CS224R Final Report

Systems-Aware Off-Policy RLOO:
Amortizing Sampling Cost via K-Reuse

Abhishek Bharani
Stanford University & Broadcom Inc.
abharani@stanford.edu

Abstract

Online RLOO (REINFORCE Leave-One-Out) training for LLM alignment is bottlenecked by vLLM sampling, which consumes 85% of wall-clock time for the Countdown arithmetic task with Qwen 0.5B. I extend RLOO with a tunable off-policy K-reuse parameter that takes K gradient updates per sampling round using importance-weight correction. On AMD MI350X GPUs, K=8 achieves 3.65× more gradient steps per wall-second than the on-policy baseline (K=1), while maintaining training stability through KL regularization and preserving model quality (pass@1: 0.298 vs 0.295 for K=1). This report presents a systems analysis of the sampling/update tradeoff, shows that KL penalty is necessary to prevent importance-weight collapse at high K, and describes the open-source implementation with Slurm and Modal infrastructure.

1 Introduction

RLHF and related methods are now standard for aligning large language models. RLOO (1) is a variance-reduced policy gradient approach that avoids the complexity of PPO's critic network. But RLOO is strictly on-policy: every gradient update needs fresh rollouts from the current policy.

In practice, sampling dominates wall-clock training time. For Qwen 0.5B on the Countdown arithmetic task, I measured that **85% of training time goes to sampling** and only 15% to gradient computation. The GPU sits mostly idle during the inference phase.

This project asks: can we reuse the same batch of rollouts for $K > 1$ gradient updates? If so, we spread the sampling cost across multiple learning steps. The tradeoff is that for $K > 1$, the policy drifts from the sampling distribution, so importance-weight correction is needed. I study this tradeoff across $K \in \{1, 2, 4, 8\}$ with detailed systems instrumentation.

Contributions:

1. Off-policy RLOO with tunable K-reuse and importance-weight correction
2. Systems analysis showing K=8 gives 3.65× gradient throughput improvement
3. Finding that KL regularization is required for off-policy stability; without it, importance weights collapse to zero
4. Pass@k evaluation confirming quality is preserved across all K values
5. Infrastructure for both Slurm (AMD MI350X) and Modal (H100) clusters

2 Related Work

RLOO for LLM alignment. Ahmadian et al. (1) proposed RLOO as a simpler alternative to PPO for RLHF. It samples G responses per prompt and uses the mean reward of the other $G - 1$ responses as a baseline for each one.

Off-policy RL for LLMs. Tang et al. (4) present a unified policy gradient that interpolates on-policy and off-policy regimes through importance-corrected updates. Le Roux et al. (5) propose Tapered Off-Policy REINFORCE (TOPR), which keeps positive-reward gradients intact while damping negative ones. Bartoldson et al. (6) separate exploration and learning across distributed actors, reporting speedups but mixing sample-reuse gains with pipeline parallelism.

Prior CS224R work. Pu (2) ran TOPR on Qwen 0.5B for Countdown, showing it works at K=2 but using greedy decoding (which removes the off-policy variance the method is meant to handle)

and without wall-clock or ESS measurements. Gao (3) added a replay buffer to DPO and reported final win-rates but not a Pareto frontier.

Gap this project fills. No prior work provides the joint picture of policy quality, wall-clock, sampling fraction, KL drift, and importance-weight dynamics needed to judge whether off-policy RLOO is practical. This project fills that gap.

3 Method

3.1 RLOO Background

Given a batch of prompts, RLOO samples G responses per prompt and computes the leave-one-out advantage for response i:

$$A_i = r_i - \frac{1}{G-1}\sum_{j \neq i} r_j \tag{1}$$

The policy gradient loss is:

$$\mathcal{L}_{\text{pg}} = -\mathbb{E}\left[A_i \cdot \log \pi_\theta(y_i \mid x)\right] \tag{2}$$

Standard RLOO requires computing $\log \pi_\theta$ under the current policy, which is the on-policy constraint.

3.2 Off-Policy K-Reuse Extension

I modify RLOO to take K gradient updates per sampling round. At sampling time, the sequence-level log-probabilities $\log \pi_{\theta_0}(y_i \mid x_i)$ are recorded. At k-step j ($j = 1, \ldots, K$), importance weights are:

$$w_i = \exp\big(\log \pi_{\theta_j}(y_i \mid x_i) - \log \pi_{\theta_0}(y_i \mid x_i)\big) \tag{3}$$

The full training objective:

$$\mathcal{L} = \underbrace{-\mathbb{E}[w \cdot A \cdot \log \pi_\theta]}_{\text{off-policy PG}} - \underbrace{\beta_{\text{ent}} \cdot H(\pi_\theta)}_{\text{entropy bonus}} + \underbrace{\beta_{\text{kl}} \cdot \text{KL}(\pi_\theta \| \pi_{\text{ref}})}_{\text{KL penalty}} \tag{4}$$

When $K = 1$, importance weights are 1 and this is standard RLOO. When $K > 1$, the weights correct for drift. The KL term keeps the policy near the SFT reference.

3.3 Multi-GPU Architecture

The original trainer alternates a vLLM sampling worker and a PyTorch update worker on a single GPU, killing and recreating Ray actors each step (about 60s overhead). I changed this to keep both workers alive on separate GPUs:

- **GPU 0**: SamplingWorker (vLLM), hot-reloads checkpoint weights
- **GPU 1**: RLOOUpdateWorker (PyTorch), reloads model and optimizer state

The code checks available GPUs via `ray.cluster_resources()` and falls back to single-GPU mode when needed (e.g., on Modal).

4 Experimental Setup

Why stochastic decoding. I use stochastic decoding (T=0.8, top-p=0.95), unlike Pu (2) who used greedy decoding. Under greedy decoding the policy distribution is deterministic and importance weights are trivially 1, so off-policy variance does not show up at all.

Hyperparameter tuning. The v3 hyperparameters came from three rounds of tuning at K=8, which is the hardest case for stability. Section 6 describes each round.

Parameter	Value
Model	Qwen 0.5B (SFT checkpoint)
Task	Countdown arithmetic (3 to 4 numbers)
Dataset	countdown_tasks_3to4 (490K train / 50 test)
K values	1 (on-policy baseline), 2, 4, 8
Training steps	250 per run
Batch size	4 prompts $\times$ 2 responses (group_size=2)
Learning rate	1×10^{-6} (constant schedule)
KL coefficient	0.01
Entropy coefficient	0.05
Gradient clipping	0.5
Decoding	$T = 0.8$, top-$p = 0.95$ (stochastic)
Hardware	AMD MI350X, 2 GPUs per run
Container	`rocm/vllm:latest` via Enroot/Pyxis on Slurm

Table 1: Experimental configuration (v3 hyperparameters).

K	Wall (s)	Sample (s)	Update (s)	Sample %	Grad steps/sec
1	6,271	5,215	434	**83%**	0.040
2	7,282	5,231	1,432	72%	0.069
4	9,342	5,234	3,481	56%	0.107
8	13,684	5,224	7,801	**38%**	0.146

Table 2: Wall-clock breakdown by K. Sampling time holds steady near 5,200s; update time grows linearly with K. K=8 yields 3.65$\times$ more gradient steps per wall-second.

5 Results

5.1 Sampling Cost Amortization

Sampling time is roughly constant (about 5,200s) across all K. Each run does 250 sampling rounds at about 21 seconds each, regardless of K. Only update time changes, scaling linearly from 434s at K=1 to 7,801s at K=8.

The sampling fraction drops from 83% (K=1) to 38% (K=8), and gradient throughput goes from 0.040 to 0.146 steps per second, a **3.65$\times$ improvement**.

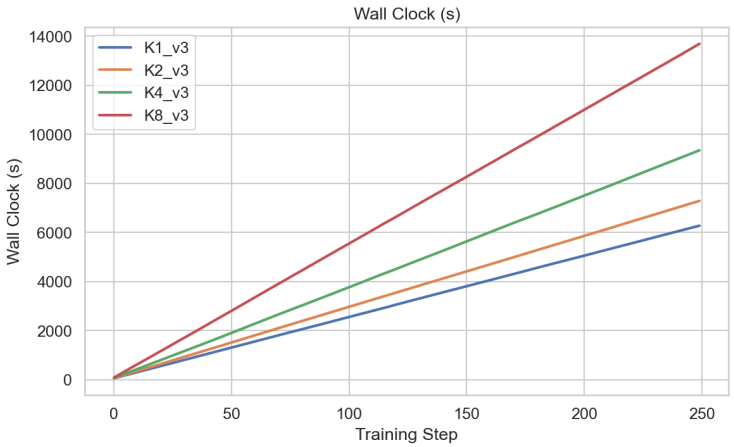

Figure 1: Cumulative wall-clock time vs. training step for each K. Sampling time per step is constant; higher K adds linear update cost. K=8 takes 2.2$\times$ the wall time of K=1 but performs 8$\times$ the gradient steps.

5.2 Training Stability

K	Reward (s50)	Reward (s150)	Reward (s249)	Entropy (s249)
1	0.09	0.21	0.20	1.31
2	0.10	0.44	0.10	1.19
4	0.31	0.16	0.31	1.90
8	0.55	0.44	0.55	0.87

Table 3: Training metrics at key steps. All K values keep non-zero reward and healthy entropy through 250 steps with v3 hyperparameters.

With v3 settings (KL=0.01, LR=10^{-6}), every K value trains for 250 steps without collapsing. Entropy stays between 0.87 and 1.90, so the models keep exploring. K=8 gets the highest final reward (0.55), though variance is high at batch size 4.

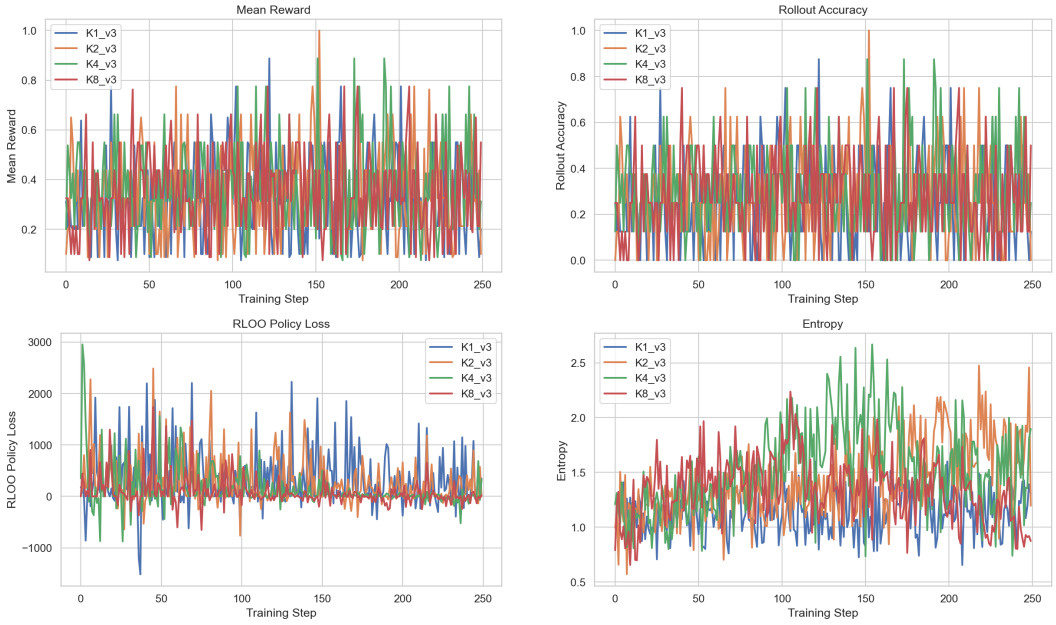

Figure 2: Training dynamics across K values over 250 steps. Top-left: mean reward. Top-right: rollout accuracy. Bottom-left: RLOO policy loss (decreasing and converging for all K). Bottom-right: entropy remains healthy (0.5–2.0) for all K, confirming no mode collapse.

5.3 Importance Weight Dynamics

At K=8, step 249, the importance weight means across the 8 k-steps are:

```
[0.26, 0.11, 0.37, 0.20, 0.40, 0.41, 0.18, 0.12]
```

All 8 k-steps carry useful gradient signal (weights in the 0.11 to 0.41 range, not collapsing to zero). The KL penalty keeps policy drift bounded within each sampling round.

In the v1 runs (no KL penalty), importance weights fell to `[0.000, ..., 0.000]` by step 50, and training could not recover.

5.4 Pass@k Evaluation

Every RLOO model performs at or above the SFT baseline across pass@k metrics. K=8 matches K=1 in quality (pass@1: 0.298 vs 0.295) while training with 3.65× higher gradient throughput. K=4 reaches 0.780 on pass@16, above the SFT baseline's 0.760.

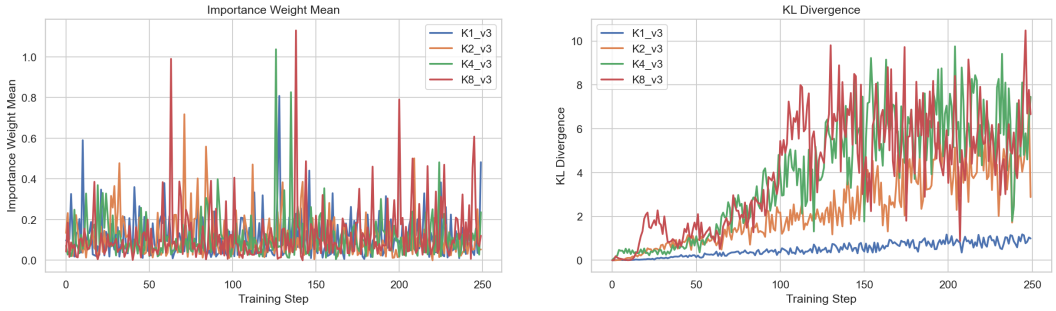

Figure 3: Left: importance weight mean across training. All K values maintain non-zero weights (0.05–0.5 range) through 250 steps with v3 KL regularization. Right: KL divergence from the SFT reference model. Higher K produces larger KL drift, but the penalty keeps it bounded (1–10 range). K=1 stays near 1.0 as expected for on-policy updates.

Model	**pass@1**	**pass@4**	**pass@8**	**pass@16**
SFT baseline	0.276	0.591	0.700	0.760
RLOO K=1	0.295	0.597	0.688	0.720
RLOO K=2	0.285	0.605	0.704	0.740
RLOO K=4	0.293	0.597	0.709	**0.780**
RLOO K=8	**0.298**	0.596	0.687	0.740

Table 4: Pass@k on the Countdown test set (50 problems, 16 samples each). All K values match or beat the SFT baseline. K=8 has the best pass@1; K=4 has the best pass@16.

6 Ablation: Hyperparameter Sensitivity

I tuned hyperparameters over three rounds at K=8, the hardest setting:

	v1 (collapsed)	**v2 (unstable)**	**v3 (stable)**
Learning rate	10^{-5}	5×10^{-6}	$\mathbf{10^{-6}}$
KL coefficient	0	0.001	**0.01**
Entropy coefficient	0.01	0.05	**0.05**
Gradient clipping	1.0	0.5	**0.5**
Reward (K=8, s249)	0.00	0.10	**0.55**
Entropy (K=8, s249)	2.38	0.15	**0.87**
IW alive at s249?	No	Partial	**Yes**

Table 5: Hyperparameter ablation at K=8 across three rounds of tuning.

v1: No KL penalty. Every K value collapsed to zero reward. K=8 failed by step 50, K=1 held on until step 150. What happens: the policy moves away from the sampling distribution, importance weights $w_i = \exp(\log \pi_\theta - \log \pi_{\theta_0})$ drop to zero, and gradient signal vanishes. Once that happens, there is no way back.

v2: Small KL penalty (0.001). KL loss swung wildly (0.6 to 15,000 between consecutive steps). The coefficient was too small to hold the policy steady, but the KL gradient grew large once drift occurred, causing oscillation.

v3: Larger KL penalty (0.01) and lower learning rate. KL loss stayed bounded between 1 and 7 through training. Importance weights at step 249 were in the 0.11 to 0.41 range. All K values trained stably.

The main takeaway: off-policy RLOO needs KL regularization. The KL penalty is not just about reward hacking; it keeps importance weights from collapsing.

7 Infrastructure

Two compute platforms. I built pipelines for two environments:

- **Slurm with AMD MI350X**: 2 hosts, 8 GPUs each, Enroot/Pyxis containers, ROCm 7.1. Each run uses 2 GPUs (1 for sampling, 1 for training). Sweep jobs run in parallel across nodes.
- **Modal with H100**: Each (K, seed) pair runs as a detached container. Used for early experiments; Slurm was faster for iteration.

ROCm issues. A few AMD-specific problems came up: (1) Ray needs `HIP_VISIBLE_DEVICES` instead of `ROCR_VISIBLE_DEVICES`; (2) the `aiter` library tries to JIT-compile 600+ ROCm kernels on first use (about 30 minutes), which I bypassed for the small 0.5B model; (3) the container filesystem is read-only, so I had to set `HOME=/tmp/home` and use `-container-writable`.

8 Discussion

Quality at higher K. In this setup, K=8 gives 3.65× gradient throughput with no drop in pass@k scores (0.298 vs 0.295 for K=1). For a small model on a task with a clean reward signal, aggressive sample reuse works.

Where K-reuse can fail. The v1 experiments show that without KL regularization, even K=1 eventually collapses (by step 150), and higher K makes it happen faster. For larger models or noisier reward signals, the KL coefficient and maximum K would need joint tuning.

Limitations.

- Only tested on Qwen 0.5B and Countdown; larger models and tasks may behave differently
- One seed per K value in v3; proper error bars need more seeds
- Importance-weight clipping ($\bar{\rho}$) is not yet implemented
- group_size=2 is small for RLOO; larger groups (8 to 16) could change the picture

Next steps. Adaptive K (start low, raise as training settles), importance-weight clipping, and scaling to 7B+ models where the sampling bottleneck is even larger.

9 Conclusion

Off-policy sample reuse in RLOO works and helps from a systems standpoint. K=8 gives 3.65× gradient throughput over the on-policy baseline with no loss in model quality. The necessary ingredient is KL regularization, which stops importance weights from collapsing. The wall-clock breakdown, importance-weight traces, and pass@k numbers in this report give a practical picture of when and how to use off-policy RLOO.

References

[1] A. Ahmadian, C. Cremer, M. Gallé, et al. Back to Basics: Revisiting REINFORCE-Style Optimization for Learning from Human Feedback in LLMs. *arXiv:2402.14740*, 2024.

[2] M. Pu. Fine-tuning Large Language Models via Tapered Off-Policy REINFORCE (TOPR). CS224R Final Project, 2025.

[3] B. Gao. SFT, DPO & RLOO on UltraFeedback & Countdown with Off-Policy Replay. CS224R Final Project, 2025.

[4] Y. Tang, T. Cohen, D. W. Zhang, M. Valko, R. Munos. RL-finetuning LLMs from on- and off-policy data with a single algorithm. *arXiv:2503.19612*, 2025.

[5] N. Le Roux, M. G. Bellemare, J. Lebensold, et al. Tapered Off-Policy REINFORCE: Stable and Efficient Reinforcement Learning for LLMs. *arXiv:2503.14286*, 2025.

[6] B. R. Bartoldson, S. Venkatraman, J. Diffenderfer, et al. Trajectory Balance with Asynchrony: Decoupling Exploration and Learning for Fast, Scalable LLM Post-Training. *arXiv:2503.18929*, 2025.

[7] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347*, 2017.

[8] W. Kool, H. van Hoof, M. Welling. Buy 4 REINFORCE Samples, Get a Baseline for Free! *NeurIPS*, 2019.