ASYNCHRONOUS LLM REINFORCEMENT LEARNING UNDER LIMITED DATA AND CONSTRAINED HARDWARE PERFORMANCE

Owen Zeng [1] **Zongchi Xie** [1]

ABSTRACT

Recent work such as AReaL shows that asynchronous rollout and training can improve PPO-based LLM reinforcement learning by overlapping rollout generation and model optimization, but these results are demonstrated at large scale and may not transfer to constrained settings with limited data and hardware. In addition, prior systems rely on inference backends such as vLLM and SGLang, which depend on modern GPU features (e.g., BF16 and optimized attention kernels) that are not well supported on V100 GPUs. In this work, we implement a lightweight asynchronous RL system following AReaL's design from scratch on 2×V100 GPUs and study its system performance under limited overlap and hardware constraints, using a parameter-service policy clock, bounded-staleness filtering, and matched sync and async baselines. Across our constrained runs, bounded staleness serves as an effective control knob, exposing a quality–throughput trade-off: stricter freshness reduces accepted throughput, while moderate staleness can improve short-horizon learning quality relative to both sync and threaded async baselines. However, asynchronous execution does not consistently improve end-to-end runtime, and the decoupled objective underperforms naive PPO in this regime. Overall, our results show that core asynchronous RL mechanisms transfer to constrained settings, but their system-level benefits are highly dependent on scale and available overlap.

1 INTRODUCTION

Standard PPO-based reinforcement learning pipelines for Large Language Models are typically executed in a fully synchronous manner. Rollout workers first generate a batch of trajectories, and policy updates are performed only after all samples are collected. This creates a strict barrier between data collection and optimization, leading to poor hardware utilization: rollout workers remain idle during training, while the trainer is idle during generation. In practice, overall system throughput becomes bottlenecked by the slowest sample in each batch.

Recent work suggests that a strict synchronous system is not necessary. Methods such as AReaL improve efficiency by overlapping rollout generation and policy optimization through asynchronous decoupling of rollout generation and policy upgrading. However, these approaches are developed and evaluated at large scale, often relying on modern GPU features and inference backends such as vLLM and SGLang. It remains unclear whether the same benefits hold in smaller, resource-constrained environments, where older hardware such as V100 GPUs lacks support for BF16 and optimized attention kernels, and where limited parallelism restricts overlap between rollout and training.

In this work, we study asynchronous PPO-based LLM reinforcement learning under limited data and constrained hardware. We implement a lightweight asynchronous pipeline inspired by AReaL, where rollout workers continuously generate samples and place them into a shared buffer, while the trainer independently consumes accepted trajectories to perform policy updates. To support this setting, we incorporate policy version tracking and bounded staleness control, allowing us to explicitly examine the trade-off between system efficiency and training behavior in the presence of stale samples.

Our goal is to understand which aspects of asynchronous RL remain effective under constrained settings, and how system-level performance changes when scale, hardware capabilities, and available overlap are limited.

2 RELATED WORK

PPO and on-policy RL. Proximal Policy Optimization (PPO) stabilizes policy-gradient training through a clipped surrogate objective and is widely used for RL-based LLM post-training (Schulman et al., 2017). From a systems perspective, PPO is most straightforward to implement in a fully synchronous, on-policy setting, where all trajectories are generated by the current policy before each update. While this simplifies reasoning about training, it also leads to inefficient hardware usage due to the strict separation between rollout and optimization phases.

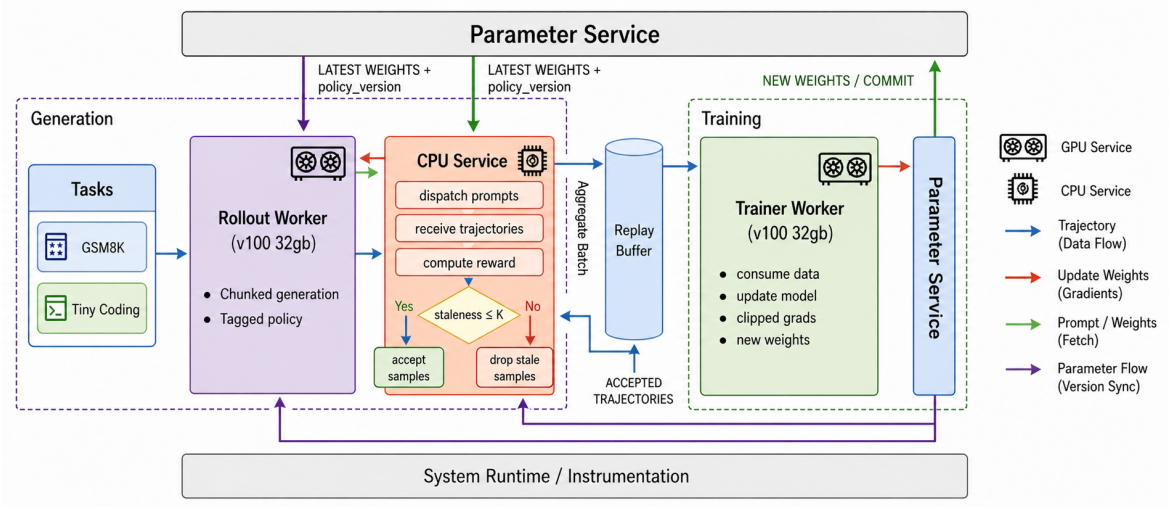

Figure 1: AReaL-style system. Rollout generation and training are decoupled through a controller and replay buffer, with bounded staleness control.

Actor–learner decoupling. Separating actors from learners has been explored well before LLM RL. Systems such as IMPALA show that allowing many actors to generate data asynchronously while a learner updates the model can significantly improve throughput (Espeholt et al., 2018). However, this comes with a trade-off: asynchrony introduces policy lag, and using stale data can lead to off-policy drift. This balance between efficiency and training stability is a central challenge in asynchronous RL systems.

AReaL and stale rollout data. AReaL extends actor–learner decoupling to LLM reinforcement learning by fully separating rollout generation from policy optimization. Instead of alternating between synchronized phases, rollout workers continuously generate trajectories using a local copy of the policy, while a trainer independently updates the model using previously collected samples. To make this work in practice, AReaL introduces a policy version clock to track which parameters were used to generate each sample, along with bounded staleness filtering to control how outdated a trajectory can be before it is discarded. It also supports interruptible rollouts, allowing workers to switch to newer policy versions without waiting for long generations to finish. Together, these mechanisms aim to improve hardware utilization by increasing overlap between generation and training, while still maintaining sufficient control over policy lag to preserve stable learning (Fu et al., 2025).

How our work differs. Rather than reproducing AReaL's large-scale results, we focus on a more constrained and practical setting: a 0.5B model running on 2×V100-32GB GPUs, with limited parallelism and without support for BF16 or modern attention kernels. In this setting, system bottlenecks are different, and the benefits of asynchrony might be less obvious. Our goal is to understand which components of asynchronous RL remain effective under these constraints, what engineering changes are required to make the system stable, and which results from large-scale systems do not carry over.

3 Methodology

Our implementation is designed around the constraints of commodity V100 GPUs, which lack support for modern features such as BF16 precision and optimized attention kernels. These limitations introduce several practical challenges. First, CUDA multiprocessing must be handled carefully to avoid instability during worker initialization. Second, we separate model initialization from measured runtime to ensure that results reflect actual rollout and training behavior rather than cold-start overhead. Finally, we instrument the system with queue statistics, policy-version tracking, and staleness metrics to verify that asynchronous overlap is realized in practice rather than only in code structure.

Our system follows an AReaL-style asynchronous design that decouples rollout generation from policy optimization (Figure 1). The architecture consists of a rollout worker, a CPU-based controller, a replay buffer, and a trainer worker, all coordinated through a parameter service that maintains the global model state and policy version.

The rollout worker runs on `cuda:0` and continuously generates trajectories from input tasks. Each trajectory is tagged with the policy version used for generation, allowing the system to track policy lag. The worker periodically fetches updated weights from the parameter service and reloads them at safe boundaries.

The controller coordinates the system by dispatching prompts, receiving trajectories, computing rewards, and filtering samples. To control policy lag, it computes the staleness of each trajectory relative to the current policy version and applies bounded staleness filtering. Samples with staleness less than or equal to a threshold K are accepted, while stale samples are discarded. Accepted samples are batched and placed into the replay buffer.

When a rollout is interrupted by a version change, the controller may resubmit the same source task up to a bounded retry limit before finally marking it as dropped, preventing interruption events from being conflated with stale-sample filtering.

The replay buffer acts as an asynchronous interface between generation and training, decoupling the rate of rollout production from consumption. This allows the trainer to operate continuously without blocking on generation.

The trainer worker runs on `cuda:1` and continuously consumes batches from the replay buffer to perform PPO updates, including gradient computation, clipping, and weight updates. Updated parameters are sent back to the parameter service, which serializes commits, writes the new trainable state into shared memory, and increments the global policy version. Rollout and trainer workers then reload the new state when they observe that the global version has advanced.

Formally, for a trajectory generated under policy version v_{rollout}, the controller computes

$$\text{staleness} = v_{\text{current}} - v_{\text{rollout}},$$

and accepts the sample only if staleness $\leq K$. The staleness bound K is the primary control knob: smaller values enforce fresher data but discard more samples, while larger values improve utilization at the cost of increased policy lag.

The staleness bound K serves as the primary control knob: smaller values enforce stricter on-policy behavior but lead to more discarded samples, while larger values improve utilization at the cost of increased policy lag.

We use a trainable Hugging Face causal language model as the policy backbone. During training, the backend supports two update variants under the same runtime: a clipped-ratio objective that uses stored rollout log-probabilities (`decoupled_objective=1`) and a simpler policy-gradient baseline without ratio correction (`decoupled_objective=0`). This allows us to test whether the more AReaL-style update path remains beneficial under constrained hardware and limited overlap.

To ensure reliable measurement, we adopt several engineering practices. We use spawn-based multiprocessing so each worker initializes its own CUDA context, avoiding instability from forking. We introduce a worker-ready barrier to separate cold-start overhead from measured runtime. Finally, we log queue depths, policy versions, staleness distributions, and sample acceptance rates to verify that asynchronous overlap is realized in practice and to diagnose system bottlenecks.

4 Implementation Details and Code Snippets

We build three matched runners—`sync_train`, `async_train`, and `async_areal_style`—that share the same task loader, reward harness, tokenizer path, summary format, and plotting pipeline, so the main difference is how rollout and training are scheduled and overlapped.

4.1 System Runtime

The AReaL style runtime is implemented as a system with a controller, rollout worker, trainer worker, and parameter service. The main process launches child processes using Python's `spawn` context. The runtime maintains a shared trainable state dictionary, a global `policy_version`, rollout and trainer queues, and an event queue for queue-depth and version change tracing.

We implemented each component needed for the system. The `src/areal_runtime.py` module handles overall process orchestration and synchronization between workers, while `src/areal_controller.py` manages reward computation, applies the staleness filter, and dispatches samples to the replay pipeline. Rollout generation is handled by `src/areal_rollout_worker.py`, which also reloads updated policy versions, and training is performed by `src/areal_trainer_worker.py`, which executes PPO updates. The `src/model_backends.py` module provides the underlying Hugging Face implementation and update logic, and `src/run_experiment_grid.py` runs experiments and supports resuming runs.

Within this architecture, the controller is responsible for the core coordination logic. It performs prompt dispatch, reward evaluation, bounded-staleness filtering, replay buffering, and trainer-batch dispatch. The parameter service is the only writer of the global trainable state, which makes the policy clock explicit and prevents concurrent state corruption.

4.2 Bounded-Staleness Control

The bounded-staleness logic is implemented in `src/areal_controller.py`. For each emitted trajectory, the controller compares the current global policy version against the version used to generate the sample and decides whether to keep or drop it. The rule is lightweight

and sits directly on the critical path:

Listing 1: Actual controller-side bounded-staleness filtering and replay-batch dispatch.

```
reward_info = evaluate_response(
   response_text=sample["response"],
   task=task_obj,
   timeout_sec=args.reward_timeout_sec,
)
staleness = max(0, current_global_version - int(sample[
    "policy_version"]))
dropped = staleness > args.staleness_k

row = {
   "sample_id": sample_id,
   "task_id": task_id,
   "policy_version": int(sample["policy_version"]),
   "current_global_policy_version":
       current_global_version,
   "staleness": int(staleness),
   "dropped": bool(dropped),
   "reward": float(reward_info["reward"]) if not
       dropped else 0.0,
   "pass": bool(reward_info["pass"]) if not dropped
       else False,
   "response": sample["response"],
   "metadata": dict(sample.get("metadata", {})),
   "logprob_sum": float(sample.get("logprob_sum", 0.0))
       ,
}

if dropped:
   dropped_rows.append(row)
else:
   accepted_rows.append(row)
   replay_buffer.append(row)

while len(replay_buffer) >= args.update_batch_size:
   batch_rows = replay_buffer[: args.update_batch_size]
   replay_buffer = replay_buffer[args.update_batch_size
       :]
   trainer_id = _dispatch_training_batch(
      trainer_queues=trainer_queues,
      trainer_rr_idx=trainer_rr_idx,
      batch_id=batch_id,
      batch_rows=batch_rows,
   )
```

This controller rule is the main systems knob in the paper. Smaller K enforces fresher data but wastes more rollout work; larger K improves accepted throughput but increases policy lag.

4.3 Rollout and Trainer Workers

The rollout worker builds its local backend on the assigned device, tags each emitted sample with the current `policy_version`, and checks whether the global version has advanced before continuing generation. The trainer worker continuously consumes accepted replay batches and computes a local policy update. Importantly, the trainer does not directly mutate the global state. Instead, it submits the candidate updated state and associated metadata to a separate parameter service, which is the only component allowed to commit a new global policy version.

4.4 Parameter Service and Policy-Version Commit

A key systems mechanism in our runtime is that trainer updates are committed centrally in `src/areal_parameter_service.py`. This process owns the global trainable state and the policy clock. When a trainer finishes computing an update, it sends the candidate state to the parameter service. If the update is valid, the parameter service writes the new state into shared memory, increments the global `policy_version`, and emits a commit record. If no update is applied, it records a skipped commit without advancing the version.

Listing 2: Parameter service serializes trainer commits and advances the global policy version.

```
if not update_info.get("updated", False):
   commit_queue.put(
      {
         "type": "update_commit",
         "trainer_worker_id": trainer_worker_id,
         "batch_id": batch_id,
         "sample_ids": sample_ids,
         "updated": False,
         "policy_version": int(policy_version_value.
             value),
      }
   )
   continue

shared_state["trainable_state"] = new_state
old_version = int(policy_version_value.value)
policy_version_value.value += 1
global_update_count_value.value += 1

_emit_event(
   event_queue,
   {
      "type": "version_change",
      "trainer_worker_id": int(trainer_worker_id),
      "batch_id": int(batch_id),
      "old_version": int(old_version),
      "new_version": int(policy_version_value.value),
      "timestamp": time.time(),
   },
)

commit_queue.put(
   {
      "type": "update_commit",
      "trainer_worker_id": trainer_worker_id,
      "batch_id": batch_id,
      "sample_ids": sample_ids,
      "updated": True,
      "policy_version": int(policy_version_value.value)
          ,
   }
)
```

This commit path is the concrete realization of the policy clock in our system. By centralizing state mutation in one process, we avoid concurrent writes to shared model state and ensure that staleness is always measured against a single well-defined global version.

4.5 Interruptible Rollout and Version Reload

A core systems feature of the AReaL-style runtime is that rollout workers do not blindly finish long generations under an outdated policy. Instead, each worker periodically checks

the global `policy_version`; when a newer version is detected, the worker interrupts at a safe boundary, reloads the latest trainable state, and resumes future generation under the updated policy. This mechanism is what makes the policy clock operational in practice and helps keep realized staleness bounded during asynchronous execution.

Listing 3: Actual interruptible rollout at chunk boundaries with version reload.

```
global_version = int(policy_version_value.value)
if global_version > local_version:
   interrupt_count += 1
   interrupted_samples += 1
   interrupted = True
   _emit_event(
      event_queue,
      {
         "type": "rollout_interrupt_observed",
         "worker_id": int(worker_id),
         "sample_id": int(sample_id),
         "old_version": int(local_version),
         "new_version": int(global_version),
         "timestamp": time.time(),
      },
   )
   local_backend.load_trainable_state(shared_state.get(
       "trainable_state", {}))
   local_version = global_version
   loaded_versions.append(int(local_version))
   break
```

4.6 Trainable Hugging Face Backend

A key implementation detail is that we use a real, trainable Hugging Face causal language model backend in `src/model_backends.py`, rather than a simplified or toy policy. During rollout, the backend stores the prompt IDs, generated response IDs, and token-level log-probabilities. During training, it recomputes the log-probabilities under the current model, which allows us to correctly apply PPO-style updates.

The training update uses a PPO-style clipped-ratio path when `decoupled_objective=1`, including stored rollout log-probabilities and a non-negative Schulman k3 KL penalty to control policy drift. When `decoupled_objective=0`, the backend instead uses a simpler policy-gradient baseline without ratio correction. This makes it possible to compare a more faithful AReaL-style update path against a simpler baseline under the same runtime and hardware constraints.

Listing 4: Actual policy update in the trainable HF backend.

```
adv_t = torch.full_like(new_token_logprobs, adv)
if self.decoupled_objective:
   old_t = torch.tensor(
      old_logps, dtype=new_token_logprobs.dtype, device
          =self.device
   )
   log_ratio = (new_token_logprobs - old_t).clamp(min
       =-20.0, max=20.0)
   ratio = torch.exp(log_ratio)
   unclipped = ratio * adv_t
   clipped = torch.clamp(ratio, 1.0 - eps, 1.0 + eps) *
        adv_t
   token_loss = -torch.min(unclipped, clipped)
   kl_token = (ratio - 1.0 - log_ratio).clamp(min=0.0,
       max=20.0)
else:
   token_loss = -adv_t * new_token_logprobs
   kl_token = torch.zeros_like(new_token_logprobs)

loss = token_loss.mean() + kl_coef * kl_token.mean()
```

This snippet is important because it directly supports our main empirical finding: under constrained V100 hardware, the clipped-ratio decoupled path does not outperform the simpler baseline in our setting.

4.7 Measurement and Reproducibility

To ensure fair timing, the runtime waits until rollout and trainer workers finish loading their backends before starting the controller wall-clock timer, so cold-start model loading is excluded from measured async runtime. Because the target hardware is 2×V100-32GB rather than A100/H100-class GPUs, the implementation uses FP32 weights with optional FP16 autocast, disables FlashAttention-2, and relies on spawn-based multiprocessing. Each run writes a unified `summary.json`, per-sample results, queue traces, staleness statistics, and config metadata. The full sweep is orchestrated through `src/run_experiment_grid.py` and `scripts/run_final_report.sh`, allowing all figures and tables to be regenerated from one driver script.

5 Experiments

Unless otherwise noted, the experiments in this section use the same model and training configuration across all runners: `Qwen/Qwen2.5-0.5B-Instruct` as the trainable policy, `max_new_tokens=96`, `update_batch_size=4`, matched reward evaluation, and the same prompt/task pipeline. All async comparisons use the same controller and backend logic and differ only in how rollout and training are scheduled.

Across all experiments, asynchronous methods primarily improve training quality and only slightly improved system speed. The learning dynamics in Figure 2 highlight the main qualitative difference. AReaL-style async reaches higher pass rates earlier in training and maintains stronger performance compared to both sync and simple async baselines.

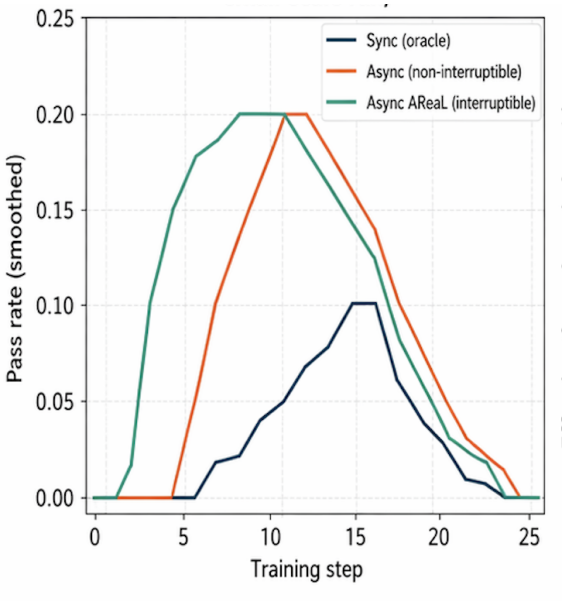

Figure 2: Pass rate over training steps for sync, threaded async, and AReaL-style async.

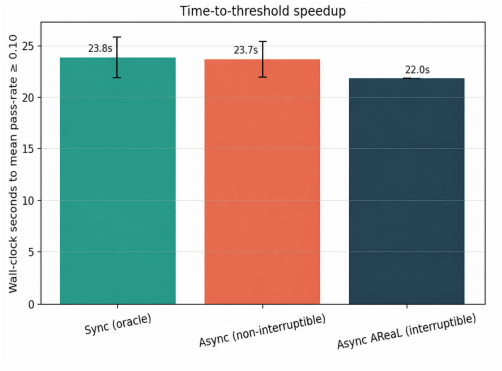

Figure 3: Time required to reach a fixed performance threshold (mean pass rate ≥ 0.10).

The synchronous baseline learns more slowly and peaks at a lower pass rate, indicating weaker overall performance. In contrast, both asynchronous variants improve more quickly, with AReaL-style async achieving strong performance earlier in training. This suggests that asynchronous execution changes the training dynamics, leading to faster early progress rather than simply increasing throughput.

This behavior is further supported by the time-to-threshold results in Figure 3, which measures the wall-clock time required to reach a fixed performance level (mean pass rate ≥ 0.10). AReaL-style async reaches this threshold fastest (22.0s), compared to sync (23.8s) and threaded async (23.7s).

Although the absolute differences are small due to the short training horizon, this result shows that AReaL-style async is more effective at reaching useful performance quickly in real time.

Table 1: Summary of key metrics aligned with Figures 2, 3, and 4.

Mode / config	Pass rate	Time-to-thresh ↓	Sample eff. ↑
Sync	≈ 0.10	23.8	0.104
Async	≈ 0.20	23.7	0.094
Async AReaL	≈ 0.20 **(earlier)**	**22.0**	**0.153**

Finally, Figure 4 measures the total reward per 1k generated tokens. AReaL-style training achieves the highest sample efficiency, indicating that it extracts more useful learning signal from each generated token.

Table 1 summarizes the main trends observed across Figures 2, 3, and 4. The synchronous baseline achieves the lowest overall pass rate and takes the longest time to reach the performance threshold, indicating slower and less effective learning. The threaded async baseline improves the final pass rate compared to sync but does not significantly reduce the time-to-threshold and has lower sample efficiency. In contrast, AReaL-style async reaches similar or slightly better pass rates earlier in training, achieves the fastest time-to-threshold, and has the highest sample efficiency. This highlights the key tradeoff: AReaL does not drastically improve raw speed, but it improves how efficiently the model learns and how quickly it reaches useful performance.

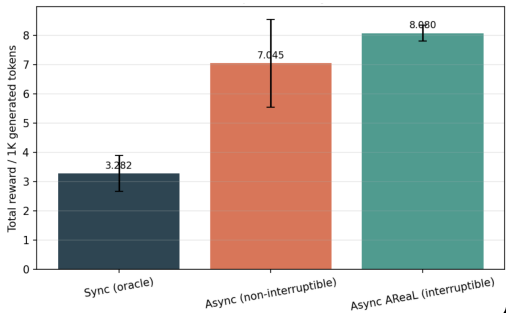

Figure 4: Generation efficiency measured as total reward per 1k generated tokens.

However, we also observe that the variance in sample efficiency is quite large, especially for the AReaL-style async setup. While it achieves the highest average efficiency, the results are less stable across runs, which suggests that these gains are not always consistent. This means the improvements should be interpreted with some caution, rather than as a guaranteed advantage.

Taken together, our results suggest that asynchronous LLM RL is not primarily a speed optimization in small-scale settings. Instead, its main benefit is in changing how the model learns. In particular, it helps the model make faster early progress, use data more effectively, and reach better performance under moderate staleness. At the same time, these benefits come with tradeoffs. The system be-

comes more complex, and the training behavior can be more variable. Overall, this indicates that the effectiveness of asynchronous methods depends heavily on the scale of the system, the amount of parallelism available, and the specific implementation choices.

6 Analysis

Our results show that asynchronous training primarily improves learning efficiency rather than delivering large system-level speedups. As seen in Figure 2, AReaL-style async reaches higher pass rates earlier in training compared to both sync and threaded async baselines, indicating faster early learning progress. This is consistent with Figure 3, where AReaL reaches a fixed performance threshold fastest in wall-clock time. In addition, Figure 4 shows that AReaL achieves the highest sample efficiency, suggesting that it extracts more useful signal from each generated token. Taken together, these results indicate that bounded staleness and asynchronous execution can improve how effectively the model learns from data.

However, these improvements are modest and do not match the large speedups reported in prior work such as AReaL (Fu et al., 2025), which demonstrates up to 2.7× gains. In our setting, the improvements are much smaller, on the order of a few seconds in time-to-threshold and moderate gains in sample efficiency. This gap is primarily due to our constrained experimental setup: we use a 0.5B model, a short training horizon, and only one rollout worker and one trainer worker on V100 GPUs. Under these conditions, system overhead and limited parallelism reduce the benefits of asynchronous execution, preventing large throughput gains.

We also find that moderate staleness improves the best quality operating point. Allowing stale samples into training changes the effective data distribution and enables better short-horizon performance, as seen in the earlier and higher pass rates achieved by AReaL-style async. This suggests that stale samples are not inherently harmful and can, when controlled, improve learning dynamics.

In contrast, not all design choices transfer well. In particular, the decoupled objective underperforms naive PPO in our experiments, often leading to unstable training and reward collapse. This indicates that while the asynchronous system design transfers to small-scale settings, the objective-level improvements are more sensitive to scale, model capacity, and training duration.

Overall, our findings suggest that asynchronous LLM RL should be viewed as a system-level tradeoff rather than a guaranteed speedup. In constrained environments, its primary benefit is improved learning efficiency and earlier convergence to useful performance, rather than large reductions in total runtime. These results highlight that the effectiveness of asynchronous methods depends strongly on hardware scale, worker parallelism, and training regime.

7 Conclusion

We implement and study an AReaL-style asynchronous LLM RL system in a constrained 2×V100 setting to evaluate how well asynchronous training mechanisms transfer to small-scale hardware. Rather than reproducing large-scale results, this work focuses on understanding system behavior under realistic resource limitations.

Our results show that asynchronous execution provides modest but consistent improvements in learning efficiency rather than large system-level speedups. In particular, AReaL-style async achieves higher sample efficiency and reaches useful performance earlier in wall-clock time, indicating faster early-stage learning. We also find that moderate staleness ($k = 2$–4) provides the best quality–efficiency tradeoff, suggesting that controlled staleness can beneficially alter the training dynamics rather than degrade them.

However, these gains are significantly smaller than the large speedups reported in prior work such as AReaL, which demonstrates up to 2.7× improvements at scale. In our constrained setting, limited parallelism, short training horizons, and system overhead reduce the potential benefits of asynchronous execution. Additionally, not all design components transfer: the decoupled objective underperforms naive PPO, highlighting that objective-level improvements are more sensitive to scale and training regime.

Overall, our results suggest that asynchronous LLM RL should be viewed as a system-level tradeoff rather than a guaranteed acceleration. In small-scale environments, its primary benefit is improved learning efficiency and earlier convergence to useful performance, rather than substantial reductions in total runtime. These findings emphasize that the effectiveness of asynchronous methods depends strongly on hardware scale, worker parallelism, and training configuration, and should be evaluated accordingly.

8 Team Member Contributions

- **Owen Zeng:** Worked on the async AReaL system implementation (controller, runtime, trainer pipeline), debugging multiprocessing and policy version flow, and running experiments.
- **Zongchi Xie:** Worked on the model backend, data pipeline, experiment setup, debugging training behavior, and analyzing results.
- **Both:** Worked together on the report, figures, and poster.

REFERENCES

Espeholt, L., Soyer, H., Munos, R., et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, 2018.

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. *arXiv preprint arXiv:2505.24298*, 2025.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.