approach maps rollout outcomes directly to an ordinal hierarchy based on iteration efficiency. By deriving relative advantages strictly from pairwise win-loss distributions within the GRPO group, the resulting optimization signal becomes dynamically responsive to local prompt difficulty. For simpler prompts dominated by rapid success, the mathematical baseline shifts upward, penalizing excessive turns and driving the policy toward zero-shot efficiency. Conversely, for complex prompts characterized by low base pass rates, the framework naturally scales to reward incremental debugging breakthroughs, maximizing the gradient signal on hard samples. This formulation effectively mitigates reward dilution, enforces a strict temporal penalty that naturally minimizes inference tokens, and completely bypasses the instability of manual reward engineering.

As illustrated in Fig. 1, our method converges to similar performance $1.93\times$ faster than existing approaches and outperform them in the end. In summary, our key contributions are:

- We propose **LEAP**, an efficient and scalable multi-turn RL framework optimized for low-level GPU kernel generation that drastically reduces the compilation and execution overhead typical of vanilla multi-turn RL methods.
- We demonstrate through extensive experiments that our **Difficulty-Conditioned Pruning (DCP)** strategy does not degrade model performance; instead, it encourages the model to extract richer signals from hard samples while enhancing first-turn accuracy on simpler ones.
- We introduce a scale-free, rank-based rewarding strategy within the GRPO paradigm that replaces rigid scalar rewards with relative pairwise advantages. This formulation dynamically enforces a temporal efficiency bias—incentivizing faster solutions on simple tasks while maximizing learning signals on complex, low-pass-rate benchmarks.
- Our empirical results confirm that the LEAP framework achieves superior performance in both model capability and training efficiency for from-scratch CUDA kernel generation task and general coding task compared to existing baselines.

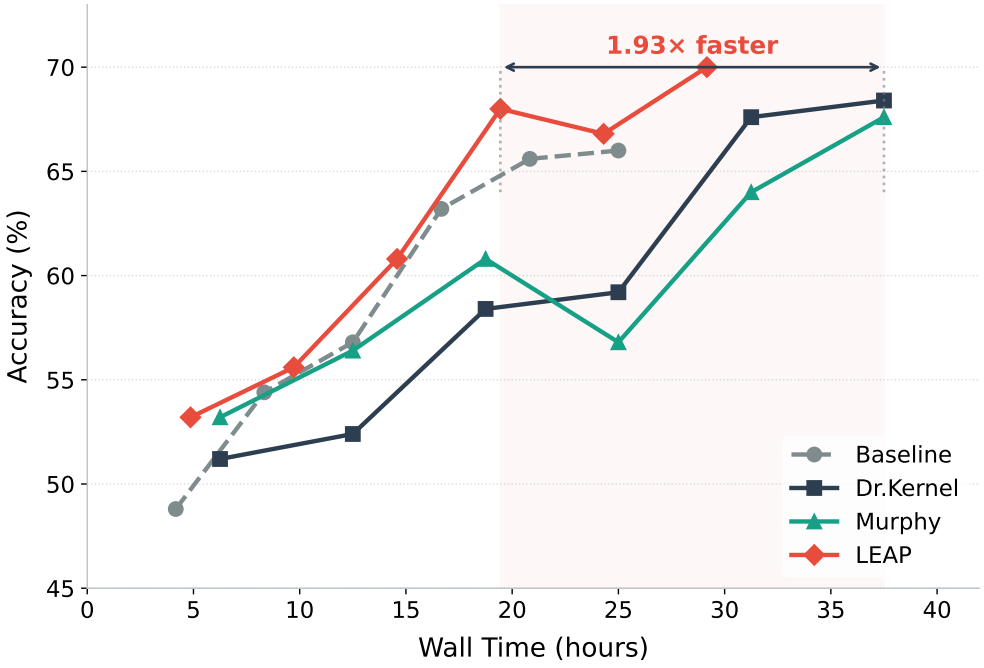

Figure 1: Test accuracy versus wall-clock training time. LEAP reaches the same accuracy roughly 1.93× faster than other works, and outperforms the Baseline at every wall-time budget.

2 Related Work

2.1 Code Reinforcement Learning for LLMs

Post-training large language models via Reinforcement Learning (RL) has significantly advanced complex reasoning. Early paradigms relied on Proximal Policy Optimization (PPO) Schulman et al. (2017) using heavy value networks or Process Reward Models (PRMs) to supply continuous feedback Yue et al. (2025); Cui et al. (2025), though at extreme computational and memory overhead. To eliminate critic-related bottlenecks, Group Relative Policy Optimization (GRPO) Guo et al. (2025) estimates relative advantages across localized rollouts via deterministic, verifiable rewards. Subsequent extensions like Dr.GRPO Liu et al. (2025), DAPO Yu et al. (2026), and related industry systems Xiao et al. (2026); Team et al. (2025) focus on mitigating GRPO's inherent reward sparsity and stabilizing group variance, while intermediate frameworks incorporate Monte Carlo tree search Hou et al. (2025); Yang et al. (2025b) or LLM-as-a-judge rubrics Gunjal et al. (2025); Huang et al. (2025); Viswanathan et al. (2026); DeepSeek AI (2026)—yet these incur prohibitive latency and API costs.

Crucially, existing methodologies remain ill-equipped for multi-turn RL over interactive horizons. Current multi-turn approaches like μCode Jain et al. (2025a), RLEF Gehring et al. (2024), and REVEAL Jin et al. (2025) still depend on costly verifier or critic networks, while critic-free alternatives such as MURPHY Ekbote et al. (2025) suffer from unpruned exploration spaces that fail in production. LEAP directly solves these multi-turn scalability bottlenecks by introducing dynamic task-conditioned pruning within a critic-free framework.

2.2 From-Scratch CUDA Kernel Generation

The vast majority of reinforcement learning frameworks for code synthesis rely on academic-centric training datasets like KodCode Xu et al. (2025) and evaluate performance on high-level benchmarks such as HumanEval Liu et al. (2023) and MBPP Austin et al. (2021), which focus on lightweight interpreted languages for which modern foundational models Yang et al. (2025a); Zeng et al. (2026) are already heavily optimized. To bridge this evaluative gap, our work targets from-scratch CUDA kernel generation. While KernelBench Ouyang et al. (2025) evaluates model proficiency in synthesizing CUDA code, its evaluation protocol lacks structural rigor, allowing models to artificially inflate pass rates by bypassing custom logic to invoke pre-existing PyTorch operators or optimized CUDA library wrappers. To enforce true architectural reasoning, we restrict our evaluation strictly to from-scratch kernel generation, strictly

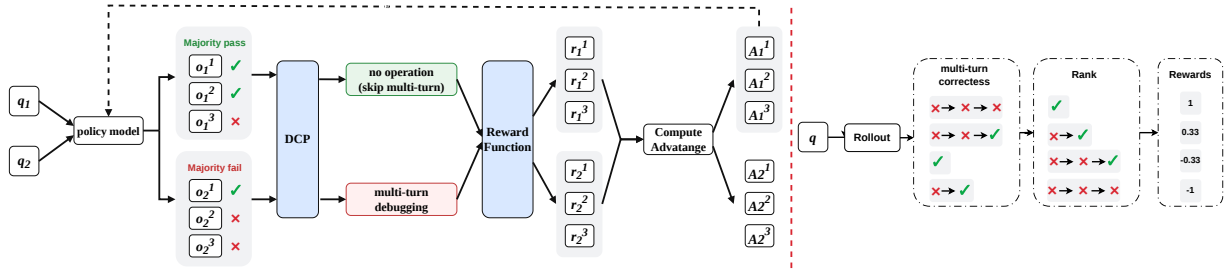

Figure 2: The overall structure of the LEAP (left) and rank-based reward (right). DCP refer to the difficulty conditioned pruning module. No-op stands for no operations (skip multi-turn debugging step). We skip the KL loss for simplicity.

prohibiting external native implementations. A few parallel works have begun exploring the intersection of LLMs and accelerator optimization, but each introduces distinct inefficiencies. CUDA-L1 Li et al. (2025) adopts a standard GRPO framework but tasks the model with a detailed text-based analysis prior to code generation, creating a verbose execution overhead. Dr.Kernel Liu et al. (2026) focuses on synthesizing Triton kernels utilizing the REINFORCE-with-leave-one-out (RLOO) framework Ahmadian et al. (2024), but penalizes failed compilation attempts with non-informative zero rewards and naively permits every rollout to enter the debugging phase, which severely exacerbates the sandbox compilation latency bottleneck. Meanwhile, CUDA Agent Dai et al. (2026) proposes an agentic framework centered on optimizing kernel speeds using a traditional PPO backend, exposing the model to high-overhead tool environments while relying on empirical stabilizing heuristics rather than structural algorithmic efficiency. Distinct from these approaches, LEAP bypasses the hardware and compilation bottlenecks that restrict existing frameworks by introducing an efficient, critic-free multi-turn RL architecture tailored for low-level systems alignment.

3 LEAP

Our framework is largely based on the standard GRPO baseline. The over all flow is illustrated in figure 2. The differences between this work and standard GRPO is shown in the coloured box – DCP and Reward Function. DCP act as a gating mechanism, controlling which groups should enter multi-turn debug step and the proposed reward function further balance the first-turn generation quality with iterative debugging capabilities. The details are explained in the following sections.

3.1 Difficulty-Conditioned Pruning (DCP)

The Difficulty-Conditioned Pruning (DCP) module stems from an intuitive assumption regarding task complexity in Reinforcement Learning (RL): training distributions naturally comprise simpler tasks with high empirical pass rates and harder tasks with low pass rates. While simpler questions inherently enjoy dense learning signals derived from numerous successful optimization paths, more difficult questions suffer from sparse feedback, hindering gradual policy improvement. Under these conditions, uniformly applying multi-turn debugging across all tasks introduces two distinct negative effects: (1) for simpler tasks, errors in initial turns are insufficiently penalized, causing the model to over-rely on multi-turn corrections rather than converging on immediate correctness; and (2) it imposes a prohibitive computational burden on the training pipeline. To resolve these inefficiencies, we introduce DCP to selectively gate which questions transition into the multi-turn debugging environment based on task difficulty.

Let q be a question sampled from the dataset distribution $P(Q)$, and let $\{o_i\}_{i=1}^{G}$ be a group of G independent candidate responses sampled from the current policy $\pi_\theta(\cdot \mid q)$ within a group-based reinforcement learning framework. To quantify the empirical difficulty of q at the current training step, DCP measures the cohort failure rate. Let $\mathbb{I}(\cdot)$ be an indicator function that returns 1 if the response fails execution verification and 0 if it passes. The total number of failed trajectories within a sampled group is defined as:

$$N_{\text{fail}}(q) = \sum_{i=1}^{G} \mathbb{I}(\text{Response } o_i \text{ failed}) \tag{1}$$

DCP introduces a conditional gating function $\mathcal{G}(q) \in \{0, 1\}$ that dynamically determines the computational path for the entire group:

$$\mathcal{G}(q) = \begin{cases} 1, & \text{if } \tau_{\text{min}} \leq N_{\text{fail}}(q) \leq \tau_{\text{max}} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

where τ_{min} and τ_{max} are strict hyperparameters defining the optimal learning zone for multi-turn optimization. When a question exhibits a high pass rate ($N_{\text{fail}}(q) < \tau_{\text{min}}$), it is classified as simple and bypasses the debugging loop ($\mathcal{G}(q) = 0$). This forces the model to learn from single-turn penalties, preventing it from developing a reliance on multi-turn corrections for easy tasks. Conversely, when a question exhibits a low pass rate ($N_{\text{fail}}(q) > \tau_{\text{max}}$), it is classified as overly difficult and the current model is hopeless to solve it. It similarly bypasses the debugging loop ($\mathcal{G}(q) = 0$) to protect compute resources from being wasted on unrecoverable trajectories. Only when the group failure count falls within the bounded threshold does $\mathcal{G}(q) = 1$, routing the group into the multi-turn debugging environment.

This high-level routing logic alters the standard GRPO objective by conditioning the optimization path on $\mathcal{G}(q)$. The objective function selectively scales the sequence length and computational layout based on this gate:

$$\begin{aligned} L_{\text{DCP}}(\theta) = &\mathbb{E}\left[q \sim P(Q), \{o_i\}_{i=1}^{G} \sim \pi_\theta\right] \\ &\left[\frac{1}{G}\sum_{i=1}^{G} L_{\text{surr}}\left(\theta, o_i \mid \mathcal{G}(q)\right) - \beta D_{\text{KL}}(\pi_\theta \parallel \pi_{\text{ref}})\right] \end{aligned} \tag{3}$$

where the surrogate loss L_{surr} evaluates a standard single-turn sequence when $\mathcal{G}(q) = 0$, and dynamically expands to evaluate a concatenated, multi-turn debugging sequence when $\mathcal{G}(q) = 1$.

3.2 Rank-Based Reward

Empirically, we observed that varying the scalar reward assignments for multi-turn trajectories yields highly divergent optimization outcomes, requiring extensive and costly ablation studies to discover an optimal configuration. To fully operationalize the training paths established by the DCP module without introducing the optimization instability of hand-tuned heuristic values, LEAP introduces a non-parametric, rank-based reward framework coupled with GRPO. Instead of assigning continuous scalar "magic numbers" to different rollouts, our approach derives optimization gradients by first computing a baseline-free reward from the relative pairwise win-loss distribution within each rollout group, mapping outcomes directly to an execution efficiency hierarchy.

Depending on the gating state $\mathcal{G}(q)$, candidate responses enter an execution environment allowed up to M maximum turns. We rank all the rollouts in the same group according to:

$$\mathcal{O}_{\text{1-st turn pass}} \succ \mathcal{O}_{\text{2-nd turn pass}} \succ \cdots \succ \mathcal{O}_{\text{failure}} \tag{4}$$

where $\succ$ denotes strict preference. That is, the rollouts use less turns to succeed are preferred over those use more turns. Specifically, we assign discrete rank values $r(o_i) \in \mathbb{N}$. A response that achieves successful verification on the first turn receives the highest rank $r = M$, a successful correction on the m-th turn receives $r = M - m + 1$, and a persistent execution or compilation failure at the end of the trajectory maps to the lowest rank $r = 0$.

For any candidate output o_i within a group of size G, we calculate its baseline-free reward R_i by performing a complete pairwise tournament against its $G - 1$ local peers. We define the count of strictly worse peers $N_{\text{worse}}(o_i)$ and strictly better peers $N_{\text{better}}(o_i)$ as:

$$N_{\text{worse}}(o_i) = \sum_{j \neq i}^{G} \mathbb{I}\Big(r(o_i) > r(o_j)\Big), \quad N_{\text{better}}(o_i) = \sum_{j \neq i}^{G} \mathbb{I}\Big(r(o_i) < r(o_j)\Big) \tag{5}$$

The pairwise tournament reward R_i is computed as the net peer matchup score, normalized exclusively by the competing peer cohort size $G - 1$:

$$R_i = \frac{N_{\text{worse}}(o_i) - N_{\text{better}}(o_i)}{G - 1} \tag{6}$$

where the calculated reward R_i is bounded strictly to the closed interval $[-1.0, 1.0]$. This normalization guarantees that an absolute winner within the rollout group receives a reward of exactly $+1.0$, while an absolute loser is anchored firmly to -1.0.

Finally, to optimize the policy, we compute the final advantage A_i applied to the surrogate loss $\mathcal{L}_{\text{surr}}$ using the standard GRPO formulation.

The design of our rank-based framework deliberately eliminates reward hyperparameter tuning overhead. Compared to rigid, hand-tuned heuristic reward functions, our method offers a streamlined alternative that achieves two critical objectives: it inherently incentivizes execution efficiency by penalizing unnecessary reasoning turns, and it self-adaptively adjusts relative reward scaling based on the empirical difficulty of the problem context. More detailed analysis can be seen in Supplement.

3.3 Discounted Per-Turn Advantage Assignment

Given a rollout-level advantage A_i for rollout i computed as above-mentioned, we assign token-level advantages according to the turn structure of the generated trajectory. Let the trajectory contain T_i model-generated turns, indexed by $t \in \{0, 1, \ldots, T_i - 1\}$. Let $\mathcal{M}_{i,t}$ denote the set of response tokens in rollout i that belong to generated turn t. Tokens outside all generated turns, such as padding, prompt tokens, or environment feedback tokens, are excluded from optimization and receive zero advantage.

For each rollout, we first choose a discount base according to the sign of its rollout-level advantage, where $\gamma \in (0, 1]$:

$$\beta_i = \begin{cases} \gamma, & A_i \geq 0, \\ 1, & A_i < 0. \end{cases}$$

The advantage assigned to every token in turn t is then the rollout-level advantage multiplied by the turn discount:

$$\hat{A}_{i,k} = \beta_i^{T_i - t - 1} A_i, \qquad k \in \mathcal{M}_{i,t}.$$

All tokens in the same generated turn therefore share the same advantage. The last generated turn has exponent zero and receives the original rollout-level advantage A_i. Earlier turns receive progressively smaller positive advantages when $A_i \geq 0$. When $A_i < 0$, the discount base is 1, so every generated turn receives the same negative advantage A_i.

This assignment keeps the rollout-level preference signal unchanged while distributing it across turns. Positive trajectories place more credit on later turns, which are often closer to the final successful repair in a debugging trajectory. Negative trajectories apply the same penalty to all generated turns, discouraging the full sequence of actions that led to the unfavorable outcome.

3.4 Analysis of Adaptive Reward Scaling

To systematically demonstrate the self-adaptive properties of the rank-based reward framework against static heuristic assignments, we analyze a restricted three-tier trajectory setting. Let the maximum number of turns be bounded to $M = 2$, naturally segmenting the trajectory outcome space into three distinct tiers: first-turn success ($\mathcal{O}_1$), second-turn success ($\mathcal{O}_2$), and failure ($\mathcal{O}_3$).

Let N_1, N_2, and N_3 denote the frequency of responses landing in $\mathcal{O}_1, \mathcal{O}_2$, and $\mathcal{O}_3$ respectively within a rollout group of size G, such that $G = N_1 + N_2 + N_3$. Under the proposed pairwise tournament framework, the baseline-free reward R_i for an individual response o_i in each respective tier is formulated as:

$$R(\mathcal{O}_1) = \frac{N_2 + N_3}{G - 1} \tag{7}$$

$$R(\mathcal{O}_2) = \frac{N_3 - N_1}{G - 1} \tag{8}$$

$$R(\mathcal{O}_3) = \frac{-(N_1 + N_2)}{G - 1} \tag{9}$$

The critical distinction between our non-parametric framework and standard hand-tuned rewards lies in the mathematical behavior of the intermediate tier, $R(\mathcal{O}_2)$. In a traditional reinforcement learning setup, intermediate successes are typically assigned a static scalar hyperparameter $c \in (0, 1)$ (e.g., $R_{\text{static}}(\mathcal{O}_2) = c$). This static assignment fundamentally assumes that the value of a multi-turn correction is independent of the problem's inherent difficulty.

Conversely, our rank-based formulation in Equation 8 reveals that $R(\mathcal{O}_2)$ is strictly a function of the divergence between the failure density N_3 and the optimal success density N_1. This yields a mathematically rigorous adaptive curriculum that scales dynamically with empirical prompt difficulty:

Regime 1: Low-Difficulty Contexts ($N_1 > N_3$). When the problem is easy, the empirical distribution skews heavily toward first-turn successes. Consequently, the intermediate reward satisfies:

$$\lim_{N_1 \gg N_3} R(\mathcal{O}_2) < 0 \tag{10}$$

In this regime, requiring a second turn indicates relative execution inefficiency. The rank-based framework automatically penalizes $\mathcal{O}_2$, driving the policy gradient toward optimal first-turn efficiency.

Regime 2: High-Difficulty Contexts ($N_3 > N_1$). When the problem is exceedingly difficult, the empirical distribution is dominated by compilation or execution failures. Under these conditions, the intermediate reward satisfies:

$$\lim_{N_3 \gg N_1} R(\mathcal{O}_2) > 0 \tag{11}$$

Because first-turn successes are rare ($N_1 \to 0$), successfully correcting an error on the second turn represents a significant statistical achievement relative to the peer cohort. The rank-based reward dynamically scales toward $+1.0$, heavily rewarding the correction capability and preventing gradient collapse.

In summary, because the partial derivatives satisfy $\frac{\partial R(\mathcal{O}_2)}{\partial N_3} > 0$ and $\frac{\partial R(\mathcal{O}_2)}{\partial N_1} < 0$, the intermediate reward acts as a continuous, baseline-free interpolator on $[-1, 1]$. By mathematically coupling the reward of partial successes to the underlying outcome distribution of the peer cohort, LEAP wholly eliminates the optimization instability and hyperparameter search space intrinsic to static, hand-tuned heuristic values.

4 Experiments

4.1 Implementation Details

We conduct experiments based on VERL Sheng et al. (2024). The experiments, including ablation studies, are mainly performed on CUDA generation task. For the sake of reproducibility, we also report the performance in general coding task using public data. For from-scratch CUDA kernel generation task, we perform cold-start training on pytorch-to-CUDA dataset Cheng et al. (2026c) to establish a foundational knowledge base in CUDA programming. Subsequently, we apply Reinforcement Learning (RL) training using the CUDA-Agent dataset Dai et al. (2026), validating the model within a custom-built sandbox that provides detailed execution tracebacks. For benchmarking, we evaluate our approach on KernelBench Ouyang et al. (2025). For general coding task, we follow KodCode implementation Xu et al. (2025) and perform RL with Qwen2.5-7B on KodCode dataset. The performances are reported with KodCode test set and LiveCodeBench Jain et al. (2025b).

When comparing with existing works, we choose standard GRPO as a baseline and compare our work with MURPHY Ekbote et al. (2025) as well as Dr.Kernel Liu et al. (2026), two GRPO-based multi-turn training frameworks. For MURPHY implementation, it is infeasible to perform a complete tree-search for multi-turn training in practice due to massive sandbox verification cost, we restrict the leaf node branching to one. For more implementation details, please refer to Supplement.

During RL training, the hyperparameters are configured with a learning rate of 1×10^{-6}, a weight decay of 0.1, a batch size of 32, and 8 rollouts per prompt. For the generation settings, the sampling temperature, top_k, and top_p are set to 1, -1, and 0.95 during rollout, and 0.7, 20, and 0.95 during validation, respectively. Because our training framework mixes single-turn and multi-turn data, the variance in training sequence lengths can be quite high. To prevent lengthy multi-turn trajectories from disproportionately dominating each gradient update, we adopt the sample-level loss objective proposed in DAPO Yu et al. (2026). For a fair comparison, all baseline experiments are conducted under this identical setting.

We perform all the training on a server with 8 Nvidia B200 GPUs and the verification sandbox for CUDA kernel is deployed on two server with 8 Nvidia A100 GPUs. For Kodcode-style training, we follow the official implementation carefully.

4.2 Main Results

4.2.1 KernelBench

The empirical evaluation of LEAP against existing baselines is summarized in Table 1. Models are evaluated across a maximum of three turns, with cumulative accuracy reported at each iteration. Notably, LEAP achieves the highest first-turn accuracy among all tested frameworks. Both LEAP and Dr.Kernel consistently outperform the remaining baselines. However, compared to Dr.Kernel, LEAP yields superior results in the first turn while maintaining highly competitive performance in subsequent turns, demonstrating its ability to preserve zero-shot proficiency without sacrificing multi-turn error correction.

Table 1: Model performance across methods and difficulty levels on KernelBench

Level	Turn	Baseline	Murphy	Dr.Kernel	LEAP
Level 1 (%)	1	80	79	**84**	**84**
	2	91	92	**95**	93
	3	91	93	**96**	95
Level 2 (%)	1	76	**78**	75	76
	2	85	82	**88**	**88**
	3	85	83	**89**	**89**
Level 3 (%)	1	18	24	24	**30**
	2	24	**34**	32	**34**
	3	24	34	32	**36**
Overall (%)	1	66	67.6	68.4	**70**
	2	75.2	76.4	**79.6**	79.2
	3	75.2	77.2	80.4	**80.8**

Table 2: Training cost and turn-efficiency of different methods.

Method	Baseline	Murphy	Dr.Kernel	LEAP
Step Time (s)	$\sim$600	$\sim$950	$\sim$950	$\sim$700
Turn Per Pass	1.90	1.93	1.84	1.77

Beyond model capabilities, LEAP delivers substantial computational savings. As detailed in Table 2, our framework requires less training time than competing methods. Using first-turn accuracy as a metric to measure model convergence, we plot accuracy versus wall-clock time in Fig. 1. Notably, LEAP demonstrates wall-clock efficiency, reaching comparable convergence $1.93\times$ faster than alternative methods. To report "How many total turns did this run spend for each successful sample it got?", the turn-efficiency is also reported in Table 2, which is computed by (total turns used by all samples) / (passed samples). LEAP demonstrate the best turn-efficiency among all the peers, using less turns on average to solve the problems.

Taken together, these results validate that LEAP effectively balances optimization across varying problem difficulties—safeguarding first-turn performance, retaining robust multi-turn debugging capabilities, and drastically accelerating training throughput.

4.2.2 KODCODE

We also conducted evaluation experiments on the KodCode dataset Xu et al. (2025), with the corresponding results presented in Table 3. Because KodCode is a relatively straightforward, Python-centric dataset, performance across subsequent iterations (Turns 2 and 3) remains highly comparable. Notably, the primary advantage of LEAP lies in its superior first-turn accuracy, where it achieves the top performance. To further illustrate this advantage, we plot the first-turn accuracy against the training steps in Figure 3. As the curves demonstrate, LEAP consistently outperforms competing methods throughout the entire training duration.

4.2.3 LIVECODEBENCH

After training on KodCode dataset, we follow the previous work to evaluate the model on LiveCodeBench in Table 4. LEAP demonstrate the best overall performance among peers, which suggest LEAP's potential to be applied to general coding tasks.

Table 3: Performance comparison across different methods on KodCode test set

Turn	Murphy	Dr.Kernel	LEAP
1	88.5	88.3	**90.2**
2	94.5	94.1	**94.7**
3	**96.7**	96.5	**96.7**

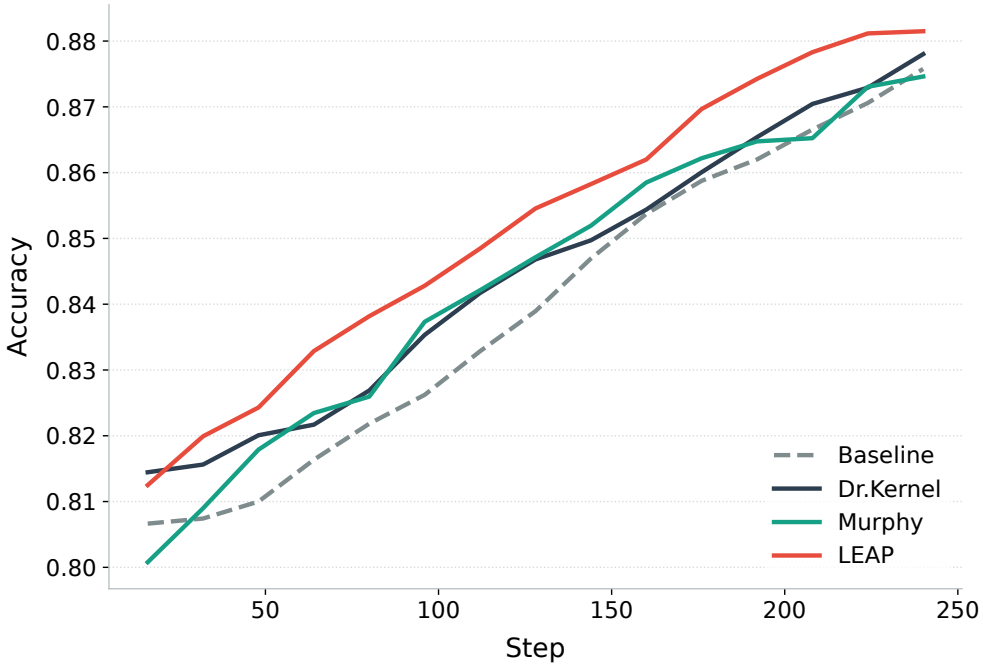

Figure 3: The first turn accuracy v.s. training steps on KodCode test set. The values are smoothed with factor 0.8.

4.3 Study on Branch Pruning

In this section, we analyze the impact of different pruning ranges within the DCP module, with the experimental results summarized in Table 5. The "Pruning Fail Range" specifies the pass-rate thresholds that dictate whether a rollout group enters the multi-turn debugging phase. Specifically, $\leq$50% targets less challenging problems (where the initial pass-rate is $\geq$50%), while $\geq$50% isolates more difficult problems (where the initial pass-rate is $\leq$50%). The "Full" setting denotes a baseline configuration where all problems bypass pruning and proceed directly to the multi-turn phase.

To evaluate these dynamics, we track several key metrics during training. "Activated groups" represents the percentage of total rollout groups that trigger the multi-turn debugging phase. "Per-rollout Recovery Rate" measures the proportion of individual multi-turn trajectories that successfully recover from an initial execution failure, whereas "Per-group Recovery Rate" denotes the ratio of groups that yield at least one successfully debugged trajectory.

Our findings reveal that deploying the multi-turn debugging phase exclusively on harder problems ($\geq$50%) yields superior final accuracy compared to focusing on easier problems ($\leq$50%), despite the fact that easier problems inherently exhibit higher baseline recovery rates. This strongly validates our core hypothesis: for code generation tasks, simpler problems already benefit from dense learning signals provided by an abundance of successful initial trajectories. Conversely, more difficult problems suffer from sparse signals. By selectively routing only the harder problems into the multi-turn debugging pipeline, the model encounters a significantly more effective learning gradient, maximizing performance gains while avoiding the prohibitive training costs associated with the "Full" multi-turn configuration.

Table 4: Performance comparison across different methods on LiveCodeBench

Method	**Baseline**	**Murphy**	**Drkernel**	**LEAP**
Easy Pass@1	0.609	0.614	0.609	**0.619**
Medium Pass@1	**0.088**	0.085	0.085	0.085
Hard Pass@1	0.007	0.010	0	**0.017**
Pass@1	0.174	0.181	0.175	**0.185**
Easy Pass@5	0.744	0.767	0.767	**0.814**
Medium Pass@5	0.173	**0.192**	0.153	0.173
Hard Pass@5	0.038	**0.05**	0.041	**0.05**
Pass@5	0.251	0.269	0.234	**0.274**

Table 5: Impact of pruning range on performance

Metric	**Baseline**	$\leq$ **50%**	$\geq$ **50%**	**Full**
Acc@3-turns	75.2	77.6	80.8	80.4
Activated groups	–	57.1%	43.9%	42.3%
Per-rollout Recovery	–	43.3%	20.5%	26.4%
Per-group Recovery	–	73.9%	61.8%	64.5%

4.4 Study on Reward functions

In this section, we isolate and analyze the specific impact of our non-parametric, rank-based advantage formulation. As formalized in Section 3.2 and further analyzed in Supplement, this rewarding paradigm dynamically scales advantages based on local prompt difficulty, naturally regularizing the policy to prioritize minimal-turn solutions. In the table, GRPO-MT stands for regular GRPO multi-turn training with standard advantages computation.

The empirical evidence for this behavior is presented in Table 6. Compared to standard GRPO loss, which relies on rigid, static scalar rewards, our rank-based approach yields increases in each turn accuracy. This gap demonstrates that the dynamically shifting baseline successfully penalize verbose or inefficient trajectories on simpler tasks while preserve the gradient signal required for incremental debugging on complex ones. Specifically, in the early training where the model is under-performing, the multi-turn trajectories provide rich signals and it is rewarded heavily, whereas in the later stages where the model is performing well, the redundant multi-turn trajectories will be punished for those easier tasks.

Consequently, these findings confirm that transitioning from heuristic scalar rewards to relative pairwise advantages is essential for aligning multi-turn reinforcement learning with inference-token efficiency, effectively mitigating the trade-off between zero-shot proficiency and multi-turn resilience.

4.5 Training Observation

Figure 4 presents the rollout-state composition of effective training samples under GRPO multi-turn and LEAP. Rollouts are grouped by problem instance and training step, and their advantages are computed from the corresponding reward definitions. The rollouts are bucketed by observed pass rate and categorized by outcome type: immediate pass, debug-assisted pass, or failure, with the sign of the advantage indicated in the legend. The y-axis reports the percentage composition of nonzero-advantage rollouts within each pass-rate bucket. The x-axis denotes the observed pass rate of the rollout group, used as a proxy for sample difficulty: lower pass rates correspond to harder samples, while higher pass rates correspond to easier samples. As shown in the figure, easier problems already have higher first-turn pass rates and therefore receive dense learning signals from immediate successes, while the additional signal contributed by multi-turn recoveries is relatively small. This causes GRPO-MT to place a disproportionate amount of effective

Table 6: Rank-based reward function comparison

Setting	Turn	GRPO-MT	Rank-based
Level 1 (%)	1	83	**84**
	2	91	**93**
	3	91	**95**
Level 2 (%)	1	**79**	76
	2	86	**88**
	3	**89**	**89**
Level 3 (%)	1	26	**30**
	2	32	**34**
	3	**36**	**36**
Overall (%)	1	**70.0**	**70.0**
	2	77.2	**79.2**
	3	79.2	**80.8**

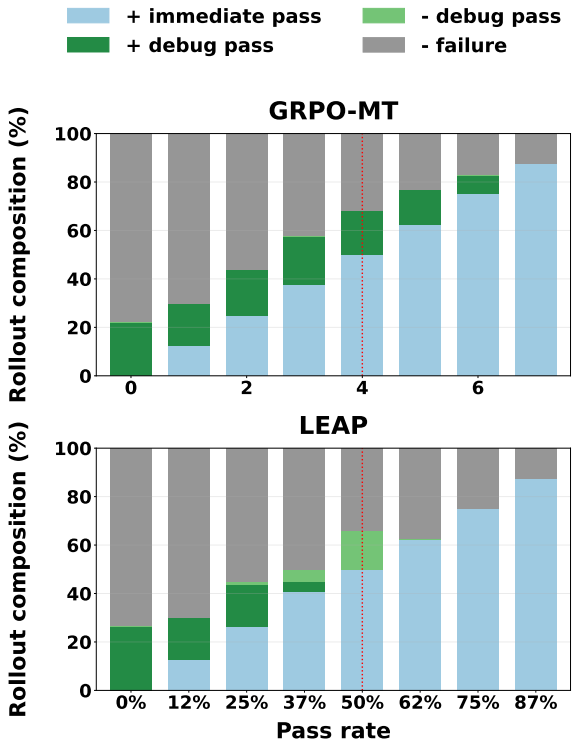

Figure 4: The rollout composition during training of LEAP and GRPO-MT.

training signal on easier problems. In contrast, LEAP uses the DCP mechanism to gate easier problems from entering additional debugging turns, leading to a more balanced distribution of learning signal across difficulty levels. This more targeted allocation of training signal helps LEAP converge faster and reach a better optimum.

5 Conclusion

In this work, we presented LEAP, an efficient, critic-free multi-turn reinforcement learning framework designed specifically to overcome the computational and architectural bottlenecks of from-scratch CUDA kernel generation. By introducing Difficulty-Conditioned Pruning (DCP), we successfully converted the traditionally expensive multi-turn tree-search into an adaptive execution path, isolating resource-intensive compilation and hardware sandboxing onto tasks

where dense signals are critically required. Furthermore, our non-parametric, rank-based reward framework eliminates the unstable search space of hand-tuned scalar rewards. Through a localized pairwise tournament scheme within the GRPO paradigm, the optimization objective self-adaptively shifts its baseline to penalize sub-optimal token efficiency on simple tasks while amplifying sparse learning gradients on complex problems. Our empirical results on low-level systems benchmarks demonstrate that LEAP significantly mitigates the traditional trade-offs of multi-turn code RL, preserving exceptional first-turn proficiency while retaining robust error-correction capabilities. By drastically reducing runtime latencies without sacrificing alignment quality, LEAP establishes a highly scalable and viable paradigm for post-training LLMs in specialized, computationally intensive hardware optimization domains.

REFERENCES

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting reinforce-style optimization for learning from human feedback in llms. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12248–12267, 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Daixuan Cheng, Shaohan Huang, Xuekai Zhu, Bo Dai, Xin Zhao, Zhenliang Zhang, and Furu Wei. Reasoning with exploration: An entropy perspective. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 30377–30385, 2026a.

Jorge Zhoujun Cheng, Shibo Hao, Tianyang Liu, Fan Zhou, Yutao Xie, Feng Yao, Yuexin Bian, Nilabjo Dey, Yonghao Zhuang, Yuheng Zha, et al. Revisiting reinforcement learning for llm reasoning from a cross-domain perspective. *Advances in Neural Information Processing Systems*, 38, 2026b.

Kun Cheng, Songshuo Lu, Sicong Liao, Tankun Li, Yafei Zhang, Dong Yang, Qiheng Lv, Hua Wang, Zhi Chen, and Yaohua Tang. Musacoder: Native gpu kernel generation with full-stack training on moore threads gpu. *arXiv preprint arXiv:2606.04847*, 2026c.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Yuchen Zhang, Jiacheng Chen, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, et al. Process reinforcement through implicit rewards. *arXiv preprint arXiv:2502.01456*, 2025.

Weinan Dai, Hanlin Wu, Qiying Yu, Huan-ang Gao, Jiahao Li, Chengquan Jiang, Weiqiang Lou, Yufan Song, Hongli Yu, Jiaze Chen, et al. Cuda agent: Large-scale agentic rl for high-performance cuda kernel generation. *arXiv preprint arXiv:2602.24286*, 2026.

DeepSeek AI. DeepSeek API News Update. https://api-docs.deepseek.com/news/news260424, 2026. Accessed: 2026-05-26.

Chanakya Ekbote, Vijay Lingam, Behrooz Omidvar Tehrani, Jun Huan, Sujay Sanghavi, Anoop Deoras, and Stefano Soatto. Murphy: Reflective multi-turn reinforcement learning for self-correcting code generation in large language. In *First Workshop on Foundations of Reasoning in Language Models*, 2025. URL https://openreview.net/forum?id=x0Ir7cWEiA.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains. *arXiv preprint arXiv:2507.17746*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081): 633–638, 2025.

Zhenyu Hou, Ziniu Hu, Yujiang Li, Rui Lu, Jie Tang, and Yuxiao Dong. Treerl: Llm reinforcement learning with on-policy tree search. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12355–12369, 2025.

Zenan Huang, Yihong Zhuang, Guoshan Lu, Zeyu Qin, Haokai Xu, Tianyu Zhao, Ru Peng, Jiaqi Hu, Zhanming Shen, Xiaomeng Hu, Xijun Gu, Peiyi Tu, Jiaxin Liu, Wenyu Chen, Yuzhuo Fu, Zhiting Fan, Yanmei Gu, Yuanyuan Wang, Zhengkai Yang, Jianguo Li, and Junbo Zhao. Reinforcement learning with rubric anchors, 2025. URL https://arxiv.org/abs/2508.12790.

Arnav Kumar Jain, Gonzalo Gonzalez-Pumariega, Wayne Chen, Alexander M Rush, Wenting Zhao, and Sanjiban Choudhury. Multi-turn code generation through single-step rewards. *arXiv preprint arXiv:2502.20380*, 2025a.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pages 58791–58831, 2025b.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024.

Yiyang Jin, Kunzhao Xu, Hang Li, Xueting Han, Yanmin Zhou, Cheng Li, and Jing Bai. Reveal: Self-evolving code agents via reliable self-verification. *arXiv preprint arXiv:2506.11442*, 2025.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving cuda optimization via contrastive reinforcement learning. *arXiv preprint arXiv:2507.14111*, 2025.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in neural information processing systems*, 36:21558–21572, 2023.

Wei Liu, Jiawei Xu, Yingru Li, Longtao Zheng, Tianjian Li, Qian Liu, and Junxian He. Dr. kernel: Reinforcement learning done right for triton kernel generations. *arXiv preprint arXiv:2602.05885*, 2026.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels? *arXiv preprint arXiv:2502.10517*, 2025.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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

Kimi Team, Yifan Bai, Yiping Bao, Y Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Vijay Viswanathan, Yanchao Sun, Xiang Kong, Meng Cao, Graham Neubig, and Sherry Wu. Checklists are better than reward models for aligning language models. *Advances in Neural Information Processing Systems*, 38:114728–114754, 2026.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. *arXiv preprint arXiv:2312.02120*, 2023.

Bangjun Xiao, Bingquan Xia, Bo Yang, Bofei Gao, Bowen Shen, Chen Zhang, Chenhong He, Chiheng Lou, Fuli Luo, Gang Wang, et al. Mimo-v2-flash technical report. *arXiv preprint arXiv:2601.02780*, 2026.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. Kodcode: A diverse, challenging, and verifiable synthetic dataset for coding. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 6980–7008, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

Zhicheng Yang, Zhijiang Guo, Yinya Huang, Xiaodan Liang, Yiwei Wang, and Jing Tang. Treerpo: Tree relative policy optimization. *arXiv preprint arXiv:2506.05183*, 2025b.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2026.

Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, T Fan, Z Du, et al. Vapo: Efficient and reliable reinforcement learning for advanced reasoning tasks. *URL https://arxiv.org/abs/2504.05118*, 2025.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, et al. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025.