arXiv:2611.17924v1 [cs.LG] 13 Nov 2026

# TIDE: RECLAIMING LOST PERFORMANCE IN ON-POLICY REINFORCEMENT LEARNING

**Mathew Jacob**[*1] **Morgan Borjigin-Wang**[*2] **Kan Zhu**[1] **Po Hao Chen**[2] **Yi Pan**[3]
**Andy Ruan**[1] **Nayani Modugula**[4] **Ethan Lavi**[2] **Luis Ceze**[1] **Malte Schwarzkopf**[2]
**Akshay Narayan**[2] **Deepti Raghavan**[2] **Stephanie Wang**[1]

## ABSTRACT

Synchronous, colocated RL post-training of large language models (LLMs) today suffers from severe under-utilization due to *long-tail rollouts*. Inference engines fall idle waiting for long-tail rollouts or decode at low, inefficient batch sizes. To address this, we present TIDE, an RL training system that preserves on-policy training semantics while recovering resource utilization. TIDE does so by two complementary techniques: (1) dynamically repurposing rollout GPUs for training while long-tail rollouts complete and (2) dynamically migrating rollouts between inference engines to boost decoding throughput. TIDE automatically determines the right time to repurpose rollout resources through an adaptive migration policy, obtaining performance improvements that generalize across multiple applications and hardware configurations without requiring bespoke configuration. In our evaluation, TIDE provides end-to-end speedups of up to 1.36× over colocated systems, outperforming all evaluated on-policy baselines and achieving up to 97% of optimal throughput.

## 1 INTRODUCTION

Reinforcement learning (RL) is now an essential step in post-training of large language models (LLMs). RL techniques adapt models to specialized domains, improve general model reasoning capabilities, and align model outputs with human preferences (Guo et al., 2025; OpenAI et al., 2026; Olmo et al., 2026). RL works by repeatedly executing *rollouts* to generate data online using the current model weights, then *trains* on the generated data to update the model. The rollout and training phases have different computational characteristics and each employs specialized systems, such as vLLM (Kwon et al., 2023) or SGLang (Zheng et al., 2024) for inference and Megatron-LM (Shoeybi et al., 2020) or PyTorch FSDP (Zhao et al., 2023) for training. To make efficient use of GPUs, many RL systems *colocate* both phases on the same hardware, switching between inference and training systems within each RL step.

However, this colocated setup results in significant resource under-utilization. Current implementations are *synchronous*: they insert a barrier between inference and training so that all GPUs first execute rollouts, then after all rollouts complete, the entire cluster proceeds to the training phase. This is a simple way to ensure *on-policy training*, which requires that the training inputs come from current model weights.

Unfortunately, the barrier also creates two sources of under-utilization. First, due to the variation in rollout completion times, a single straggler will cause all other GPUs to remain *idle* in the rollout phase. Moreover, even the GPUs that remain active lose *decoding efficiency* as rollouts complete and decode batch sizes shrink. Across two different RL tasks on GPU configurations ranging from 4×L40S to 8×H200, we find that GPU idleness accounts for up to 30.0% of total run time, with inefficient decoding contributing another 9.8%. Together with phase-switching overheads (0.2-3.6%), these losses leave conventional colocated execution achieving as little as 56.6% of the optimal utilization.

Some prior efforts to address these inefficiencies adopt asynchronous, off-policy RL training (Fu et al., 2025). While this allows for higher GPU utilization, off-policy training must use stale weights and thus changes the training semantics. Other efforts preserve on-policy training (Hu et al., 2026) but empirically fail to generalize to RL training's diverse workloads (§6), which can span many end tasks, training backends, and hardware topologies.

We therefore ask: *can we eliminate the underutilization in colocated RL while preserving on-policy training semantics, without requiring workload-specific tuning?*

Our key observation is that *preserving on-policy training requires a barrier only before the weight update, not before*

[*]Equal contribution [1]University of Washington, Seattle, Washington, USA [2]Brown University, Providence, Rhode Island, USA [3]UC Berkeley, Berkeley, California, USA [4]New York University, New York, New York, USA. Correspondence to: Mathew Jacob <mjacob2@cs.washington.edu>, Morgan Borjigin-Wang <mbwang@brown.edu>.

Preprint. Under review.

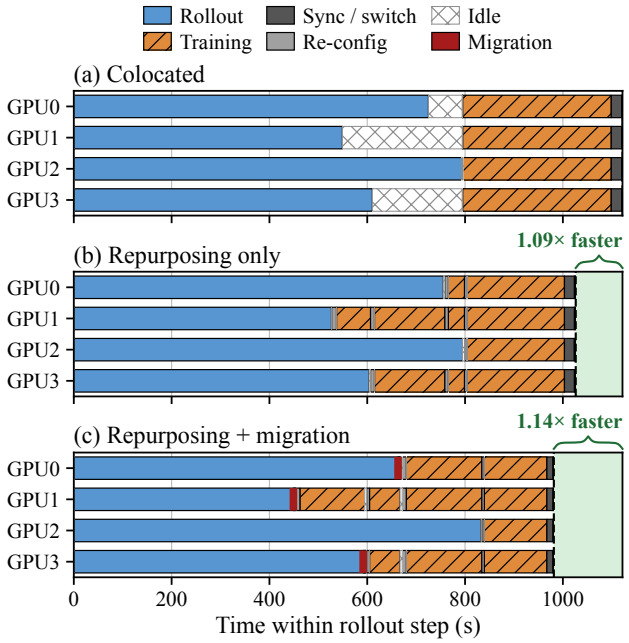



*Figure 1.* DAPO Math on 4×H100. Repurposing fills the idle time of the colocated baseline, and migration raises decode throughput through larger batches, together reducing the GPU-hours spent in the rollout phase.

*training begins*. Although the policy must remain fixed while generating rollouts for the current step, training can begin on completed rollouts while others are still being generated, provided that the weight update is deferred until training work is complete. Deferring the traditional barrier between rollout and training to before the weight update creates an opportunity to overlap inference and training within one RL step, using GPU capacity for training that would otherwise remain idle.

Building on this observation, we introduce TIDE, an on-policy RL training system that uses fine-grained resource reallocation to maximize GPU utilization. TIDE builds on two complementary mechanisms. First, we *repurpose* GPUs that have finished their rollouts to begin training on completed rollouts, enabling overlap of the rollout and training phases. Second, *request migration* consolidates outstanding rollouts onto fewer GPUs to improve decoding efficiency per GPU, while enabling further GPU repurposing. Together, these mechanisms allow TIDE to reclaim lost GPU-hours from both idleness and decoding inefficiency. We illustrate these components of our system in Figure 1.

A key challenge in TIDE is determining the best time during the rollout phase to migrate rollouts and repurpose GPUs to training. Both repurposing too soon and too late lead to under-utilization, and the best time depends on application characteristics and hardware configuration (§3). To address this, we design a robust migration policy that uses a simple and instantaneous signal to estimate the balance between rollout vs. training throughput, combined with a linear controller inspired by network congestion control.

A second challenge in TIDE is handling the now dynamically growing training group, while preserving on-policy semantics and supporting diverse training backends. We design an orchestrator for the training group to dynamically assign completed rollouts and manage synchronization across training GPUs. We also show that TIDE's techniques generalize across training backends, both data parallelism (DP) with tensor parallelism (TP) via Megatron and ZeRO via PyTorch FSDP.

Overall, we show that TIDE consistently achieves the best speedup over a colocated baseline compared to prior efforts that preserve on-policy training but fail to fully capture GPU waste. Across a range of workloads and hardware, TIDE achieves up to 36% faster runtime than the colocated baseline. Furthermore, TIDE's repurposing and migration policies effectively recover 77-97% of the optimal possible throughput.

In summary, this paper makes the following contributions:

- We systematically categorize and quantify the sources of under-utilization in synchronous colocated RL training and identify the missed opportunities (§3).
- We design an adaptive policy for repurposing under-utilized rollout GPUs (§4.1) and build a system, TIDE, that generalizes to a range of application, training backends, and hardware.

## 2 Related Work

**Overlapping-Based Methods.** Like TIDE, some past RL systems attempt to repurpose GPUs to overlap tasks and hide latency across or within stages of the RL pipeline.

RollPacker (Gao et al., 2026) proposes tail batching: it over-provisions rollout samples and trains on the fastest completions. The unfinished long tails are batched into a later training step. Tail batching implicitly skews the distribution of the training samples, so it does not preserve the original RL semantics. RollPacker additionally proposes overlapping inference and training by opportunistically switching a subset of GPUs from inference to training using a policy that switches exactly once when certain heuristic conditions are satisfied. TIDE repurposes GPUs at a finer granularity, and we show in §6.2 that this results in greater speedups.

RLHFuse (Zhong et al., 2025b) overlaps rollout generation with reward computation, but not training. RLHFuse also focuses on the PPO algorithm (Schulman et al., 2017), and does not apply to the critic-free GRPO (Shao et al., 2024) setting that is popular today and which TIDE targets.

**Speeding Up Long-Tail Inference.** RL systems like TLT (Hu et al., 2026), DAS (Shao et al., 2026), and ReSpec (Chen et al., 2026) attempt to mitigate straggler rollouts via speculative decoding (SD). Other methods attempt to use previous historical rollouts of the same prompt across epochs as a speculative draft for the rollout in a future RL step(He et al., 2026).

In §6.2, we show that TIDE outperforms TLT due to the implicit tradeoffs of enabling SD. Further, SD cannot accelerate tool calls and relies on training an acceptable draft model for good performance.

**Training-Inference Disaggregation.** Instead of a colocated architecture, some RL systems such as StreamRL (Zhong et al., 2025a) and ThunderSyncRL (Kang et al., 2026) instead partition GPUs into separate inference and training pools and stream completed rollouts to training engines. This allows, for example, using hardware specialized for inference and training, respectively. However, disaggregation (i) requires significantly more GPU resources, and (ii) requires careful tuning to balance inference and training resource allocations. TIDE similarly streams generated samples to trainers, but does so for colocated architectures and automatically balances inference and training resource allocations.

**Algorithmic Modifications.** Asynchronous RL systems such as AReaL (Fu et al., 2025) make algorithmic modifications to the RL workflow by allowing off-policy rollouts, i.e., rollouts from older iterations of the model. While this design supports high GPU utilization, training on rollouts from prior model versions changes the training semantics and can result in training instability. More recently, QwenGyre (Wang et al., 2026) also allocates GPUs elastically between inference and training, but uses off-policy learning and requires a static configuration of its repurposing threshold. While TIDE focuses on improving utilization while maintaining the semantics of synchronous on-policy RL training, its techniques—specifically, migrating rollouts and automatically determining the migration threshold—could apply to repurposing GPUs in other RL training designs as well.

## 3 Addressing Under-Utilization

This section quantifies the impact of two sources of GPU under-utilization, both caused by long-tailed rollouts (§3.1). First, GPUs that finish their work early sit idle waiting for the last GPU. Second, as each inference engine nears the end of its assigned work, its batch size decreases, causing under-utilization. We then quantify the potential speedup from repurposing GPUs mid-step (§3.2) and the challenges TIDE must solve to realize this speedup (§3.3).

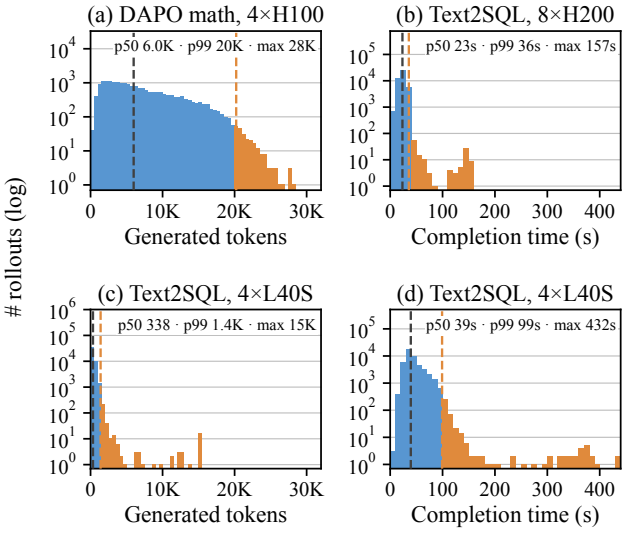



*Figure 2.* Rollout lengths vary across workloads (a, c), and completion times vary across hardware (b, d).

### 3.1 Sources of GPU Under-utilization

**Long-Tailed Rollouts.** In RL rollouts, stragglers can arise from long responses, multi-turn rollouts, and long tool calls. Stragglers' occurrence varies across workloads, models, and hardware. For example, Figure 2 (a) and (c) show the variance in sequence length distribution across two datasets (§6.1 describes these datasets in detail). DAPO Math has a p99 response length of 20k tokens and a maximum response length of 28k, while Text2SQL, which requires multi-turn generation and tool use, has a p99 and max response length of 1.4k and 15k, respectively.

Hardware differences also influence stragglers. For example, Figure 2 (b) and (d) show rollout completion times on NVIDIA 8xH200 and 4xL40S, respectively. For Text2SQL, running the benchmark on 8xH200s results in a p99 and maximum completion time of 36s and 157s, respectively; on the 4xL40S, the p99 and maximum completion time of rollouts is 99s and 432s, respectively.

**Idle GPUs** When stragglers occur, some inference engines finish the rollout stage long after the rest (Figure 3). With a barrier between the rollout and training phases, all inference engines must complete before training begins. So, while engines processing stragglers complete, other inference engines *idle*, wasting up to 30% of GPU-hours.

**Decode Inefficiency** Stragglers can cause under-utilization even for non-idle GPUs. While training is typically compute-bound, LLM decoding is memory-bound since generating a single token requires loading the full model weights. Batching multiple sequences together amortizes this load. However, on-policy RL uses a fixed

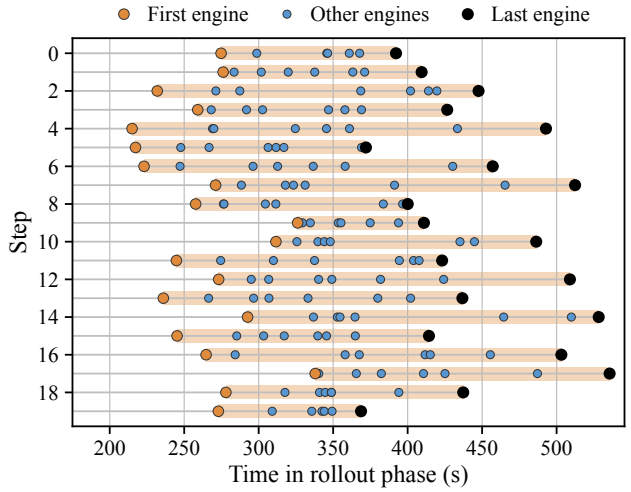



*Figure 3.* Skew in inference-engine finish times across a 20-step colocated DAPO-Math run (DeepSeek-R1-Distill-8B, 8×H200). In every step, most engines finish well before the last one, leaving their GPUs idle.

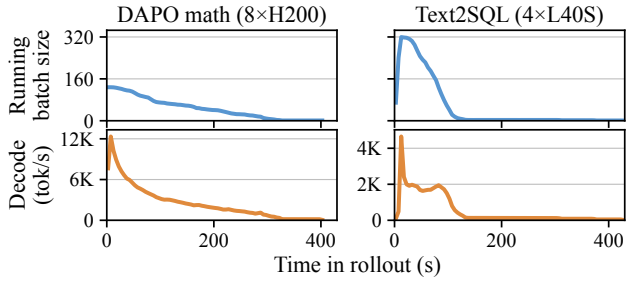



*Figure 4.* As rollouts complete, the batch shrinks and throughput falls far below peak while engines decode the last few stragglers.

set of rollouts in the inference phase, so as Figure 4 shows, running batch sizes and correspondingly decode throughput per engine declines over time, resulting in wasted GPU-hours from underutilized hardware.

To quantify this *decode inefficiency loss*, we measure the throughput of a single GPU running all of the rollouts for a step. This allows the batch size on the GPU to be as large as possible, for as long as possible, maximizing the throughput and minimizing the number of GPU-hours the rollout stage takes. This allows us to measure the best-case time required to generate all tokens in a step.

We then measure the GPU-hours spent by the colocated RL system during active rollout generation and compare it with the maximal throughput's GPU hours. We see that during inference, on the DAPOxH200 configuration, colocated achieves only 84% of optimal decode throughput, resulting in 12.6% wasted rollout-GPU hours, or 8.4% wasted GPU hours across the entire run.

*Table 1.* Breakdown of each step's GPU hours under colocated execution (%). The three losses and the fraction of optimal throughput sum to 100%. Misc. idle is GPU time outside both phases, spent offloading and onloading engine and trainer state at each phase switch.

| Application | GPUs | Idle loss | Ineff. loss | Misc. idle | Of optimal |
|---|---|---|---|---|---|
| DAPO Math | 8×H200 | 15.2 | 8.4 | 1.3 | **75.1** |
| R1-Distill-8B | 4×H100 | 12.7 | 3.9 | 0.2 | **83.2** |
| Text2SQL | 8×H200 | 30.0 | 9.8 | 3.6 | **56.6** |
| Qwen2.5-Coder-7B | 4×L40S | 18.5 | 2.5 | 0.6 | **78.4** |

**Utilization-optimal** We now seek to understand the optimal performance on-policy RL can achieve on a given workload, if GPU utilization is maximized. We assume that because training is compute-bound, it already operates at its peak throughput. Thus, we can compute the optimal GPU-hours by adding the optimal inference GPU-time described above with the GPU-time needed for training, collected from the actual workload. We also add the actual time needed to weight-sync between the inference and training engines. We call this upper bound the overall utilization-optimal.

The percentage of utilization-optimal that synchronous colocated RL achieves under various workloads and settings is recorded in Table 1. Across both hardware setups for the Text2SQL and DAPO Math datasets, idle and inefficient GPU time together cost 17–40% of the GPU hours in each step (Table 1). Assuming that the training phase is already fully utilized, colocated execution thus achieves only 57–83% of the optimal throughput.

### 3.2 Opportunities: Repurposing GPUs and Migrating Requests

§3.1 shows how both idleness and decode inefficiency cause significant waste in the colocated RL system. We propose two complementary mechanisms, repurposing and migration, to reclaim these wasted GPU-hours. Figure 1 illustrates with data from a real run how these mechanisms complement each other.

**Repurposing** allows idle GPUs to begin training on completed GRPO groups without waiting for the entire rollout phase to finish. Since training is compute-bound, it can efficiently utilize these otherwise idle GPUs, reclaiming the idle loss it would have otherwise incurred.

**Migration** improves inference efficiency by consolidating the remaining rollouts onto fewer engines, allowing for a larger batch size and thus higher decode throughput. Meanwhile, the GPUs that migration vacates can in turn be repurposed for training. This optimization can reclaim the lost GPU-hours from decode inefficiency loss.

### 3.3 Challenges

Realizing repurposing and migration in a colocated RL system raises two main challenges.

**Challenge 1: Migration Across Diverse Workloads and Hardware** Determining the best time to migrate an inference engine is challenging. The ideal migration point maximizes GPU utilization in both rollout and training, but no one point works across workload and hardware configurations. It is difficult to determine this point analytically for various reasons:

- **Hardware** sets KV cache capacity, memory bandwidth, and compute bandwidth. This directly affects the inference vs. training throughput.
- Both the **Model** and **Dataset** affect the token length distribution of rollouts.
- **Agentic settings** add extra costs from tool-calling times, as well as extra prefill cost from appending the output of a tool call to the context.
- **Model learning** shifts the rollout distribution from step to step as the policy learns.

The system therefore needs a *low-cost online policy that picks a good migration point during the rollout phase*, rather than a static value tuned offline for one setting.

**Challenge 2: RL Training with Dynamic GPU Allocation** Flexible repurposing of GPUs introduces new challenges when implementing the training phase.

First, training requires more memory than inference, and thus typically uses a different parallelism configuration. For example, the DAPO Math × H200 configuration runs a data-parallel inference engine on each of eight GPUs, while its training phase runs four data-parallel training engines, each with TP=2. As a result, it is not always possible to immediately repurpose inference engines since their GPUs may not fit a training engine. Similarly, the repurposing granularity depends on the training backend. With FSDP, training can begin once enough GPUs are free to collectively hold the trainer state; after that, GPUs can join one at a time, at some reconfiguration cost. With data parallelism composed with model parallelism (e.g., Megatron), training needs at least one complete model-parallel group, and capacity grows only in whole replicas. Repurposing must therefore free this many GPUs to be worthwhile.

Meanwhile, work assignment to training engines cannot know the set of rollouts ahead of time. In synchronous colocated execution, all samples exist before training starts, so they can be divided evenly across replicas upfront. When GPUs are repurposed mid-rollout, completed groups instead arrive one at a time, replicas join at different times, and the longest groups arrive last. Because every replica must finish before the shared gradient synchronization, a replica that receives a large share of these late groups delays all others; the trainer must therefore assign work online so that the tail of the rollout is balanced across replicas.

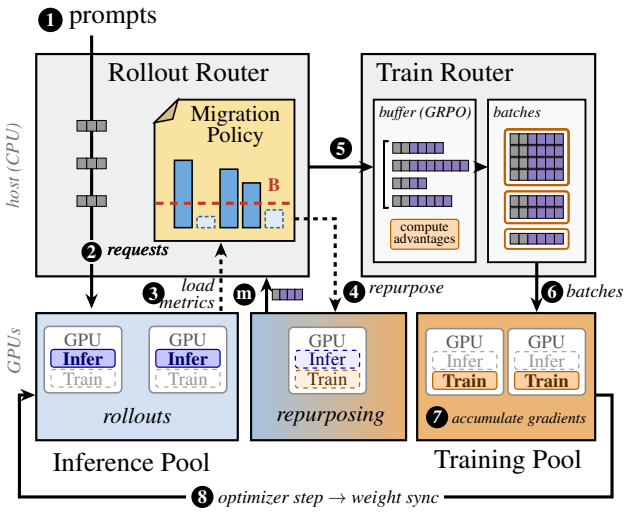



*Figure 5.* TIDE overview mid-step. Each GPU hosts an inference and a training engine (one live, one asleep); routers run on the host. ❶ Prompts arrive at the Rollout Router, which ❷ dispatches them to inference GPUs that ❸ report load. When a GPU's live rollouts fall below $B$ decided by the migration policy (§4.1), its requests are ⓜ re-dispatched and the GPU is ❹ repurposed for training. ❺ Completed rollouts stream to the Train Router (§4.2), which computes GRPO advantages and ❻ dispatches batches. ❼ Gradients accumulate; ❽ after the last rollout, the optimizer steps and weights sync via host memory.

## 4 TIDE

To address the challenges described in §3.3 we introduce TIDE, an on-policy RL framework that automatically maximizes GPU utilization. Figure 5 illustrates TIDE's two primary components:

- The *Rollout Router* orchestrates the inference engines. It assigns rollout work to inference engines, uses a *migration policy* (§4.1) to decide when to migrate work away from an engine, and repurposes inference engines for training by handing them to the Train Router.
- The *Train Router* orchestrates the training engines. It dynamically incorporates new GPUs into the trainer, assigns training work to replicas, and synchronizes the training replicas (§4.2).

At the beginning of each RL step, the Rollout Router distributes the batch of prompts across inference engines. As inference engines generate rollouts, they continuously report load metrics to the Rollout Router. The Rollout Router consults the migration policy to detect when to trigger mi-

gration and repurposing. When an inference engine meets migration conditions, the Rollout Router preempts the engine's running requests, migrates them to the least-loaded engine, scales down the now idle inference engine, and hands ownership of the GPUs to the Train Router.

The Train Router consumes rollouts produced by the Rollout Router and dispatches batches to training engines for gradient generation. Once all rollouts in the batch have been processed by training, TIDE performs the configured optimizer step and weight synchronization. The process repeats for the next step.

In the rest of this section, we describe TIDE's key design features: a migration policy that adaptively finds the best migration point for a dynamic workload while remaining training-aware (§4.1), and a parallelism-aware Train Router (§4.2).

### 4.1 Migration Policy

The simplest policy for repurposing is to disable migration entirely and repurpose each inference engine when it finishes its rollouts. However, this wastes efficiency from under-utilized inference GPUs. TIDE's migration policy thus aims to identify these under-utilized GPUs and compact their work to maximize overall utilization.

Determining the conditions for migration is therefore key to maximizing GPU utilization. We frame this problem as a search for a threshold $B$: when the number of remaining rollouts falls below $B$ on an inference engine, TIDE triggers a migration from that engine and repurposes the victim engine's GPUs for training.

As Figure 6 illustrates, late migrations (low values of $B$) cause inference engines to linger at low utilization. Meanwhile, early migrations (high values of $B$) can reduce inference throughput prematurely and create a bottleneck for training, again leading to underutilization. As a result, the optimal migration point is somewhere in the middle.

#### *4.1.1 Finding B*

The optimal $B$ depends on workload and hardware characteristics. To discover the optimal $B$ value, TIDE uses a simple linear controller inspired by delay-based network congestion controllers (Brakmo et al., 1994), shown in Algorithm 1 alongside the associated migration logic. Intuitively, the controller should set $B$ such that inference vs. training throughput are evenly matched, i.e. neither phase is the bottleneck. Training is compute-bound and its inputs are known ahead of time, so training throughput is relatively easy to estimate. On the other hand, rollouts generate inference work dynamically and it is challenging to estimate the maximum possible inference throughput for a given rollout a priori, as this is workload-dependent (§3.3). Thus, the main challenge is finding a reliable signal for the balance point between inference and training.

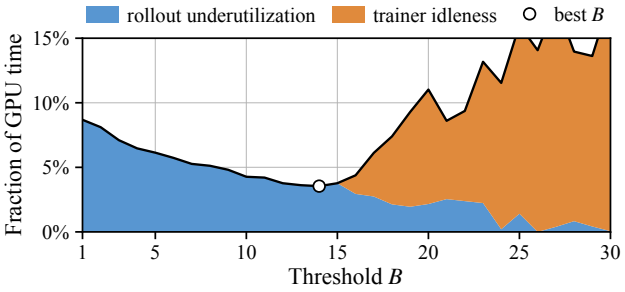



*Figure 6.* Simulated (§6.4) performance loss from the utilization-optimal at different values of $B$ for a single step of DAPO-Math. Migrating too early (higher $B$) and migrating too late (lower $B$) both result in underutilization.

We observe that increasing $B$ generally improves total GPU utilization until the trainer begins to starve (Figure 6). Thus, the best $B$ is the largest value that avoids training starvation. Meanwhile, training starvation is also easy and robust to measure online; we simply check for idle time on each training GPU. Therefore, TIDE uses starvation as a binary feedback signal. After each RL step, if *any* training starvation occurred, the controller decreases $B$ by a configurable fixed step $\Delta B$; otherwise, it increases $B$ by $\Delta B$. Because the controller updates $B$ every step, it requires no offline profiling or user tuning and adapts as the rollout distribution shifts over the course of training.

TIDE must also pick $B_0$, the initial value of $B$. A single value may not work across all workloads and hardware environments. Two reasonable defaults would be the minimum and maximum possible $B$. To this end, we observe that the two extremes Figure 6 illustrates are not equally costly; while inference engines still make progress with low batch sizes, starved training engines make no progress at all. As a result, a high-$B$ policy (migrate too early) would be worse than a low-$B$ policy (migrate too late). So, TIDE conservatively sets $B_0$ to the controller's step size by default, although $B_0$ can be manually configured if desired. As we will show in §6.4, even with a conservative $B_0$ TIDE delivers good performance over a range of workloads and hardware configurations.

#### *4.1.2 KV cache contention*

While our evaluation shows that training starvation is a reliable signal for balancing rollout vs. training throughput, one shortcoming is that we can only apply the signal to the *subsequent* rollout steps. Thus, if a particular step deviates significantly in its optimal $B$ from the previous step—for example, due to an unexpected straggler—then TIDE may migrate earlier than is optimal. Also, TIDE cannot read the

**Algorithm 1** Batch-threshold repurposing and linear control algorithm

```
1:  // On rollout completion
2:  for each engine e ∈ RolloutEngines do
3:    if NumRollouts(e) <= B then
4:      dest ← LeastLoaded(RolloutEngines)
5:      if KV(e) + KV(dest) > 100% then
6:        B -= ΔB
7:        continue
8:      end if
9:      Migrate(dest, e)
10:     RepurposeToTraining(e)
11:   end if
12: end for
13:
14: // On full step completion
15: if StarvationDetected(TrainEngines) then
16:   B -= ΔB
17: else
18:   B += ΔB
19: end if
```

training starvation signal until trainers begin execution; so, if migration happens too early, the described policy cannot recover for the current step. This manifests in slowdowns from overloaded inference engines.

As a safeguard for such cases, we supplement the inter-step controller with intra-step information that we can collect at the intended migration time. In particular, we add the *KV cache veto* (line 5 of Algorithm 1) which estimates whether a migration destination is at capacity by reading the receiving engine's KV cache capacity and delays rollout migration (line 6) if it is saturated. By the end of the step, this procedure reduces $B$ to the maximum value that would avoid saturating the most constrained KV cache. This helps constrain the range of $B$ to values which trigger migrations with ample KV cache capacity.

#### *4.1.3 Training-aware migration*

Training in TIDE remains on-policy as long as there is barrier between rollout generation and the optimizer step. So, to ensure on-policy training, TIDE defers the optimizer step and all nonlinear whole-batch operations such as gradient clipping until all gradients have been accumulated.

Training engine configurations can both introduce reconfiguration overheads and affect whether a migration results in a speedup. For example, a Megatron training engine with TP=2 cannot incorporate a single GPU to do training, meaning vacating only one GPU does not allow any repurposing to happen. Likewise, an FSDP training engine may not be able to fit unsharded model weights on a single GPU. To avoid wasteful migrations, the Rollout Router only triggers a migration if there exists a set of migration-eligible inference engines which can immediately contribute to training.

Similarly, the Rollout Router must consider the overhead of adding GPUs to training. FSDP shards model weights across all GPUs, so adding a GPU to an FSDP training engine requires pausing to redistribute weights. To reduce the number of training reconfigurations, the Rollout Router can bundle GPU migrations which happen in quick succession into a single event so the Train Router can add all GPUs with a single reconfiguration.

### 4.2 Train Router

Synchronous RL systems know the entire training workload when training starts, allowing for optimal work distribution between training engines. With TIDE, training engines must process a stream of rollouts on a dynamically changing group of training GPUs. Because rollout lengths can vary, it is possible for streaming rollouts to cause workload imbalance on training engines. For example, longer rollouts tend to finish later. Thus, if independent data-parallel trainers pull batches of work greedily from the work queue with no load-balancing mechanism, a small subset of training engines then train on a batch of much longer requests. This causes starvation among other training workers, which harms performance (Table 3).

To address this challenge, TIDE includes a Train Router, which ensures that data-parallel training engines pull gradually smaller batches near the end of an RL step. At the beginning of a step, the Train Router sends 64 rollouts at a time to training engines. As the number of remaining rollouts decreases, the Train Router sends at most one fourth of the remaining rollouts to the training engines. Reducing the training batch size given to engines only towards the end of training outperforms both the naive work-distribution scheme that causes trainer idling and an alternative policy that uses small training batches throughout training. This is because the Train Router must do some fixed work per batch, so always using small batches incurs overhead.

## 5 Implementation

We support two training backends to show TIDE's applicability across workloads. TIDE-Megatron builds on the Slime (Zhu et al., 2025) post-training framework and uses SGLang for inference and Megatron (Shoeybi et al., 2020) for training. TIDE-FSDP, specialized for memory-constrained hardware, builds on the SkyRL (Cao et al., 2025) post-training framework and uses vLLM for inference and FSDP2 (Zhao et al., 2023) for training. TIDE-FSDP additionally applies memory-saving optimizations to SkyRL to minimize the memory footprint of training. Most signifi-

cantly, it completely offloads the optimizer to CPU memory akin to ZeRO-Offload (Ren et al., 2021), allowing training to begin even when limited GPU memory is available. TIDE-FSDP also enables sequence packing of variable-length sequences (Krell et al., 2021), allowing denser microbatches. Both TIDE-MEGATRON and TIDE-FSDP use Ray (Moritz et al., 2018) actors to encapsulate GPU workers, use SkyRL-Gym for tool-calling support, and use the Adam optimizer (Kingma & Ba, 2017).

**Rollout Router** The Rollout Router assigns rollouts to inference engines and manages request migration once TIDE repurposes inference engines for training. It implements group affinity, which ensures the same inference engine handles all of a GRPO group's samples. This increases prefix cache hit rates, which are especially important for multi-turn trajectories.

**Request Migration** To trigger a migration, the Rollout Router (i) preempts all in-flight rollouts, (ii) captures any generated tokens, and (iii) sends them to the least-loaded remaining inference engine. We experimented with using KV-cache occupancy to select migration destinations–including distributing requests across multiple engines–but found that the least-loaded inference engine policy performed best. We choose to migrate tokens rather than KV cache contents since it is simpler and offered the same performance.

**Repurposing GPUs** While GPUs process rollouts, TIDE swaps all training state to CPU memory using the base RL system's existing mechanisms. TIDE-MEGATRON uses data parallelism (DP) and tensor parallelism (TP), so TP groups are independent and synchronization across groups is not needed. In TIDE-FSDP, adding a new FSDP rank requires re-sharding training state and re-building NCCL communicators. So, when TIDE-FSDP triggers a reconfiguration, it waits for existing workers to complete their assigned microbatches, then gathers the sharded gradient in a pinned CPU buffer. Then, it assigns new sharding keys to the new set of train workers, which then pull their corresponding gradient and weight shards from CPU memory. This adds 2-3% overhead overall to TIDE-FSDP. Future work might reduce this overhead with direct GPU-GPU communication.

# 6 EVALUATION

To evaluate TIDE, we ask the following questions:

1. How do the end-to-end speedups TIDE provides compare to prior RL training systems? (§6.2)
2. How do TIDE's components contribute to the speedups it offers? (§6.3)
3. Does TIDE's controller discover effective migration thresholds, despite workload variation? (§6.4)

## 6.1 Setup

**Applications** We evaluate TIDE on two applications: SkyRL's Text2SQL dataset (Liu et al., 2025) and DAPO-Math (Yu et al., 2025). For Text2SQL, we follow the training recipe from the SkyRL-SQL paper, fine-tuning Qwen2.5-Coder-7B-Instruct on Text2SQL over 35 steps. A Text2SQL rollout consists of 1-5 sequential LLM generations of at most $3,000$ tokens, interspersed with SQL requests against a SQLite database in the style of a ReAct agent (Yao et al., 2023). We show the rollout distribution in Figure 2(c).

For the DAPO-Math task, we fine-tune DeepSeek-R1-Distill-Llama-8B over 20 steps. A DAPO-Math rollout consists of a single long LLM generation of up to $32,000$ tokens. Synchronous runs of DAPO-Math are bottlenecked by the long-tail, but to a lesser extent than Text2SQL: as shown in Figure 2(a), the maximum observed rollout generation length is $28,000$ tokens, and the median is $6,000$ tokens. A synchronous RL system exhibits significantly less idleness on DAPO-Math than on Text2SQL (Table 1).

**Hardware** We evaluate on the following single-node testbeds: `8×H200` (141 GB HBM3e each; NVLink; 2× Intel Xeon Platinum 8480C (112 cores); 2 TB DDR5 host memory), `4×H100` SXM GPUs (80 GB HBM3) in an NVIDIA DGX H100 node, connected all-to-all by fourth-generation NVLink through NVSwitch (900 GB/s per GPU), with 2× Intel Xeon Platinum 8480C CPUs (112 cores) and 2 TB of host memory. `4×L40S GPUs` (48 GB GDDR6 each) connected over PCIe Gen4 (no NVLink), with 2× AMD EPYC 9954 CPUs (128 cores) and 1.5 TB of host memory. We refer to each testbed by its GPU configuration.

**System Configurations** We cover both memory-abundant and memory-constrained settings. Memory-abundant settings use Megatron with TP=2 and DP on the H200 testbed. Memory-constrained settings use FSDP, running Text2SQL and DAPO-Math on the L40S and H100 testbeds respectively. Unless stated otherwise, TIDE's migration policy uses $B_0 = 8$ and $\Delta B = 8$ (§4.1).

## 6.2 End to End Results

We show the results of the Text2SQL (35 steps) and DAPO Math (20 steps) experiments across the testbeds in Figure 7.

We compare TIDE against three baselines:

1. Synchronous colocated RL training ("Colocated").
2. RollPacker's algorithm (Gao et al., 2026) for GPU repurposing ported onto TIDE for fair comparison.
3. TLT (Hu et al., 2026), a synchronous RL system that accelerates long-tail rollouts using speculative decoding.

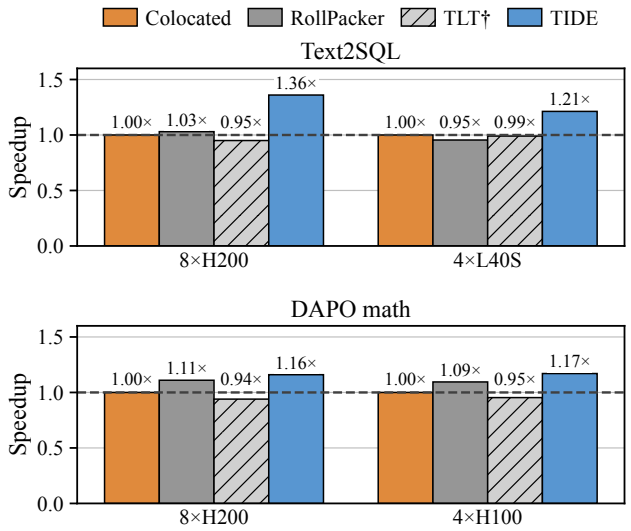



*Figure 7.* End-to-end speedup over fully synchronous colocated execution (baseline step time divided by system step time). †TLT's speedup is relative to its own run without speculative decoding.

*Table 2.* Fraction of utilization-optimal throughput (optimal / system step time).

| Application | GPUs | Colocated | TIDE |
|---|---|---|---|
| DAPO Math | 8×H200 | 75.1% | **87.0%** |
| R1-Distill-8B | 4×H100 | 83.2% | **97.4%** |
| Text2SQL | 8×H200 | 56.6% | **77.2%** |
| Qwen2.5-Coder-7B | 4×L40S | 78.4% | **95.1%** |

To evaluate the best possible speedup TLT could offer, we made two changes. First, TLT calls for online training of the draft model, which can add run-time overhead; we instead trained TLT's EAGLE-2 draft model on application rollouts offline, eliminating any possible run-time overhead. Second, TLT uses verl (Sheng et al., 2025) as its base RL system, which introduces substantial overheads over TIDE's base systems Slime and SkyRL; we thus report TLT's speedup over itself without speculation instead.

Over all workloads and testbeds, TIDE outperforms all baselines. On the H200 testbed it improves upon colocated execution by 1.36× on Text2SQL and 1.16× on DAPO-Math. On the memory-constrained testbeds the speedups are 1.21× (Text2SQL) and 1.17× (DAPO Math). We also see that, on all workloads, TIDE recovers a large amount of the optimal throughput, ranging from 77.2% to 97% (Table 2). Text2SQL broadly exhibits larger speedups than DAPO-Math, as multi-turn trajectories increase the dispersion of rollout lengths, leading to added idle time. Under colocated execution, 30.0% of the step's GPU-time is idle on Text2SQL against 15.2% on DAPO Math (§3.1), so there is twice as much idle time for TIDE to reclaim.

**Rollpacker** harvests some idle GPU time, but due to its one-shot coarse migration of exactly half of all GPUs, it regularly leaves training or rollout generation underutilized.

**TLT**'s speculative decoding (SD) yields poor end-to-end performance, as improvements in tail decoding speed are heavily workload-dependent and offset by TLT's overheads. The draft model achieves mean acceptance lengths of 4.86 and 2.25 (out of a maximum of 9) on Text2SQL and DAPO-Math, against break-even length of 2.48 (4×L40s) and 2.25 (4×H100), respectively: speculation accelerates tail decoding by 64% on Text2SQL but only breaks even on DAPO-Math. However, because SD can reduce inference throughput when the workload is compute-bound, TLT only enables SD below a certain inference batch size. Unfortunately, this phase switch between SD enabled vs. disabled still incurs overheads including an additional prefill with the draft model at the phase switch and the inability to selectively enable certain engine optimizations only when SD is disabled. Finally, the draft model adds memory overhead, which can slow down training in memory-constrained settings. Thus, in all evaluated cases, TLT's performance actually regresses from that of its colocated counterpart.

**Step-by-step speedups** Digging deeper into TIDE's performance, we show how TIDE's gains vary across RL steps. Figure 8 shows the distribution of per-step speedups for all 4 workloads. On the H200 testbed, TIDE achieves speedups in 100% of DAPO Math steps and 91% of Text2SQL steps. The speedups in Text2SQL are more variable because the Text2SQL rollout distribution is more skewed (Figure 2); in some steps, a few straggler rollouts allow TIDE to almost entirely hide training work.

The memory-constrained settings (4xL40S and 4xH100) use TIDE-FSDP to fit training on the available resources. Because repurposing for FSDP incurs more reconfiguration overheads than repurposing for DP, we sometimes see neutral or negative speedup on Text2SQL on the L40S testbed, in particular when there are fewer stragglers. Fixed repurposing costs are more exposed in Text2SQL, which has shorter steps than DAPO. In aggregate, TIDE-FSDP still achieves a significant speedup over the colocated baseline.

### 6.3 Ablation Study

To measure the contribution of each TIDE feature, we enable them one at a time and evaluate the speedup on DAPO-Math (Table 3), on both the 8×H200 testbed and the memory-constrained 4×H100 testbed. The features are, in order: *repurposing*, which enables inference engines to release their GPUs to training as soon as their rollouts finish; *load balancing*, which evenly distributes work across the training GPUs (§4.2); *migration*, which migrates the remaining rollouts off underutilized GPUs once their count

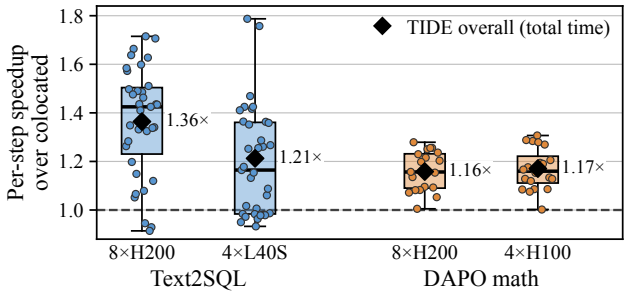



*Figure 8.* Per-step speedup of TIDE over colocated (points: steps; diamonds: overall; dashed: 1×). The median step is faster in every configuration, with overall speedups of 1.16–1.36×.

*Table 3.* Ablation on DAPO Math: seconds per step, with speedup over colocated in parentheses. Each row adds one mechanism to the row above; bold marks the best per column.

| System | 8×H200 (TIDE-MEGATRON) | 4×H100 (TIDE-FSDP) |
|---|---|---|
| Colocated | 689.1 (1.00×) | 1142.6 (1.00×) |
| + Repurposing | 666.8 (1.03×) | 1025.5 (1.11×) |
| + Load balancing | 617.9 (1.12×) | N/A† |
| + Migration | **572.5 (1.20×)** | 1052.4 (1.09×) |
| + KV veto | 576.9 (1.19×) | **1002.9 (1.14×)** |

† TIDE-FSDP always dispatches microbatches to FSDP workers with centralized bin-packing, so repurposing already includes load balancing.

falls below an adaptively tuned threshold (§4.1); and the *KV veto*, which blocks migrations that would overflow the KV cache of the receiving engines (§4.1.2).

We find that repurposing GPUs alone does not yield benefits on the 8×H200 testbed (1.03×) since large batches cause load imbalance near the end of each step. With load balancing, the speedup on 8×H200 is 1.12×, and on the 4×H100 testbed the speedup is 1.11×.

Adding migration raises the 8×H200 speedup from 1.12× to 1.20×: consolidating the remaining rollouts onto fewer engines frees GPUs for training earlier, instead of waiting for each engine to drain on its own.

Finally, while the KV veto has no measurable effect in the 8×H200 testbed since inference engines are well-provisioned and rarely experience KV-cache pressure, it is essential on the memory-constrained 4×H100 testbed. Without it, migration is *slower* than load balancing alone (1.09× vs. 1.11×) because when KV caches are full, migrations add marginal benefits but incur overheads from contention. Adding the KV veto prevents these unnecessary migrations, increasing the overall speedup to 1.14×.[1]

[1]Speedups differ from end-to-end results as the ablation sampled a fresh set of rollouts with different lengths.

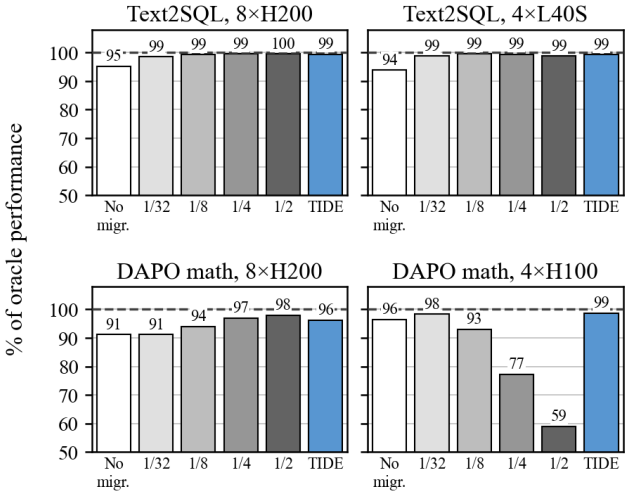



*Figure 9.* Simulated performance of fixed-$B$ policies and TIDE's adaptive policy, relative to an oracle that picks the best $B$ every step (higher is better). Fixed policies set $B$ to the given fraction of the initial per-GPU batch size. The best fixed $B$ differs across workloads, and a poor choice costs much more on some (DAPO Math) than others (Text2SQL); TIDE is comparable to the best fixed policy without manual tuning.

### 6.4 Policy Study

We study how effective TIDE's controller is in finding the optimal migration threshold. To quickly search a wide range of policies, we use ServingStudio (Zhu et al., 2026) to simulate different migration policies. In simulation, we replay a collected trace using every possible $B$ value (§4.1). We define the oracle policy as one that uses the best $B$ value at each step, and compare various fixed policies and TIDE's controller to this oracle.

Figure 9 shows the results. While at least one fixed policy can achieve most of the oracle's performance, the best fixed policy varies by workload. TIDE's adaptive policy provides performance comparable to that of the best fixed policy across all the workloads we evaluate without manual configuration.

## 7 Future Work

TIDE was evaluated on single node configurations with varying capability levels of hardware. Meanwhile, hardware innovation continues to move at break-neck speed, from NVIDIA's NVL72 clusters, in which much larger amounts of GPUs are connected via high-speed NVLink interconnects (NVIDIA, 2026a;b), to wafer-scale chips like Cerebras' CS-4 systems (Yeung & Gardner, 2026). The hardware that is used changes key factors, such as the ratio of inference to training throughput, which affects the optimal migration time. Thus, adaptive policies such as those intro-

duced by TIDE are critical for adapting to new hardware. Additionally, when operating at scale, stragglers have even more impact, offering larger opportunities for TIDE to reclaim GPU-hours.

TIDE is also complementary to other techniques that accelerate the long tail in RL, such as speculative decoding. While TIDE migration enlarges batch sizes, speculative decoding can realize further gains, especially when migration is no longer useful and at small batch sizes.

The ideas in TIDE extend beyond on-policy, colocated deployments. They could be used to improve the performance of disaggregated deployments where the number of GPUs in the inference and training pools is traditionally static, such as a one-step overlap async RL policy; in this deployment, even with the overlapping of training and inference, the optimal split of inference and training pools changes over time in order to keep the GPUs fully utilized. The flexible allocation in TIDE can translate to help manage this split dynamically while preserving the semantics. In other deployments, such as fully-async RL, the GPUs are able to be kept at much higher utiilization in both the inference and training pools; thus, while there might be some benefits from flexibly allocating inference and training pools, we expect there to be less systems-level benefit from TIDE.

## 8 CONCLUSION

In this paper, we identified and quantified two main sources of GPU underutilization in colocated on-policy training: idle GPUs waiting for straggler rollouts (*idle loss*) and inefficient decoding from low batch sizes (*decode inefficiency loss*). We presented TIDE, a system that improves GPU utilization in synchronous colocated RL while preserving on-policy training semantics. TIDE overlaps training with ongoing rollouts by repurposing GPUs and consolidates remaining rollouts through request migration. An adaptive migration policy balances inference and training without workload-specific tuning. Across diverse workloads, hardware configurations, and training backends, TIDE achieves up to 1.36× end-to-end speedup and up to 97% of utilization-optimal throughput.

## A Case Study

To examine the behavior of our system under a real workload, we take DAPO Math run on the 8xH200 setting as a case study. Figure 10 shows one step of that run on every GPU under colocated and TIDE. Each inference engine requires 1 GPU (resulting in 8 inference engines), while each training engine uses 2 GPUs (TP=2, resulting in 4 training engine data-parallel groups). We see how, unlike the colocated, TIDE is able to fill in the gaps that rollout idle time with training. Under colocated execution, 20.8% of the step's GPU-time, is spent idle. In contrast, TIDE fires three migrations progressively as batch sizes decrease across the inference engines(at 249, 271 and 341 s, moving 80, 80 and 88 requests), while flipping the now-idle GPUs to training. As a result, idle step time falls to 3.2% of GPU-time. Additionally, we see that most of the remaining idle GPU hours is the result of training ranks waiting for the stragglers trainers to finish on their samples.

We also see that the decode throughput increased as a result of the migration. Per active engine, decode throughput rose from 2,766 to 2,897 tok/s, because after the migrations the tail is decoded at a larger batch on two engines rather than at a batch of a few requests on eight. Counting all GPU-time spent in inference mode, the rollout costs 1.14 GPU-hours under colocated execution but 0.80 GPU-hours under TIDE. With these two improvements, our system allows the step to complete in 595 s as opposed to 746 s.

These trends generalize to the entire run as well; over the 20 steps, idle time drops from 15.2% to 4.8% of GPU-time and active decoding from 15.48 to 14.82 GPU-hours (−4.3%).

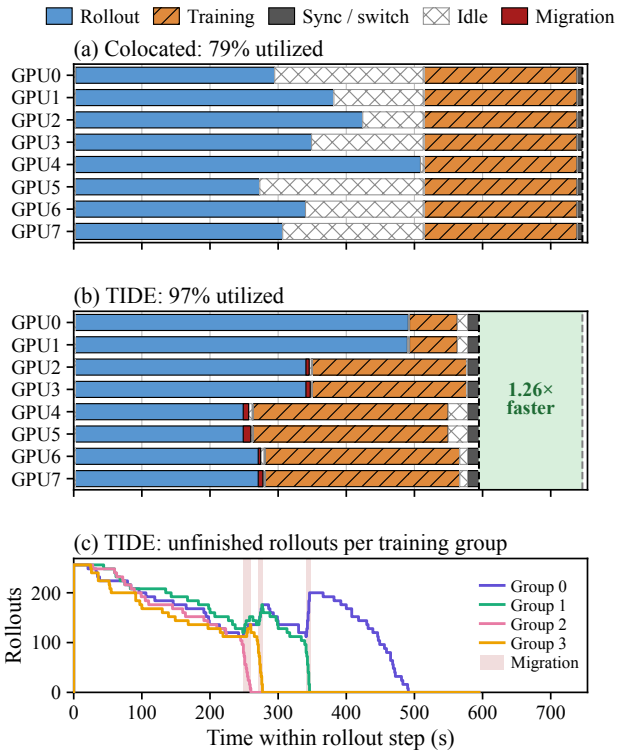



*Figure 10.* Per-GPU timeline of a sampled rollout step under the colocated baseline (top) and TIDE (bottom). As batch sizes shrink during the rollout tail, TIDE consolidates in-flight requests onto fewer engines and switches the freed GPUs to training. This removes most of the idle time that colocation leaves and shortens the step.

## References


Brakmo, L. S., O'Malley, S. W., and Peterson, L. L. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pp. 24–35, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916824. doi: 10.1145/190314.190317. URL https://doi.org/10.1145/190314.190317.

Cao, S., Li, D., Zhao, F., Yuan, S., Hegde, S. R., Chen, C., Ruan, C., Griggs, T., Liu, S., Tang, E., Liaw, R., Moritz, P., Zaharia, M., Gonzalez, J. E., and Stoica, I. SkyRL-Agent: Efficient RL Training for Multi-Turn LLM Agent, 2025. URL https://arxiv.org/abs/2511.16108.

Chen, Q., Liu, Z., Sun, P., Li, S., Wang, G., Liu, Z., Wen, Y., Feng, S., and Zhang, T. ReSpec: Towards Optimizing Speculative Decoding in Reinforcement Learning Systems. In *Proceedings of the 9th Annual Conference on Machine Learning and Systems*, MLSys '26, 2026. doi: 10.48550/arXiv.2510.26475. arXiv:2510.26475.

Fu, W., Gao, J., Shen, X., Zhu, C., Mei, Z., He, C., Xu, S., Wei, G., Mei, J., Wang, J., Yang, T., Yuan, B., and Wu, Y. AReaL: A Large-Scale Asynchronous Reinforcement Learning System for Language Reasoning, 2025. URL https://arxiv.org/abs/2505.24298.

Gao, W., Zhao, Y., An, D., Wu, T., Cao, L., Xiong, S., Huang, J., Wang, W., Yang, S., Su, W., Wang, J., Qu, L., Zheng, B., and Wang, W. RollPacker: Taming Long-Tail Rollouts for RL Post-Training with Tail Batching. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, pp. 849–866, Renton, WA, May 2026. USENIX Association. ISBN 978-1-939133-54-0. URL https://www.usenix.org/conference/nsdi26/presentation/gao-wei.

Guo, D., Yang, D., Zhang, H., Song, J., Wang, P., Zhu, Q., Xu, R., Zhang, R., Ma, S., Bi, X., Zhang, X., Yu, X., Wu, Y., Wu, Z. F., Gou, Z., Shao, Z., Li, Z., Gao, Z., Liu, A.,

Xue, B., Wang, B., Wu, B., Feng, B., Lu, C., Zhao, C., Deng, C., Ruan, C., Dai, D., Chen, D., Ji, D., Li, E., Lin, F., Dai, F., Luo, F., Hao, G., Chen, G., Li, G., Zhang, H., Xu, H., Ding, H., Gao, H., Qu, H., Li, H., Guo, J., Li, J., Chen, J., Yuan, J., Tu, J., Qiu, J., Li, J., Cai, J. L., Ni, J., Liang, J., Chen, J., Dong, K., Hu, K., You, K., Gao, K., Guan, K., Huang, K., Yu, K., Wang, L., Zhang, L., Zhao, L., Wang, L., Zhang, L., Xu, L., Xia, L., Zhang, M., Zhang, M., Tang, M., Zhou, M., Li, M., Wang, M., Li, M., Tian, N., Huang, P., Zhang, P., Wang, Q., Chen, Q., Du, Q., Ge, R., Zhang, R., Pan, R., Wang, R., Chen, R. J., Jin, R. L., Chen, R., Lu, S., Zhou, S., Chen, S., Ye, S., Wang, S., Yu, S., Zhou, S., Pan, S., Li, S. S., Zhou, S., Wu, S., Yun, T., Pei, T., Sun, T., Wang, T., Zeng, W., Liu, W., Liang, W., Gao, W., Yu, W., Zhang, W., Xiao, W. L., An, W., Liu, X., Wang, X., Chen, X., Nie, X., Cheng, X., Liu, X., Xie, X., Liu, X., Yang, X., Li, X., Su, X., Lin, X., Li, X. Q., Jin, X., Shen, X., Chen, X., Sun, X., Wang, X., Song, X., Zhou, X., Wang, X., Shan, X., Li, Y. K., Wang, Y. Q., Wei, Y. X., Zhang, Y., Xu, Y., Li, Y., Zhao, Y., Sun, Y., Wang, Y., Yu, Y., Zhang, Y., Shi, Y., Xiong, Y., He, Y., Piao, Y., Wang, Y., Tan, Y., Ma, Y., Liu, Y., Guo, Y., Ou, Y., Wang, Y., Gong, Y., Zou, Y., He, Y., Xiong, Y., Luo, Y., You, Y., Liu, Y., Zhou, Y., Zhu, Y. X., Huang, Y., Li, Y., Zheng, Y., Zhu, Y., Ma, Y., Tang, Y., Zha, Y., Yan, Y., Ren, Z. Z., Ren, Z., Sha, Z., Fu, Z., Xu, Z., Xie, Z., Zhang, Z., Hao, Z., Ma, Z., Yan, Z., Wu, Z., Gu, Z., Zhu, Z., Liu, Z., Li, Z., Xie, Z., Song, Z., Pan, Z., Huang, Z., Xu, Z., Zhang, Z., and Zhang, Z. DeepSeek-R1 Incentivizes Reasoning in LLMs through Reinforcement Learning. *Nature*, 645(8081):633–638, Sept 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

He, J., Li, T., Feng, E., Du, D., Liu, Q., Liu, T., Xia, Y., and Chen, H. History doesn't repeat itself but rollouts rhyme: Accelerating reinforcement learning with rhymerl. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26, pp. 929–945, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400723599. doi: 10.1145/3779212.3790172. URL https://doi.org/10.1145/3779212.3790172.

Hu, Q., Yang, S., Guo, J., Yao, X., Lin, Y., Gu, Y., Cai, H., Gan, C., Klimovic, A., and Han, S. Taming the Long-Tail: Efficient Reasoning RL Training with Adaptive Drafter. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '26. Association for Computing Machinery, 2026. doi: 10.1145/3779212.3790231.

Kang, S., Kang, H., Suresh, T., Kim, Y., Pimpalgaonkar, S., Hwang, S. J., and Mirhoseini, A. ThunderSyncRL: Lossless Acceleration of Agentic Reinforcement Learning, 2026. URL https://arxiv.org/abs/2610.05935.

Kingma, D. P. and Ba, J. Adam: A Method for Stochastic Optimization, 2017. URL https://arxiv.org/abs/1412.6980.

Krell, M. M., Kosec, M., Perez, S. P., and Fitzgibbon, A. Efficient Sequence Packing without Cross-Contamination: Accelerating Large Language Models without Impacting Performance, 2021.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Liu, S., Zhu, A., Hegde, S., Cao, S., Yuan, S., Suwito, S., Griggs, T., Zaharia, M., Gonzalez, J. E., and Stoica, I. SkyRL-SQL: Multi-Turn SQL Data Agents via RL. In *First Workshop on Multi-Turn Interactions in Large Language Models*, 2025. URL https://openreview.net/forum?id=VNPGUGbC1p.

Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., and Stoica, I. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 561–577, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/moritz.

NVIDIA. NVIDIA GB200 NVL72. https://www.nvidia.com/en-us/data-center/gb200-nvl72/, August 2026a.

NVIDIA. NVIDIA GB300 NVL72. https://www.nvidia.com/en-us/data-center/gb300-nvl72/, July 2026b.

Olmo, T., :, Ettinger, A., Bertsch, A., Kuehl, B., Graham, D., Heineman, D., Groeneveld, D., Brahman, F., Timbers, F., Ivison, H., Morrison, J., Poznanski, J., Lo, K., Soldaini, L., Jordan, M., Chen, M., Noukhovitch, M., Lambert, N., Walsh, P., Dasigi, P., Berry, R., Malik, S., Shah, S., Geng, S., Arora, S., Gupta, S., Anderson, T., Xiao, T., Murray, T., Romero, T., Graf, V., Asai, A., Bhagia, A., Wettig, A., Liu, A., Rangapur, A., Anastasiades, C., Huang, C.,

Schwenk, D., Trivedi, H., Magnusson, I., Lochner, J., Liu, J., Miranda, L. J. V., Sap, M., Morgan, M., Schmitz, M., Guerquin, M., Wilson, M., Huff, R., Bras, R. L., Xin, R., Shao, R., Skjonsberg, S., Shen, S. Z., Li, S. S., Wilde, T., Pyatkin, V., Merrill, W., Chang, Y., Gu, Y., Zeng, Z., Sabharwal, A., Zettlemoyer, L., Koh, P. W., Farhadi, A., Smith, N. A., and Hajishirzi, H. Olmo 3, 2026. URL https://arxiv.org/abs/2512.13961.

OpenAI, :, Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., Helyar, A., Madry, A., Beutel, A., Carney, A., Iftimie, A., Karpenko, A., Passos, A. T., Neitz, A., Prokofiev, A., Wei, A., Tam, A., Bennett, A., Kumar, A., Saraiva, A., Vallone, A., Duberstein, A., Kondrich, A., Mishchenko, A., Applebaum, A., Jiang, A., Nair, A., Zoph, B., Ghorbani, B., Zhang, B., Rossen, B., Sokolowsky, B., Barak, B., McGrew, B., Minaiev, B., Hao, B., Baker, B., Houghton, B., McKinzie, B., Eastman, B., Lugaresi, C., Bassin, C., Hudson, C., Li, C. M., de Bourcy, C., Voss, C., Shen, C., Zhang, C., Koch, C., Orsinger, C., Hesse, C., Fischer, C., Chan, C., Roberts, D., Kappler, D., Levy, D., Selsam, D., Dohan, D., Farhi, D., Mely, D., Robinson, D., Tsipras, D., Li, D., Oprica, D., Freeman, E., Zhang, E., Wong, E., Proehl, E., Cheung, E., Mitchell, E., Wallace, E., Ritter, E., Mays, E., Wang, F., Such, F. P., Raso, F., Leoni, F., Tsimpourlas, F., Song, F., von Lohmann, F., Sulit, F., Salmon, G., Parascandolo, G., Chabot, G., Zhao, G., Brockman, G., Leclerc, G., Salman, H., Bao, H., Sheng, H., Andrin, H., Bagherinezhad, H., Ren, H., Lightman, H., Chung, H. W., Kivlichan, I., O'Connell, I., Osband, I., Gilaberte, I. C., Akkaya, I., Kostrikov, I., Sutskever, I., Kofman, I., Pachocki, J., Lennon, J., Wei, J., Harb, J., Twore, J., Feng, J., Yu, J., Weng, J., Tang, J., Yu, J., Candela, J. Q., Palermo, J., Parish, J., Heidecke, J., Hallman, J., Rizzo, J., Gordon, J., Uesato, J., Ward, J., Huizinga, J., Wang, J., Chen, K., Xiao, K., Singhal, K., Nguyen, K., Cobbe, K., Shi, K., Wood, K., Rimbach, K., Gu-Lemberg, K., Liu, K., Lu, K., Stone, K., Yu, K., Ahmad, L., Yang, L., Liu, L., Maksin, L., Ho, L., Fedus, L., Weng, L., Li, L., McCallum, L., Held, L., Kuhn, L., Kondraciuk, L., Kaiser, L., Metz, L., Boyd, M., Trebacz, M., Joglekar, M., Chen, M., Tintor, M., Meyer, M., Jones, M., Kaufer, M., Schwarzer, M., Shah, M., Yatbaz, M., Guan, M. Y., Xu, M., Yan, M., Glaese, M., Chen, M., Lampe, M., Malek, M., Wang, M., Fradin, M., McClay, M., Pavlov, M., Wang, M., Wang, M., Murati, M., Bavarian, M., Rohaninejad, M., McAleese, N., Chowdhury, N., Chowdhury, N., Ryder, N., Tezak, N., Brown, N., Nachum, O., Boiko, O., Murk, O., Watkins, O., Chao, P., Ashbourne, P., Izmailov, P., Zhokhov, P., Dias, R., Arora, R., Lin, R., Lopes, R. G., Gaon, R., Miyara, R., Leike, R., Hwang, R., Garg, R., Brown, R., James, R., Shu, R., Cheu, R., Greene, R., Jain, S., Altman, S., Toizer, S., Toyer, S., Miserendino, S., Agarwal, S., Hernandez, S., Baker, S., McKinney, S., Yan, S., Zhao, S., Hu, S., Santurkar, S., Chaudhuri, S. R., Zhang, S., Fu, S., Papay, S., Lin, S., Balaji, S., Sanjeev, S., Sidor, S., Broda, T., Clark, A., Wang, T., Gordon, T., Sanders, T., Patwardhan, T., Sottiaux, T., Degry, T., Dimson, T., Zheng, T., Garipov, T., Stasi, T., Bansal, T., Creech, T., Peterson, T., Eloundou, T., Qi, V., Kosaraju, V., Monaco, V., Pong, V., Fomenko, V., Zheng, W., Zhou, W., Zhan, W., McCabe, W., Zaremba, W., Dubois, Y., Lu, Y., Chen, Y., Cha, Y., Bai, Y., He, Y., Zhang, Y., Wang, Y., Shao, Z., and Li, Z. OpenAI o1 System Card, 2026. URL https://arxiv.org/abs/2412.16720.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. ZeRO-Offload: Democratizing Billion-Scale Model Training, 2021. URL https://arxiv.org/abs/2101.06840.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal Policy Optimization Algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *arXiv preprint arXiv:2402.03300*, 2024. doi: 10.48550/arXiv.2402.03300.

Shao, Z., Srivatsa, V., Srivastava, S., Wu, Q., Ariyak, A., Wu, X., Patel, A., Wang, J., Liang, P., Dao, T., Zhang, C., Zhang, Y., Athiwaratkun, B., Xu, C., and Wang, J. Beat the Long Tail: Distribution-Aware Speculative Decoding for RL Training. In *Proceedings of the 9th Annual Conference on Machine Learning and Systems*, MLSys '26, 2026. doi: 10.48550/arXiv.2511.13841. arXiv:2511.13841.

Sheng, G., Zhang, C., Ye, Z., Wu, X., Zhang, W., Zhang, R., Peng, Y., Lin, H., and Wu, C. HybridFlow: A Flexible and Efficient RLHF Framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297, 2025.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

Wang, W., Zhou, Y., Chen, M., Zhang, S., Zhang, Y., Luo, Y., Yin, Z., Wu, C., Jiang, J., Yao, W., Zheng, C., and Zhang, J. QwenGyre: An Elastic Reinforcement Learning Framework for Training xLong-Horizon Agents, 2026. URL https://arxiv.org/abs/2609.33848.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing Reasoning and Acting in Language Models, 2023. URL https://arxiv.org/abs/2210.03629.

Yeung, A. and Gardner, E. Introducing Cerebras CS-4: The Fastest AI Just Got Faster. Cerebras Systems Blog, https://www.cerebras.ai/blog/introducing-cerebras-cs-4, August 2026.

Yu, Q., Zhang, Z., Zhu, R., Yuan, Y., Zuo, X., Yue, Y., Dai, W., Fan, T., Liu, G., Liu, L., Liu, X., Lin, H., Lin, Z., Ma, B., Sheng, G., Tong, Y., Zhang, C., Zhang, M., Zhang, W., Zhu, H., Zhu, J., Chen, J., Chen, J., Wang, C., Yu, H., Song, Y., Wei, X., Zhou, H., Liu, J., Ma, W.-Y., Zhang, Y.-Q., Yan, L., Qiao, M., Wu, Y., and Wang, M. DAPO: An Open-Source LLM Reinforcement Learning System at Scale, 2025. URL https://arxiv.org/abs/2503.14476.

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel, 2023. URL https://arxiv.org/abs/2304.11277.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. SGLang: Efficient Execution of Structured Language Model Programs. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NeurIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc. ISBN 9798331314385.

Zhong, Y., Zhang, Z., Song, X., Hu, H., Jin, C., Wu, B., Chen, N., Chen, Y., Zhou, Y., Wan, C., Zhou, H., Jiang, Y., Zhu, Y., and Jiang, D. StreamRL: Scalable, Heterogeneous, and Elastic RL for LLMs with Disaggregated Stream Generation, 2025a. URL https://arxiv.org/abs/2504.15930.

Zhong, Y., Zhang, Z., Wu, B., Liu, S., Chen, Y., Wan, C., Hu, H., Xia, L., Ming, R., Zhu, Y., and Jin, X. Optimizing RLHF Training for Large Language Models with Stage Fusion. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '25, USA, 2025b. USENIX Association. ISBN 978-1-939133-46-5.

Zhu, K., Gu, M., Sriram, S., Jacob, M., Kamahori, K., Li, V., Pan, Y., Xie, D., Wang, S., Krishnamurthy, A., and Kasikci, B. Introducing ServingStudio: An Integrated Workbench for Simulating, Analyzing, and Optimizing LLM Serving Systems. https://syfi.cs.washington.edu/blog/2026-09-24-introducing-servingstudio/, September 2026. Code: https://github.com/SyFI-ServingStudio/ServingStudio.

Zhu, Z., Xie, C., Lv, X., and slime Contributors. slime: An LLM Post-Training Framework for RL Scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Corresponding author: Xin Lv.

## AI USAGE DISCLOSURE

The authors used coding agents (Claude Code) to implement parts of the system described, and to perform data analysis on exploratory features. The authors also used LLMs (Gemini, Claude) to perform literature search. The authors used coding agents to generate figures and evaluation code. The authors verified the correctness of the outputs and code. Additionally, the authors used large language models to suggest copy editing and formatting suggestions to the paper text.