AstraFlow: Dataflow-Oriented Reinforcement Learning for Agentic LLMs

Haizhong Zheng[1] **Yizhuo Di**[1] **Jiahui Wang**[1] **Shuowei Jin**[2]
Xueshen Liu[2] **Yongji Wu**[3] **Z. Morley Mao**[2] **Ion Stoica**[3]
Jiawei Zhao[4] **Beidi Chen**[1]
[1]Carnegie Mellon University [2]University of Michigan
[3]UC Berkeley [4]Meta

Abstract

Reinforcement learning (RL) is increasingly used to improve the reasoning, coding, and tool-use capabilities of large language models, but agentic RL remains prohibitively expensive. Scaling RL to agentic LLMs requires supporting complex workloads, including multi-policy collaborative training, while efficiently using elastic, heterogeneous, and cross-region compute resources. Existing LLM RL systems support some of these capabilities, but each new extension often requires dedicated system engineering. This burden arises from trainer-centered control architectures and the lack of principled abstractions for RL system components. To address these limitations, we propose AstraFlow, a dataflow-oriented RL system that replaces conventional trainer-centered control with principled component abstractions. In AstraFlow, rollout services, dataflow management, and training are decoupled into autonomous components, enabling the system to natively support complex multi-policy agentic RL workloads and efficiently exploit diverse compute resources. We evaluate AstraFlow across math, code, search, and AgentBench workloads, showing that the same system supports multi-policy training, elastic scaling, heterogeneous cross-region execution, and composable data algorithms without system-level code changes. In multi-policy collaborative training, AstraFlow achieves accuracy comparable to that of existing RL systems.

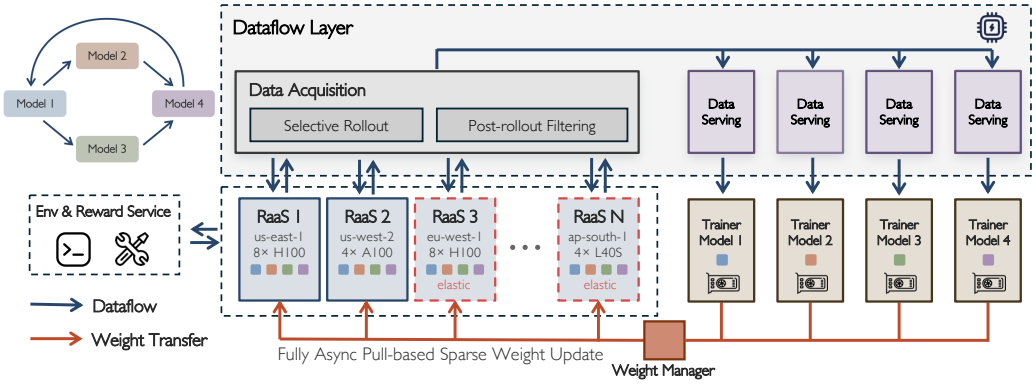


Figure 1: Overview of the AstraFlow architecture. A dataflow-oriented RL framework natively supports multi-policy collaborative training, elastic rollout, heterogeneous and cross-region rollout, and substitutable Rollout-as-a-Service (RaaS) and Trainer.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

1 Introduction

Large language models (LLMs) are rapidly moving beyond standalone use into complex agentic systems, including coding agents [15, 35], search agents [6, 50], and multi-agent workflows [2, 16]. In these settings, reinforcement learning (RL) has emerged as a key technique for improving reasoning and tool-using capabilities [7, 24, 27, 32, 43]. Yet scaling RL for agentic systems remains challenging. It must accommodate the complexity of agentic workloads, including dynamic execution and multi-policy coordination, as well as the diversity of underlying compute environments, such as elastic and heterogeneous computing. This underscores the need for a general RL infrastructure that unifies agent execution, training, and resource management under a flexible and scalable system design.

Despite this need, existing systems [1, 12, 30, 37] remain constrained by rigid designs that limit their scalability and extensibility. **First**, existing LLM RL systems [30] are primarily designed for single-policy training. Their trainer-centered control logic coordinates rollout scheduling, data movement, policy optimization, and weight synchronization, making them rigid for multi-policy agentic RL, where collaborative training requires coordinating multiple policies and their interactions. **Second**, SOTA multi-agent systems [8, 36] primarily focus on serving. They are designed to execute complex agent workflows efficiently, but lack the training-time coordination needed to collaboratively optimize multiple policies. **Finally**, recent systems can be engineered to add individual capabilities such as multi-policy training [3, 46], elastic rollout [37], or heterogeneous rollout [11, 40]. Although these systems can support such capabilities, they do so through ad-hoc patches that require feature-specific system engineering on the existing design. Table 1 summarizes how existing systems support each capability but generally lack the abstractions needed to support and compose them natively.

Table 1: System-level comparison of LLM RL training frameworks. ✓ denotes full support, ✗ no support, and *p* partial support.

Property	AstraFlow	AReaL	SLIME	verl†	RLBoost	Dr.MAS (verl)	prime-rl
Multi-policy collaborative training	✓	✗	✗	✗	✗	✓	✗
Substitutable trainer and rollout service	✓	✗	✗	✗	✗	✗	*p*
Modular data algorithm interfaces	✓	✗	✗	✗	✗	✗	✗
Fully asynchronous training	✓	✓	✓	✓	✗	✗	✓
Disaggregated rollout-training architecture	✓	✓	✓	✓	✓	✗	✓
Runtime elastic rollout scaling	✓	✗	✗	✗	✓	✗	*p*
Cross-region / heterogeneous rollout	✓	✗	✗	✗	✗	✗	*p*

†The original verl paper does not present fully asynchronous or disaggregated architecture; these entries reflect support later added in the open-source verl repository.

Building an ideal agentic RL system for LLMs requires rethinking several assumptions built into conventional LLM RL systems. First, such a system should treat multiple trainable policies and trainers as first-class components, rather than assuming a single-model, single-trainer control workflow. Second, it should also move beyond the assumption of a fixed compute environment, allowing workloads to seamlessly run across heterogeneous, cross-region, or elastic resources through clean execution interfaces. Third, it should move beyond tightly coupled implementations by exposing simple interfaces between rollout engines, trainers, and data algorithms, so that each component can be replaced or extended independently.

Our key insight is that the limitation of existing RL systems comes from a single trainer-centered control loop and the lack of principled abstractions among RL components. To address this, we propose ***AstraFlow***, a dataflow-oriented RL training system for agentic LLMs. AstraFlow defines RL-specific component boundaries and interfaces for data and weight transfer. As shown in Fig. 1, AstraFlow consists of three components: a dataflow layer, Rollout-as-a-Service (RaaS), and trainers. **1) Dataflow layer.** The dataflow layer coordinates rollout, training, and data-processing components through shared data, rather than centralized trainer control (Section 3.2). This enables autonomous rollout services and trainers to compose naturally, supports multi-policy collaborative training, and expresses policies such as curriculum scheduling, replay, data mixing, filtering, sampling, and staleness correction as dataflow policies. **2) Rollout-as-a-Service.** RaaS decouples trajectory generation from policy optimization through rollout interfaces (Section 3.3). This allows users to plug in optimized agent inference engines or specialized rollout backends without modifying trainers or system orchestration. It also enables rollout components to scale independently across heterogeneous, cross-region, and elastic compute resources. **3) Trainers.** Trainers consume data from the dataflow layer and publish updated weights back to the system (Section 3.4). Since they no longer directly control rollout scheduling, data movement, or rollout-runtime details, trainers become independently

replaceable. This makes it easy to integrate fault-tolerant trainers, specialized optimizers, or multiple trainers for multi-policy learning without changing the rest of the system.

In the evaluation part, we demonstrate the flexibility of AstraFlow from three perspectives: multi-policy collaborative RL, system flexibility, and data algorithm flexibility. For multi-policy collaborative RL, we evaluate AstraFlow on three multi-policy workflows, achieving accuracy comparable to that of the existing multi-agent RL system while reducing iteration time through asynchronous execution. Also, to the best of our knowledge, even without any multi-agent-specific system modifications, AstraFlow is the first fully asynchronous multi-policy collaborative RL framework. For system flexibility, we first show that rollout auto-scaling can be implemented through the resource-management interfaces without modifying the core system, with an agentic maintainer executing the fixed scaling policy. Then we show that AstraFlow natively supports heterogeneous and cross-region training without feature-specific engineering. For data flexibility, we demonstrate the flexibility of the dataflow-layer abstraction by integrating and composing data algorithms, including dynamic sampling [43], GRESO [49], and buffer replay. Together, we demonstrate that, thanks to the dataflow-oriented RL design, AstraFlow natively supports multi-policy collaborative training, diverse compute environments, and composable data algorithms without feature-specific system code.

Our source code is publicly available at `https://github.com/Infini-AI-Lab/astraflow`.

2 Related Work

2.1 RL for Agentic LLMs

RL has become a central post-training technique for improving LLM reasoning, code generation, and tool-use capabilities [7, 24, 27, 32, 43]. Many efforts improve the performance, stability, and efficiency of RL itself, including better policy-optimization objectives [26, 44, 47], reward design [34], off-policy or asynchronous training [23, 48], and data-centric algorithms [31, 38, 39, 49] that decide which prompts for sampling, which trajectories to keep, and which batches to train on. At the same time, RL workloads are expanding from single-turn reasoning to more complex agentic settings [1, 33, 45] such as software-engineering agents [15, 35], search tasks [50], and os environment interaction workflows [17]. These workloads introduce heterogeneous rollouts with variable lengths, tool feedback, intermediate artifacts, and data-policy interventions throughout training. Recent multi-agent RL workloads [3, 46] require collaboration among multiple trainable policies, further complicating training orchestration. Together, these trends make LLM RL workloads more complex and expensive, creating a need for better system support to run them efficiently on hardware resources.

2.2 LLM RL Training Frameworks

A typical RL training pipeline for LLMs has two major stages: *1) rollout*, where inference engines generate trajectories and rewards from the current policy, and *2) training*, where a trainer consumes rollout data and updates the policy. Existing LLM RL systems [1, 9, 10, 12, 13, 20, 25, 28, 30, 37, 51, 52] mainly organize these two stages in two ways. **Colocated synchronous systems** such as verl [30], Real [20], and RLHFuse [52] place training and rollout on the same GPU pool and alternate between trajectory generation and optimization. This design guarantees the on-policy training, but it suffers from long-tail rollout latency, leaving expensive trainer GPUs idle during rollout. **Disaggregated RL systems** such as AReaL [5], SLIME [53], and Laminar [29] address this utilization problem by decoupling rollout generation from policy optimization, allowing rollout workers and trainers to run on separate GPU pools and overlap execution. However, the heterogeneity between rollout and training in LLM RL complicates scheduling, resource allocation, and synchronization, while enabling optimizations such as elastic scaling [37] and heterogeneous resource management [11, 40]. Laminar addresses long-tail trajectory generation through trajectory-level asynchrony, relay-based weight distribution, and dynamic repacking. AstraFlow emphasizes RL-specific component interfaces for multi-policy workflows and data algorithms, building on general task- and actor-based execution models such as Ray [22].

3 Dataflow-Oriented RL for Agentic LLMs

In this section, we present the design of AstraFlow. We begin by motivating the shift from trainer-centered control to dataflow-oriented coordination, explaining why compute decoupling alone is insufficient for agentic RL in Section 3.1. We then introduce the three abstraction designs, as shown in Figure 1: a *dataflow layer* that manages rollouts and training batches (Section 3.2); a *Rollout-as-a-Service* abstraction that decouples trajectory generation from optimization (Section 3.3); and a *trainer* abstraction that consumes batches, updates policies, and publishes weights (Section 3.4).

3.1 Motivation: From Trainer-Centered Control to Dataflow-Oriented Coordination

Compute decoupling is not enough. Although disaggregated RL frameworks [5, 53] separate rollout and training computation, this separation is primarily a compute-placement mechanism, not a principled component abstraction. Rollout scheduling, data selection, replay, staleness handling, and weight synchronization often remain embedded in a trainer-centered control loop. As a result, new capabilities tend to require feature-specific system changes. Multi-policy collaborative training, for instance, requires coordinating multiple independently trained policies, trainers, and weight streams. Elastic, heterogeneous, or cross-region rollouts require additional mechanisms for workers to join and leave dynamically and for weights to be transferred under resource constraints. These capabilities can be added to existing systems, but usually through ad hoc patches or substantial redesign, like Areal-Hex [40]; composing several of them only amplifies the complexity. **The root cause behind this limitation is the lack of clean abstraction boundaries among rollout execution, dataflow management, training, and weight transfer.** Without these boundaries, new capabilities cannot be supported naturally by the system design and instead require explicit feature-specific engineering. Table 1 summarizes this abstraction gap: existing LLM RL frameworks may support asynchrony or rollout-training disaggregation, but generally lack abstractions for composing them.

Dataflow-oriented coordination. Motivated by this abstraction gap, we propose dataflow-oriented RL for agentic LLMs, a design principle implemented in AstraFlow. The key insight is that disaggregation should not only separate rollout and training computation, but also separate their control responsibilities. Instead of organizing the system around a trainer-centered control loop, AstraFlow uses dataflow-oriented coordination: rollout services, trainers, and the dataflow layer each run *autonomous control loops* and interact only through minimal data and weight interfaces. These interfaces turn rollout, training, data management, and weight transfer into composable system boundaries. As a result, capabilities such as multi-policy collaborative training, elastic rollout pools, heterogeneous and cross-region rollouts, and modular data algorithms can be expressed by the system architecture itself, rather than added through feature-specific engineering.

Design challenges. To realize this design, AstraFlow must address three challenges. First, it must provide a data coordination layer that manages prompts, trajectories, rewards, batching, routing, replay, and staleness across multiple rollout services and trainers without returning control to a single trainer loop. Second, it must expose stable component boundaries so that rollout engines, trainer backends, and data algorithms can be replaced or extended without pipeline rewrites. Third, it must support asynchronous and bandwidth-efficient weight flow across multiple policies and rollout pools, including elastic, heterogeneous, and cross-region deployments.

3.2 Dataflow Layer Abstraction

The dataflow layer is the coordination plane between rollout services and trainers. The layer represents RL data in its natural units, including prompts, trajectories, metadata, and training batches. RaaS nodes pull rollout tasks from the layer and push completed trajectories back, while trainers independently pull batches according to their own optimization loops.

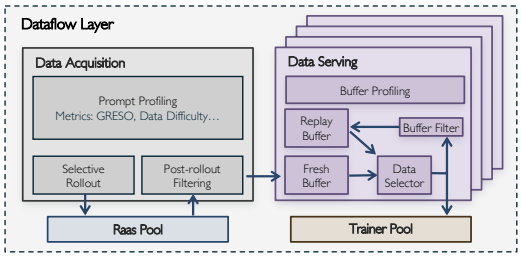


Figure 2: Dataflow layer abstraction. Prompt sources, RaaS nodes, and trainers interact through a shared layer that buffers data and applies sampling, filtering, and routing policies.

Data algorithm interface. The dataflow layer exposes a programmable interface for algorithms that operate on prompts, trajectories, rewards, and metadata. Policies such as selective rollout, curriculum scheduling, post-rollout fil-

tering, dynamic sampling [43], replay, data mixing, and staleness correction can therefore be implemented as dataflow policies without modifying the trainer, RaaS implementation, or system orchestration.

Data-driven coordination. The dataflow layer also coordinates autonomous rollout services and trainers through data availability and routing. Although each component runs its own control loop, the layer can regulate their interaction by deciding which rollout tasks, trajectories, and batches each component receives. For example, it can throttle slow or stale rollout services by assigning fewer tasks, prioritize fresher trajectories for a trainer, or block unsuitable batches through backpressure. In multi-policy training, trajectory metadata such as producing policy, model version, timestamp, reward statistics, and task type allows the layer to route policy-specific, shared, or mixed data streams to different trainers without requiring direct trainer-to-trainer coordination.

Together, these two roles make the dataflow layer both a modular data-algorithm interface and a control plane for independent components. New data policies and coordination strategies can be added in the dataflow layer rather than by rewriting a trainer-centered control loop.

3.3 Rollout-as-a-Service (RaaS) Abstraction

RaaS models rollout generation as a pure *agent-serving service*. Each RaaS node receives tasks from the dataflow layer, executes the corresponding agent workflow, and returns trajectories. The RaaS interface only requires that the service consume tasks, produce trajectories, and refresh weights through the trainer-side weight-transfer interface. This interface makes rollout execution substitutable. An efficient agent-serving runtime can be plugged into AstraFlow as long as it follows the RaaS contract. The runtime does not need to know how trajectories are sampled, replayed, filtered, or assigned to trainers. Likewise, the trainer does not need to know which serving runtime produced the trajectory. This separation allows AstraFlow to reuse specialized agent-serving systems as rollout backends instead of re-implementing their internal execution logic.

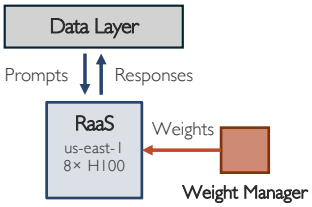
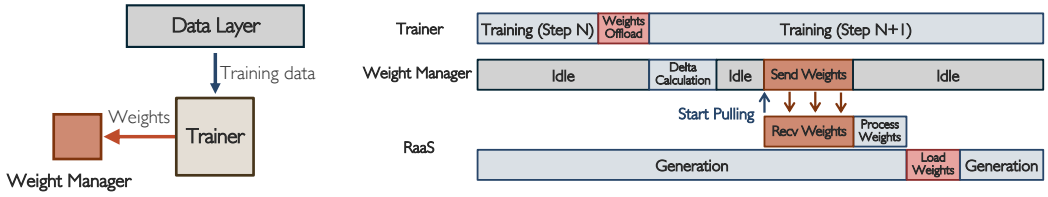


Figure 3: RaaS abstraction. Each rollout node consumes tasks, produces trajectories, and refreshes weights.

RaaS also makes rollout capacity elastic. Adding capacity means launching more RaaS nodes that connect to the same dataflow layer and weight-transfer interface. Removing capacity, slow workers, or failures affect only the rate at which trajectories arrive, not the independent trainer control loop. This property is especially useful for heterogeneous and cross-region settings, where rollout services may have different latency, throughput, and network bandwidth.

3.4 Trainer Abstraction and Weight Transfer



(a) Trainer abstraction.

(b) Delta weight transfer.

Figure 4: **(a)** Each trainer consumes batches from the dataflow layer and publishes updated weights through a trainer-side weight interface. **(b)** Fully async pull-based sparse weight update.

A trainer consumes batches from the dataflow layer, performs optimization with its own backend, and publishes updated weights through a trainer-side weight-transfer interface. From the trainer's perspective, the dataflow layer behaves like a streaming training corpus, and the weight-transfer interface behaves like a publication target. The trainer does not need to manage rollout workers, serve model weights directly, or coordinate with other trainers.

This abstraction also makes training backends substitutable. An existing RL, SFT, or fault-tolerant training backend can participate in AstraFlow if it can consume batches from the dataflow layer and publish weights through the same trainer abstraction. The same interface also supports multi-policy

collaborative training: each policy can have an independent trainer and weight stream, while the dataflow layer controls how trajectories are distributed to each trainer.

Weight Transfer. Within the trainer abstraction, the weight-transfer mechanism owns the weight flow between trainers and rollout services. It stores model versions, exposes the latest or requested versions to RaaS nodes, and handles asynchronous refresh. Because RaaS nodes pull weights when appropriate, weight delivery is not part of the trainer's critical path. The trainer-side weight interface can implement full-model transfer, sparse transfer, and version-aware refresh behind the same abstraction. This design keeps constrained or remote weight transfer isolated from trainer logic while allowing rollout services to refresh at different rates.

4 Evaluation: Applications of AstraFlow

In this section, we demonstrate the flexibility of AstraFlow from three perspectives: multi-policy collaborative training, system flexibility, and data algorithm flexibility.

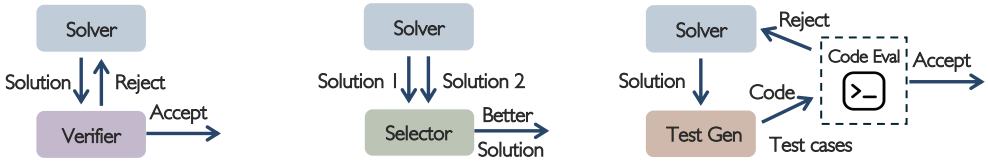
- Section 4.1 evaluates three two-policy workflows, including a three-seed math comparison with a reproduced baseline.
- Section 4.2.1 demonstrates rollout auto-scaling through existing interfaces without modifying core system code.
- Section 4.2.2 evaluates heterogeneous and simulated cross-region training without feature-specific engineering.
- Section 4.3 validates the dataflow-layer abstraction by integrating and composing data algorithms, including dynamic sampling [43], GRESO [49], and buffer replay.

Due to the space limitation, we include our detailed experimental setting in Appendix A.

4.1 Application I: Multi-Policy Collaborative Training

We first evaluate AstraFlow's flexibility on multi-policy collaborative RL through three multi-agent workflows illustrated in Figure 5: math solver–verifier, code solver–selector, and code solver–test-case generation. In AstraFlow, users only specify the multi-agent workflow: role order, context passing, and reward assignment. Existing multi-agent RL systems such as Dr. MAS [3] and Stronger MAS [46] build this support on top of verl [30], requiring pipeline-level modifications to coordinate multiple roles, policies, trainers, and weight streams, while also inheriting verl's colocated synchronous execution. In contrast, AstraFlow natively supports multi-policy workflows through its dataflow-oriented RL abstraction. As a result, multi-agent RL becomes a workflow-level change rather than a system-level pipeline modification. To the best of our knowledge, AstraFlow is the first LLM RL framework to support fully asynchronous multi-policy collaborative training.

Multi-agent Workflow. For every multi-policy run, both roles are initialized from Qwen3-8B [41] and trained as separate policies. In the math workflow, a Solver proposes a solution and a Verifier accepts or rejects it; if rejected, the Solver receives feedback and retries. In the code-selector workflow, the Solver generates two candidate programs and the Selector chooses one to submit. In the code test-case workflow, the Solver writes a program, the Test-Case Generator produces tests, an evaluator executes the program on those tests, and the Solver retries on failure. Full prompt templates for each role are given in Appendix A.11.



(a) Math: Solver + Verifier. (b) Code: Solver + Selector. (c) Code: Solver + Test-Case Generator.

Figure 5: Multi-agent workflows evaluated: **(a)** Math solver + verifier, **(b)** Code solver + selector, and **(c)** Code solver + test-case generator with retry. Different policies serve different roles in a workflow.

Comparable accuracy with asynchronous multi-policy training. Table 2 reports results from three independent training seeds for the reproduced Dr. MAS baseline and AstraFlow. Average accuracy is

$53.78 \pm 0.68\%$ for Dr. MAS and $54.96 \pm 1.33\%$ for AstraFlow, while iteration time is 221.14 ± 8.42 s and 76.86 ± 5.50 s, respectively. The shorter iteration time is enabled by asynchronous execution, which AstraFlow supports without workload-specific changes to its core system.

Table 2: Math multi-policy training with Qwen3-8B. Solver + Verifier rows report mean $\pm$ standard deviation over three independent training seeds, including our reproduced Dr. MAS baseline. Accuracy is in percent. The single-policy Solver row is reported by Dr. MAS [3] and included for reference.

Method	AIME24	AIME25	MATH500	Minerva	Avg Acc. (%)	Time/iter (s)
Solver (reported)	42.9	31.8	90.5	39.2	51.1	—
Solver + Verifier (Dr. MAS, reproduced)	44.89 $\pm$1.65	39.87 $\pm$2.10	90.63 $\pm$0.40	39.73 $\pm$2.26	53.78 $\pm$0.68	221.14 $\pm$8.42
Solver + Verifier (AstraFlow)	44.72 $\pm$2.61	38.67 $\pm$1.78	92.27 $\pm$0.56	44.17 $\pm$0.96	54.96 $\pm$1.33	76.86 $\pm$5.50

Generality to code multi-agent workflows. Table 3 evaluates whether the same multi-policy abstraction extends beyond math to code generation. We consider two interaction patterns: Solver + Selector, where the Solver generates two candidates and the Selector chooses one to submit, and Solver + Test-Case Generator, where generated tests provide execution feedback for retry. Both workflows improve over the matched single-policy Solver on every benchmark, with the stronger workflow raising average accuracy from 30.29% to 34.55% ($+4.26$). We use the matched Solver baseline rather than a specialized code-agent baseline because this experiment is intended to test system generality under controlled training conditions; Table 2 provides the direct system-to-system efficiency comparison. The result shows that AstraFlow can express substantially different multi-agent code workflows by changing only workflow logic, while reusing the same trainer, rollout, dataflow, and weight interfaces.

Table 3: Code-generation accuracy of single-policy vs. multi-policy collaborative training with Qwen3-8B [41]. We report two multi-policy workflows.

Method	LiveCodeBench v5	LiveCodeBench v6	Codeforces	Avg
Solver	36.83	32.86	21.20	30.29
Solver + Selector	38.32 (+1.49)	35.43 (+2.57)	22.67 (+1.47)	32.14 (+1.85)
Solver + Test-Case Generator	**41.62** (+4.79)	**36.29** (+3.43)	**25.74** (+4.54)	**34.55** (+4.26)

Takeaway. Across math and code, AstraFlow supports multiple two-policy workflows through the same interfaces. The reproduced math comparison shows comparable accuracy with shorter iteration times enabled by asynchronous execution.

4.2 Application II: System Flexibility

We next evaluate AstraFlow's system flexibility through two case studies: (i) rollout auto-scaling through resource-management interfaces (Section 4.2.1), and (ii) heterogeneous and simulated cross-region training over throttled GPUs and limited-bandwidth links (Section 4.2.2).

For both case studies, no feature-specific engineering is required: the same RaaS abstraction (Section 3.3) and weight-transfer runtime (Section 3.4) absorb auto-scaling, GPU heterogeneity, and slow remote synchronization without any change to the AstraFlow system.

4.2.1 Case Study I: Rollout Auto-Scaling with an Agentic Maintainer

Although AstraFlow does not control the trainer or the rollout services directly, the dataflow layer sits on the path that connects them and can therefore observe the workload balance: how many trajectories each rollout pool produces, how many the trainer consumes, and how long the trainer spends waiting on data.

Every K trainer steps, AstraFlow exports a summary of these three quantities, the trainer waiting fraction w, the recent production and consumption counts n_p and n_c, and the availability of each rollout service, and uses them to derive a target rollout-pool size with a simple three-zone policy, applying a dead band $[\tau_{\text{low}}, \tau_{\text{high}}]$ on w:

$$G_{\text{target}} = \begin{cases} \lceil G/(1-w) \rceil & \text{if } w > \tau_{\text{high}}, \\ \min\big(G,\ \lceil G \cdot (n_c/n_p) \cdot \rho \rceil\big) & \text{if } w < \tau_{\text{low}},\ n_p, n_c > 0, \\ G & \text{otherwise}, \end{cases} \tag{1}$$

where G is the current rollout GPU count and ρ is the shrink-margin factor. The policy adds GPUs when the trainer is starving ($w > \tau_{\text{high}}$), removes GPUs when the pool is over-allocated ($w < \tau_{\text{low}}$).

Table 4: Accuracy, training time, and GPU-hour comparison across the three rollout-pool configurations on the Qwen3-14B [41] math job. Auto-scaling keeps wall-clock time close to the over-allocated baseline while reducing total GPU-hours. Bold marks the best GPU-hour costs.

RaaS Strategy	Avg Acc.	Wall (h)	Rollout GPU-h	Trainer GPU-h	Total GPU-h	Wait frac.
Fixed 6 GPUs	68.6	35.8	214.8	143.2	358.0	26.9%
Fixed 11 GPUs	68.0	23.9	263.4	95.8	359.2	2.1%
Auto-scaling	67.9	24.4	**214.5**	97.4	**312.0**	3.0%

Interface-based rollout auto-scaling. The RaaS abstraction (Section 3.3) makes each rollout service hot-swappable: the dataflow layer treats instances as interchangeable producers and tolerates them joining or leaving at runtime, so resizing the rollout pool reduces to launching or retiring RaaS instances on demand. The fixed controller computes G_{target}, and the maintainer executes the prescribed resource operations through the exposed interfaces.

We use Claude Code as the agentic maintainer in our experiment: it runs alongside the training job, polls the utilization summary every K steps, follows the suggested G_{target}, and issues the corresponding cluster commands to launch or retire RaaS instances—preferentially removing unavailable or persistently low-throughput services on scale-down. AstraFlow contains no scheduler-specific code; porting the loop to Slurm, Kubernetes, or an in-house scheduler only requires adapting the maintainer's instructions to the corresponding cluster interface. The same fixed policy can also be executed by a script through these interfaces.

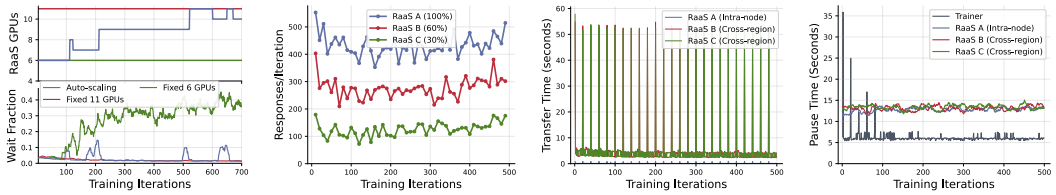
Auto-scaling minimizes GPU-hours with comparable accuracy. We compare auto-scaling against two fixed rollout-pool baselines that match the lower and upper ends of the controller's scaling range: 6 rollout GPUs and 11 rollout GPUs. Table 4 reports the resulting accuracy, wall-clock time, trainer wait fraction, and GPU-hour cost. The 6-GPU baseline uses few rollout resources but starves the trainer, while the 11-GPU baseline removes most trainer waiting by keeping the larger pool allocated for the full run. Auto-scaling achieves comparable accuracy and nearly the same wall-clock time as the 11-GPU setting, with a small increase from 23.9h to 24.4h. However, by releasing rollout capacity when it is not needed, it reduces total GPU-hours to 312.0, about 13% lower than both fixed-pool baselines. Figure 6a traces the loop in action: the maintainer scales up when trainer waiting rises, holds inside the dead band, and scales down when sustained low waiting indicates over-allocation.

4.2.2 Case Study II: Heterogeneous and Simulated Cross-Region Training

We simulate a cross-region heterogeneous deployment using three nodes (a four-GPU trainer plus three four-GPU RaaS pools, one local and two remote) and train Qwen3-14B [41] with M2PO on math. GPU heterogeneity is induced by per-GPU power caps—700 W on the local pool and 400 W and 250 W on the two remote pools—yielding rollout-throughput shares of roughly 100%, 60%, and 30% respectively in Figure 6b. To emulate the cross-region paths, the two remote links are further shaped to 4 Gbit/s effective bandwidth and 300 ms round-trip latency. All three pools contribute to every iteration and the trainer never blocks on any single pool.

The heterogeneous cross-region run reaches 67.6% average accuracy, compared with 67.2% for the homogeneous reference using 12 local H200 rollout GPUs. Iteration time is 138.6 s versus 103.6 s, an increase of approximately 34% under the simulated deployment.

Figure 6c reports per-iteration weight-transfer time on each link. With $\geq 98.9\%$ delta sparsity for Qwen3-14B [41] (Figure 7), the per-iteration payload drops from a $\sim$28 GB full sync to roughly



(a) (b) (c) (d)

Figure 6: System-flexibility experiments. **(a)** Rollout GPU count and trainer waiting time under different rollout-pool settings. **(b)** Per-iteration rollout throughput from different RaaS pools. **(c)** Weight-transfer time across local and remote rollout pools. **(d)** Trainer and rollout downtime in the heterogeneous cross-region setting.

1.5 GB of delta bytes, so most remote transfers complete in tens of seconds even at 4 Gbit/s and 300 ms RTT. The periodic full syncs (every 20 iterations) appear as the visible spikes; they remain bounded because they are amortized over many delta steps.

Figure 6d shows that even the slow full-sync transfers do not block the training loop. On the rollout side, downtime is essentially constant across the run: just the SGLang reload and prefill window after each weight update, independent of how the bytes arrived. On the trainer side, the early iterations show occasional waiting because the previous iteration's full transfer is still finishing on a remote pool; as training progresses and generation lengths grow, the training phase becomes long enough to fully overlap the next iteration's transfer, and trainer downtime converges to a near-constant baseline. This is the behavior the request-based delta-pull design is intended to deliver (Section 3.4): the cost of a slow link is masked by ongoing training work, so heterogeneous rollout and slow remote synchronization are absorbed by the existing abstractions without any change to the trainer, RaaS, or dataflow interfaces.

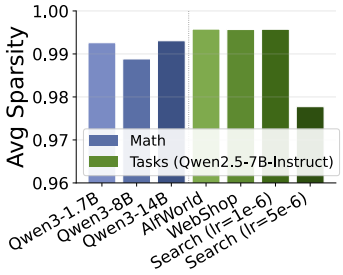


Figure 7: Weight delta sparsity on different models and tasks.

Remote Sparse Weight Transfer. As discussed in recent works [4, 21], per-iteration weight updates in RL training are extremely sparse: under bf16 most parameters are bit-exactly identical from one iteration to the next, so the trainer needs to ship only a tiny fraction of its weights to the rollout fleet. Figure 7 measures this directly across two axes: model scale (Qwen3-1.7B / 8B / 14B [41] on math) and task (Qwen2.5-7B on AlfWorld, WebShop, and Search), reporting the fraction of bf16 parameters bit-exactly equal to the previous iteration, averaged over the first 500 training iterations. At a fixed learning rate, sparsity is largely independent of model size and task: every math run lands in 0.989–0.993 and the Qwen2.5-7B tasks all reach ≥ 0.996 at $\mathrm{lr} = 1\mathrm{e}{-6}$. Learning rate is the dominant driver: raising it to $5\mathrm{e}{-6}$ on Search drops sparsity to 0.978, since larger updates push more parameters past the bf16 ULP boundary, but even in this most aggressive regime sparsity still exceeds 0.97, leaving a $\geq 30\times$ compression upper bound on the bytes the trainer must ship per iteration. The request-based remote weight-transfer path can therefore rely on a high-sparsity assumption across all workloads we measured; per-iteration sparsity curves are deferred to Appendix B.2.

4.2.3 Performance Comparison to Existing RL Framework

Table 5: Performance comparison against AReaL on Qwen3-1.7B and Qwen3-8B [41] M2PO math reasoning tasks trained for 800 iterations. The two frameworks achieve comparable accuracy and iteration time in these evaluated settings.

Model	Framework	AIME24	AIME25	AMC	MATH500	Minerva	Avg	Time/iter (s)	Time/1M tok (s)
Qwen3-1.7B	AReaL	32.5	27.5	61.1	88.2	38.2	49.5	81.5	70.1
	AstraFlow	33.3	30.6	59.2	87.5	35.8	49.3	81.1	69.4
Qwen3-8B	AReaL	60.0	55.0	72.2	94.6	45.2	65.4	137.0	117.6
	AstraFlow	62.3	50.0	72.5	94.9	44.5	64.8	139.6	119.6

We train Qwen3-1.7B and Qwen3-8B [41] math jobs with M2PO on AstraFlow and on AReaL [5], a representative trainer-centric RL framework, and compare accuracy and training speed (Table 5). At

both scales, the two systems reach comparable accuracy on the math suite (within 0.2 / 0.6 points on average) and comparable per-iteration training speed (within 1% / 2%): AstraFlow matches AReaL's performance and efficiency on a workload AReaL is specialized for, while still providing the flexibility demonstrated in the previous sections. Additional comparisons on Llama 3.2-3B-Instruct and Llama 3-8B-Instruct are reported in Appendix B.4.

4.3 Application III: Data Algorithm Flexibility

We train Qwen3-8B [41] on math with three representative data algorithms, chosen to cover the three intervention points along the RL data path: *GRESO* [49] performs **selective rollout**, deciding which prompts are sent to generation; *dynamic sampling* [43] performs **post-rollout filtering**, discarding zero-advantage trajectories after generation; and *buffer replay* performs **training-batch selection**, reusing useful trajectories across multiple trainer updates. Together they exercise pre-, post-, and serving-side hooks of the dataflow layer.

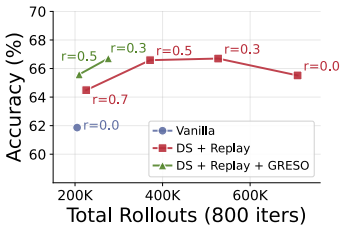


Figure 8: Math accuracy versus generated rollouts for dataflow-layer data algorithms.

Figure 8 reports accuracy versus generated rollouts for the three algorithms. Dynamic sampling [43] lifts final accuracy meaningfully but increases generation cost by roughly 3.5× (about 200k → 700k rollouts), since post-filtering discards a large fraction of generations. GRESO [49] and buffer replay sit on the opposite side of the trade-off: both reach baseline-level accuracy with substantially fewer generated rollouts—GRESO [49] by avoiding low-value prompts before rollout, buffer replay by reusing trajectories at batch serving. This data algorithm composition shows that: each algorithm is a self-contained policy class that the dataflow layer imports as a plug-in, so prompt selection, rollout filtering, and trajectory replay become composable, modular components rather than system-wide rewrites.

5 Conclusion

We presented AstraFlow, a dataflow-oriented RL training system for agentic LLMs. Our central observation is that the rigidity of existing RL training systems comes from a single architectural choice: the trainer sits at the center of the loop and orchestrates everything else. AstraFlow replaces this trainer-centered control with three narrow abstractions, a dataflow layer for coordination, a rollout service for trajectory generation, and a trainer for policy updates, that interact only through stable interfaces. Because no component owns the global control flow, properties such as full asynchrony, elastic rollout scaling, plug-in trainers and inference engines, multi-trainer multi-policy coordination, and efficient delta and request-based sparse weight transfer all fall out of the same design rather than being bolted on as separate mechanisms.

Acknowledgments

We thank NVIDIA for providing computing resources. This research received support from the Google Research Award, Google ML & System Junior Faculty Award, Amazon Research Award, Fireworks AI, Intel, Li Auto, Moffett AI, and CMU CyLab Seed funding. Additional support came from the National Science Foundation (Grant No. 2504353) and IARPA. The opinions, findings, and conclusions presented here are the authors' and do not necessarily represent the views of the National Science Foundation.

References

[1] Shiyi Cao, Dacheng Li, Fangzhou Zhao, Shuo Yuan, Sumanth R Hegde, Connor Chen, Charlie Ruan, Tyler Griggs, Shu Liu, Eric Tang, et al. SkyRL-Agent: Efficient RL training for multi-turn LLM agent. *arXiv preprint arXiv:2511.16108*, 2025.

[2] Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent LLM systems fail? *arXiv preprint arXiv:2503.13657*, 2025.

[3] Lang Feng, Longtao Zheng, Shuo He, Fuxiang Zhang, and Bo An. Dr. mas: Stable reinforcement learning for multi-agent LLM systems. *arXiv preprint arXiv:2602.08847*, 2026.

[4] Fireworks AI. Frontier RL is cheaper than you think. `https://fireworks.ai/blog/frontier-rl-delta-weight-updates`, March 2026. Fireworks.ai Blog.

[5] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, et al. AReaL: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.

[6] Jiaxuan Gao, Wei Fu, Minyang Xie, Shusheng Xu, Chuyi He, Zhiyu Mei, Banghua Zhu, and Yi Wu. Beyond ten turns: Unlocking long-horizon agentic search with large-scale asynchronous RL. *arXiv preprint arXiv:2508.07976*, 2025.

[7] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[8] Shanshan Han, Qifan Zhang, Weizhao Jin, and Zhaozhuo Xu. LLM multi-agent systems: Challenges and open problems. *arXiv preprint arXiv:2402.03578*, 2024.

[9] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, et al. Asyncflow: An asynchronous streaming RL framework for efficient LLM post-training. *arXiv preprint arXiv:2507.01663*, 2025.

[10] Jingkai He, Tianjian Li, Erhu Feng, Dong Du, Qian Liu, Tao Liu, Yubin Xia, and Haibo Chen. History rhymes: Accelerating LLM reinforcement learning with rhymerl. *arXiv preprint arXiv:2508.18588*, 2025.

[11] Yongjun He, Shuai Zhang, Jiading Gai, Xiyuan Zhang, Boran Han, Bernie Wang, Huzefa Rangwala, and George Karypis. Hetrl: Efficient reinforcement learning for LLMs in heterogeneous environments. *arXiv preprint arXiv:2512.12476*, 2025.

[12] Brad Hilton, Kyle Corbitt, David Corbitt, Saumya Gandhi, Angky William, Bohdan Kovalevskyi, and Andie Jones. Art: Agent reinforcement trainer. `https://github.com/openpipe/art`, 2025.

[13] Jian Hu, Xibin Wu, Wei Shen, Jason Klein Liu, Zilin Zhu, Weixun Wang, Songlin Jiang, Haoran Wang, Hao Chen, Bin Chen, et al. OpenRLHF: An easy-to-use, scalable and high-performance RLHF framework. *arXiv preprint arXiv:2405.11143*, 6, 2024.

[14] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[15] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

[16] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training LLMs to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

[17] Hanyu Lai, Xiao Liu, Yanxiao Zhao, Han Xu, Hanchen Zhang, Bohao Jing, Yanyu Ren, Shuntian Yao, Yuxiao Dong, and Jie Tang. Computerrl: Scaling end-to-end online reinforcement learning for computer use agents. *arXiv preprint arXiv:2508.14040*, 2025.

[18] Michael Luo, Sijun Tan, Roy Huang, Ameen Patel, Alpay Ariyak, Qingyang Wu, Xiaoxiang Shi, Rachel Xin, Colin Cai, Maurice Weber, Ce Zhang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. DeepCoder: A fully open-source 14B coder at O3-mini level. `https://www.together.ai/blog/deepcoder`, April 2025. Agentica and Together AI blog.

[19] Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Tianjun Zhang, Erran Li, Raluca Ada Popa, and Ion Stoica. Deepscaler: Surpassing o1-preview with a 1.5b model by scaling RL. `https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8ca303013a4e2`, 2025. Notion Blog.

[20] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. ReaL: Efficient RLHF training of large language models with parameter reallocation. In *Proceedings of the Eighth Conference on Machine Learning and Systems, MLSys 2025, Santa Clara, CA, USA, May 12-15, 2025*. mlsys.org, 2025.

[21] Erfan Miahi and Eugene Belilovsky. Understanding and exploiting weight update sparsity for communication-efficient distributed RL. *arXiv preprint arXiv:2602.03839*, 2026.

[22] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577. USENIX Association, 2018.

[23] Michael Noukhovitch, Shengyi Huang, Sophie Xhonneux, Arian Hosseini, Rishabh Agarwal, and Aaron Courville. Asynchronous RLHF: Faster and more efficient off-policy RL for language models. *arXiv preprint arXiv:2410.18252*, 2024.

[24] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[25] Prime Intellect. PRIME-RL. `https://github.com/PrimeIntellect-ai/prime-rl`, 2025.

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

[27] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[28] Gerald Shen, Zhilin Wang, Olivier Delalleau, Jiaqi Zeng, Yi Dong, Daniel Egert, Shengyang Sun, Jimmy Zhang, Sahil Jain, Ali Taghibakhshi, et al. NeMo-Aligner: Scalable toolkit for efficient model alignment. *arXiv preprint arXiv:2405.01481*, 2024.

[29] Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu, Xiang Li, Chi Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. Laminar: A scalable asynchronous RL post-training framework. In *Proceedings of the 21st European Conference on Computer Systems*, pages 400–422, 2026.

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

[31] Yifan Sun, Jingyan Shen, Yibin Wang, Tianyu Chen, Zhendong Wang, Mingyuan Zhou, and Huan Zhang. Improving data efficiency for LLM reinforcement fine-tuning through difficulty-targeted online data selection and rollout replay. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[32] Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. Kimi k1. 5: Scaling reinforcement learning with LLMs. *arXiv preprint arXiv:2501.12599*, 2025.

[33] Shijie Wang, Pengfei Li, Yikun Fu, Kaifeng Liu, Fangyuan Li, Yang Liu, Xiaowei Sun, Zonglin Li, Siyao Zhao, Jian Zhao, et al. Marti-mars2: Scaling multi-agent self-search via reinforcement learning for code generation. *arXiv preprint arXiv:2602.07848*, 2026.

[34] Yinjie Wang, Tianbao Xie, Ke Shen, Mengdi Wang, and Ling Yang. Rlanything: Forge environment, policy, and reward model in completely dynamic RL system. *arXiv preprint arXiv:2602.02488*, 2026.

[35] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. Swe-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.

[36] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First conference on language modeling*, 2024.

[37] Yongji Wu, Xueshen Liu, Haizhong Zheng, Juncheng Gu, Beidi Chen, Z Morley Mao, Arvind Krishnamurthy, and Ion Stoica. RLBoost: Harvesting preemptible resources for cost-efficient reinforcement learning on LLMs. *arXiv preprint arXiv:2510.19225*, 2025.

[38] Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. Less: selecting influential data for targeted instruction tuning. In *Proceedings of the 41st International Conference on Machine Learning*, pages 54104–54132, 2024.

[39] Yixuan Even Xu, Yash Savani, Fei Fang, and Zico Kolter. Not all rollouts are useful: Down-sampling rollouts in LLM reinforcement learning. *arXiv preprint arXiv:2504.13818*, 2025.

[40] Ran Yan, Youhe Jiang, Tianyuan Wu, Jiaxuan Gao, Zhiyu Mei, Wei Fu, Haohui Mai, Wei Wang, Yi Wu, and Binhang Yuan. AReaL-hex: Accommodating asynchronous RL training over heterogeneous gpus. *arXiv preprint arXiv:2511.00796*, 2025.

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

[42] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[43] Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

[44] Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, TianTian Fan, Zhengyin Du, et al. VAPO: Efficient and reliable reinforcement learning for advanced reasoning tasks. *arXiv preprint arXiv:2504.05118*, 2025.

[45] Hanchen Zhang, Xiao Liu, Bowen Lv, Xueqiao Sun, Bohao Jing, Iat Long Iong, Zhenyu Hou, Zehan Qi, Hanyu Lai, Yifan Xu, Rui Lu, Hongning Wang, Jie Tang, and Yuxiao Dong. Agentrl: Scaling agentic reinforcement learning with a multi-turn, multi-task framework, 2025.

[46] Yujie Zhao, Lanxiang Hu, Yang Wang, Minmin Hou, Hao Zhang, Ke Ding, and Jishen Zhao. Stronger-mas: Multi-agent reinforcement learning for collaborative LLMs. *arXiv preprint arXiv:2510.11062*, 2025.

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

[48] Haizhong Zheng, Jiawei Zhao, and Beidi Chen. Prosperity before collapse: How far can off-policy RL reach with stale data on LLMs? In *ICLR*, 2026.

[49] Haizhong Zheng, Yang Zhou, Brian R Bartoldson, Bhavya Kailkhura, Fan Lai, Jiawei Zhao, and Beidi Chen. Act only when it pays: Efficient reinforcement learning for LLM reasoning via selective rollouts. *arXiv preprint arXiv:2506.02177*, 2025.

[50] Yuxiang Zheng, Dayuan Fu, Xiangkun Hu, Xiaojie Cai, Lyumanshan Ye, Pengrui Lu, and Pengfei Liu. DeepResearcher: Scaling deep research via reinforcement learning in real-world environments. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 414–431, 2025.

[51] Yinmin Zhong, Zili Zhang, Xiaoniu Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, et al. Streamrl: Scalable, heterogeneous, and elastic RL for LLMs with disaggregated stream generation. *arXiv preprint arXiv:2504.15930*, 2025.

[52] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, et al. Optimizing RLHF training for large language models with stage fusion. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 489–503, 2025.

[53] Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An LLM post-training framework for RL scaling. `https://github.com/THUDM/slime`, 2025. GitHub repository.

Limitations

AstraFlow focuses on system abstractions for agentic RL workloads rather than proposing a new RL optimization algorithm. Our evaluation covers representative math, coding, multi-policy, elastic rollout, heterogeneous deployment, and data-algorithm workloads, but does not exhaust all agentic RL settings, such as long-horizon web agents, robotics environments, or safety-critical interactive systems. The observed benefits may depend on workload characteristics including rollout latency, trainer throughput, network bandwidth, weight-transfer frequency, and asynchronous data availability. In addition, some experiments use controlled or simulated deployment settings, and AstraFlow assumes that rollout tasks, trajectory metadata, and reward signals can be represented through the dataflow layer. While AstraFlow makes multi-policy and data-centric RL workloads easier to compose, model quality, robustness, and safety still depend on the underlying data, rewards, policies, and evaluation protocol.

Social Impact

AstraFlow is an infrastructure system for reinforcement learning of agentic language models. Its positive impact is to make advanced RL experimentation more modular and resource-efficient by allowing researchers to reuse rollout services and trainers, compose data algorithms, and run flexible multi-policy or heterogeneous deployments without rewriting the full training pipeline. At the same time, more efficient RL infrastructure may accelerate stronger agentic models for coding, tool use, and autonomous decision-making, which could be misused without appropriate evaluation, monitoring, access control, or domain-specific safeguards. AstraFlow does not release a new foundation model or dataset, but users should still evaluate trained agents for reliability, security, privacy, and misuse risks before deployment.

A Experimental Settings

This appendix provides the full experimental setup deferred from Section 4. We organize the appendix per-experiment so that each subsection is self-contained and reproducible. Section A.1 fixes the shared bits that recur across experiments (model cards, evaluation suites, decoding, reward functions); Sections A.2–A.8 then specify, one per main-text experiment, the training data, algorithm hyperparameters, hardware and topology, baselines, and any experiment-specific knobs.

A.1 Common Setup

Evaluation protocol. Across *all* experiments in this appendix, we evaluate each benchmark by sampling 4 independent generations per question (temperature 0.6, $n_{\text{samples}} = 1$) and reporting the mean per-question accuracy averaged over the benchmark. For every run we evaluate at a fixed cadence during training and report numbers from the checkpoint with the best benchmark-average accuracy. Per-experiment subsections below therefore omit this protocol and only note any deviations.

Training-seed repetitions. For the mathematical Solver–Verifier comparison, we repeat both the reproduced Dr. MAS baseline and AstraFlow training with three independent random seeds. We report the mean and standard deviation across these training runs for benchmark accuracy and iteration time. These statistics capture training-run variability, whereas avg@4 averages four sampled responses per evaluation question.

A.2 Multi-policy Math (Section 4.1, Table 2)

Models & Datasets. Both Solver and Verifier are initialized from Qwen3-8B [41] and trained as separate policies under the actor-and-verify workflow described in Section 4.1. We train on the DAPO RL math set [43] (Hugging Face: `aaabiao/dapo_filter`), filtering prompts to at most $2{,}000$ tokens; following Dr. MAS [3], the trailing "Let's think step by step ...\boxed{}" suffix is stripped at load time so the multi-agent workflow owns response formatting. Verbatim prompts are listed in Appendix A.11.

Training. Both policies are trained with M2PO (m^2-threshold 0.01), with group-level reward normalization, batch-level advantage normalization, and a fixed KL penalty coefficient of 10^{-3}

(no KL controller). We use AdamW with a constant learning rate of $5e{-}6$ (β_1=0.9, β_2=0.999, weight decay 0.01), no warmup, and gradient clipping at 1.0. The training batch size is 256 with 8 rollouts per prompt and 4 PPO mini-batches per iteration; rollout generation uses temperature 1.0 and $\max_{\text{new_tokens}} = 4{,}096$. We deliberately disable dynamic sampling in this run to align with the Dr. MAS [3] no-DS configuration that the Solver and Solver+Verifier (verl) baselines in Table 2 report against; this keeps the rollout regime matched across the three rows. Training runs for $1{,}200$ iterations on a single 8×NVIDIA H100 (80 GB) node, with 4 GPUs serving the two SGLang RaaS instances (Solver and Verifier each at data-parallel size 2) and 4 GPUs split into two FSDP trainer groups of size 2 (one per policy). Trainer↔RaaS weight transfer runs in TCP delta mode with a full-sync interval of 10 iterations.

Baseline and time measurement. We reproduce the verl-based Dr. MAS Solver–Verifier baseline and compare it with the corresponding AstraFlow workflow using the three-seed protocol described above. The primary comparison uses accuracy and iteration time obtained from these reproduced runs. Dr. MAS executes rollout generation and policy optimization sequentially in a colocated synchronous pipeline, whereas AstraFlow runs rollout services and trainers asynchronously. Iteration time is reported in seconds.

Evaluation. We evaluate on AIME24, AIME25, MATH500, and Minerva Math every 25 training iterations. Following the shared protocol in Section A.1, we sample 4 generations per question at temperature 0.6 and report pass@1 (avg@4).

A.3 Multi-policy Code (Section 4.1, Table 3)

Models & Datasets. The three rows of Table 3 (single-policy Solver, Solver + Selector, Solver + Test-Case Generator) all initialize from Qwen3-8B [41] and use the workflows described in Section 4.1; multi-agent runs train each role as a separate policy. We train on the DeepCoder-Preview-Dataset [18] (Hugging Face: `agentica-org/DeepCoder-Preview-Dataset`, `primeintellect` subset, `train` split) and filter prompts to at most $6{,}000$ tokens. Reward in all three runs is the binary outcome of executing the final attempt against the problem's hidden test cases. Verbatim Solver, Selector, and Test-Case Generator prompts are listed in Appendix A.11.

Training. All three runs use the same M2PO settings as Section A.2, except: learning rate $3e{-}6$, $1{,}000$ training iterations, dynamic sampling on (zero-advantage groups are filtered before training), and TCP *full*-sync weight transfer instead of delta. The training batch size is 256 with 8 rollouts per prompt; in the Solver + Selector run the Selector trainer uses a smaller batch size of 128, matching its lower per-iteration production rate. All runs use NVIDIA H200 (141 GB) GPUs. The Solver baseline runs on a single 8×H200 node, with 4 GPUs serving the policy (data-parallel size 4) and 4 GPUs as one FSDP trainer group of size 4. The two multi-agent runs each use two 8×H200 nodes (16 GPUs total): one node fully dedicated to RaaS hosting both policies, and the other node split into 4 RaaS GPUs and 4 FSDP trainer GPUs (two groups of size 2, one per policy). Rollout data-parallel sizes are 6 + 6 (Solver + Selector) and 9 + 3 (Solver + Test-Case Generator); the asymmetric v3 split reflects the Test-Case Generator's lower per-rollout invocation rate.

Evaluation. We evaluate on LiveCodeBench v5 [14], LiveCodeBench v6, and the Codeforces split of DeepCoder-Preview-Dataset every 25 training iterations; multi-agent runs are scored end-to-end through the same workflow used in training. Following the shared protocol in Section A.1, we sample 4 generations per question at temperature 0.6 and report pass@1 (avg@4).

A.4 Rollout Auto-scaling (Section 4.2.1, Table 4)

Models & Datasets. A single Qwen3-14B [41] policy is trained on the DeepScaler RL math set [19], filtering prompts to at most $2{,}000$ tokens.

Training. We use M2PO with m^2-threshold 0.002, group-level reward and batch-level advantage normalization, and a fixed KL penalty coefficient of 10^{-3} (no KL controller). We optimize with AdamW at a constant learning rate of $3e{-}6$ (β_1=0.9, β_2=0.999, weight decay 0.01), no warmup, and gradient clipping at 1.0. The training batch size is 256 with 8 rollouts per prompt and 4 PPO mini-batches per iteration; rollout generation uses temperature 1.0 with $\max_{\text{new_tokens}} = 18{,}000$ and SGLang context length $20{,}480$; dynamic sampling is on (zero-advantage groups are filtered before training). Training runs for $1{,}200$ iterations on two 8×NVIDIA H200 (141 GB) nodes (16

GPUs total). The trainer is fixed at 4 GPUs as a single FSDP group of size 4 on the main node; the remaining 4 GPUs on the main node and all 8 GPUs on the secondary node form the rollout pool. Each SGLang RaaS instance runs at tensor-parallel size 1 and is sized at data-parallel 1, 2, or 4 so that the controller can compose pool sizes by adding or retiring instances. Trainer↔RaaS weight transfer uses TCP full-sync mode.

Auto-scaling controller. We compare three rollout-pool configurations: a fixed pool of 6 GPUs, a fixed pool of 11 GPUs, and our auto-scaled pool, which the controller resizes between 6 and 11 GPUs. The dataflow layer exports the trainer waiting fraction w, recent production/consumption counts, and per-instance availability every $K = 10$ trainer steps. We use the three-zone policy from Section 4.2.1 with lower and upper waiting-fraction thresholds $\tau_{\text{low}} = 0.05$ and $\tau_{\text{high}} = 0.10$ and a shrink-margin factor of $\rho = 1.10$. The auto-scaling loop is executed by Claude Code as an agentic maintainer: at each report it reads the suggested G_{target} and issues the corresponding shell commands to launch or retire RaaS instances, preferentially removing unavailable or persistently low-throughput services on scale-down.

Evaluation. We evaluate on AIME24, AIME25, AMC, MATH500, and Minerva Math every 50 training iterations. Following the shared protocol in Section A.1, we sample 4 generations per question at temperature 0.6 and report pass@1 (avg@4).

A.5 Heterogeneous and Cross-region Training (Section 4.2.2)

Models & Datasets. A single Qwen3-14B [41] policy is trained on the DeepScaler RL math set [19], filtering prompts to at most $2{,}000$ tokens.

Training. We use M2PO with m^2-threshold 0.002, group-level reward and batch-level advantage normalization, and a fixed KL penalty coefficient of 10^{-3} (no KL controller). We optimize with AdamW at a constant learning rate of $3\text{e}{-6}$ (β_1=0.9, β_2=0.999, weight decay 0.01), no warmup, and gradient clipping at 1.0. The training batch size is 256 with 8 rollouts per prompt and 4 PPO mini-batches per iteration; rollout generation uses temperature 1.0 with $\max_{\text{new_tokens}} = 18{,}000$; dynamic sampling is on (zero-advantage groups are filtered before training). Training runs for $1{,}200$ iterations. All training settings (model, dataset, M2PO hyperparameters, batch, iterations, decoding, dynamic sampling) are identical to the auto-scaling experiment in Section A.4.

Homogeneous reference. We compare the heterogeneous and simulated cross-region deployment with a homogeneous local deployment using the same model and training configuration. The homogeneous reference allocates 12 local NVIDIA H200 GPUs to rollout generation. This deployment provides the reference for the direct comparison of accuracy and iteration time.

Cross-region topology and heterogeneity. We simulate a cross-region heterogeneous deployment over three NVIDIA H200 (141 GB) nodes: a 4-GPU FSDP trainer (data-parallel size 4) on the local node, and three 4-GPU SGLang RaaS pools (each at tensor-parallel size 1, data-parallel size 4) — one co-located with the trainer and two on remote nodes. GPU heterogeneity is induced by per-GPU power caps: 700 W on the local pool, and 400 W and 250 W on the two remote pools, yielding rollout-throughput shares of roughly 100%, 60%, and 30% respectively (Figure 6b). The two remote links are shaped with `tc/netem` to 4 Gbit/s effective bandwidth and 300 ms round-trip latency; the local link is unshaped. Trainer↔RaaS weight transfer uses TCP delta mode with a full-sync interval of 20 iterations; the periodic full syncs surface as the spikes in Figure 6c.

Evaluation. We evaluate on AIME24, AIME25, AMC, MATH500, and Minerva Math every 50 training iterations. Following the shared protocol in Section A.1, we sample 4 generations per question at temperature 0.6 and report pass@1 (avg@4).

A.6 Delta-sparsity Measurement (Section 4.2.2, Figure 7)

Models & Datasets. We instrument six independent RL training runs that span two axes: (i) model scale, training Qwen3-1.7B / 8B / 14B [41] on the DeepScaler math set [19]; (ii) task, training Qwen2.5-7B-Instruct [42] on AlfWorld, WebShop, and search-based QA via the ASearcher workflow. All runs filter prompts to at most $2{,}000$ tokens (math) or $4{,}096$ tokens (agentic).

Training. Each underlying run is trained with M2PO with the same shared base settings as Section A.2 (group reward / batch advantage normalization, fixed KL penalty 10^{-3}, AdamW with β_1=0.9,

β_2=0.999, weight decay 0.01, constant LR with no warmup, gradient clipping 1.0, 4 PPO mini-batches, dynamic sampling on, 8 rollouts per prompt at temperature 1.0); the per-run differences are summarized in Table 6. For the lr-ablation on Search, we additionally train the same workload at $\mathrm{lr} = 5\mathrm{e}{-6}$ to study the effect of larger updates on sparsity. Hardware is NVIDIA H100 (80 GB), with a 4-GPU FSDP trainer and a 4-GPU SGLang RaaS pool per run.

Table 6: Per-run hyperparameters for the delta-sparsity measurement runs.

Workload	Model	lr	m^2 thresh.	Batch	Iters	$\max_{\text{new_tok}}$
DeepScaler math	Qwen3-1.7B	$5\mathrm{e}{-6}$	0.01	256	800	14,000
DeepScaler math	Qwen3-8B	$5\mathrm{e}{-6}$	0.01	256	800	14,000
DeepScaler math	Qwen3-14B	$3\mathrm{e}{-6}$	0.002	256	1,200	14,000
AlfWorld	Qwen2.5-7B-Instruct	$1\mathrm{e}{-6}$	0.004	128	1,200	512
WebShop	Qwen2.5-7B-Instruct	$1\mathrm{e}{-6}$	0.004	256	1,200	512
Search (ASearcher)	Qwen2.5-7B-Instruct	$1\mathrm{e}{-6}$	0.004	256	1,000	1,024

Measurement. For each run we snapshot the trainer's bf16 weights at every iteration and compute, for each consecutive pair of snapshots, the fraction of parameters that are bit-exactly equal to the previous iteration. Figure 7 reports this fraction averaged over the first 500 training iterations, one bar per run. Per-iteration sparsity curves over the full training horizon are in Appendix B.2.

A.7 Performance vs. AReaL (Section 4.2.3, Table 5)

Models & Datasets. We compare AstraFlow against AReaL [5] on two math jobs at different model scales: a single Qwen3-1.7B policy and a single Qwen3-8B policy, both trained on the DeepScaler RL math set [19] with prompts filtered to at most 2,000 tokens. Both frameworks use the same model initialization, training data, training budget, and algorithm hyperparameters; only the framework implementation differs.

Training. Both jobs train with M2PO using the same configuration: m^2-threshold 0.01, group-level reward and batch-level advantage normalization, fixed KL penalty coefficient of 10^{-3} (no KL controller), AdamW with a constant learning rate of $5\mathrm{e}{-6}$ (β_1=0.9, β_2=0.999, weight decay 0.01), no warmup, and gradient clipping at 1.0. The training batch size is 256 with 8 rollouts per prompt and 4 PPO mini-batches per iteration; rollout generation uses temperature 1.0 with $\max_{\text{new_tokens}} = 14{,}000$ and SGLang context length 16,384; dynamic sampling is on (zero-advantage groups are filtered before training). Each job runs for 800 iterations on a single 8×NVIDIA H100 (80 GB) node, with 4 GPUs serving the policy (data-parallel size 4, tensor-parallel size 1) and 4 GPUs as one FSDP trainer group of size 4. Trainer↔RaaS weight transfer uses TCP full-sync mode. The AReaL baseline reuses the exact same configuration — model initialization, training data, training budget, M2PO hyperparameters, generation settings, hardware layout, and weight-transfer mode — on the same single 8×H100 node; only the framework implementation differs.

Evaluation. We evaluate on AIME24, AIME25, AMC, MATH500, and Minerva Math. Following the shared protocol in Section A.1, we sample 4 generations per question at temperature 0.6 and report pass@1 (avg@4). We additionally report two efficiency numbers: median per-iteration training time, and total wall time normalized by the number of training tokens (Time/1M tok), both averaged after the warmup phase.

Additional models: Llama. We additionally compare AstraFlow with AReaL using Llama 3.2-3B-Instruct and Llama 3-8B-Instruct. The configured context lengths are 16K and 8K tokens, respectively. Each run uses eight NVIDIA H200 GPUs. We evaluate on AIME24, AIME25, AMC, MATH500, and Minerva Math, and report benchmark accuracy, average accuracy across the five benchmarks, iteration time, and time per million tokens.

A.8 Data-algorithm Flexibility (Section 4.3, Figure 8)

Models & Datasets. A single Qwen3-8B [41] policy is trained on the DeepScaler RL math set [19], capped at the first 8,000 prompts and filtered to at most 2,000 tokens. We truncate the training set to 8,000 prompts (rather than the full split used in Sections A.4–A.7) because GRESO requires multiple passes over the same prompts to estimate per-prompt zero-variance probabilities and converge its

bucket-level submit-probability schedule; capping the set ensures every variant runs through the prompt stream multiple times within the 800-iteration budget. This is also why the y-axis in Figure 8 sits a few points below the same-model accuracy reported on the full DeepScaler set in Section A.7. All four variants share this identical underlying prompt stream so that any difference in accuracy or rollout cost reflects the data algorithm rather than the data.

Training (shared). All variants train with M2PO using the same configuration: m^2-threshold 0.01, group-level reward and batch-level advantage normalization, fixed KL penalty coefficient of 10^{-3} (no KL controller), AdamW with a constant learning rate of $5\text{e}{-6}$ (β_1=0.9, β_2=0.999, weight decay 0.01), no warmup, and gradient clipping at 1.0. The training batch size is 256 with 8 rollouts per prompt and 4 PPO mini-batches per iteration; rollout generation uses temperature 1.0 with $\max_{\text{new_tokens}} = 14{,}000$ and SGLang context length $16{,}384$. Each variant runs for 800 iterations on a single 8×NVIDIA H100 (80 GB) node, with 4 GPUs serving the policy (data-parallel size 4, tensor-parallel size 1) and 4 GPUs as one FSDP trainer group of size 4; weight transfer uses TCP full-sync mode.

Data-algorithm variants. Figure 8 compares three method families that compose dataflow-layer hooks at different intervention points: (i) *Vanilla* (1 point, r=0): the buffer applies the default `KeepAllFilter` and no replay or curator is enabled, so every prompt is submitted once per iteration and every generated trajectory is used. (ii) *DS + Replay* (4 points at replay ratios $r \in \{0.0, 0.3, 0.5, 0.7\}$): the post-rollout filter `filter_zero_adv` discards zero-advantage groups after generation [43], and the buffer reuses stored trajectories at training-batch serving time — a fraction r of each training batch is sampled from the replay pool (size $10{,}000$, max staleness 8). At r=0 this collapses to dynamic sampling alone (the rightmost point on the curve, ∼720k rollouts). (iii) *DS + Replay + GRESO* (2 points at $r \in \{0.3, 0.5\}$): on top of `filter_zero_adv` and replay, GRESO [49] layers a pre-rollout curator that admits each prompt with a per-bucket submit probability adapting toward configured easy/hard zero-variance targets. We use the GRESO hyperparameters from [49]: initial easy/hard exploration probabilities $0.5/0.5$, target zero-variance ratios α_{easy}=0.083 and α_{hard}=0.167, probability adjustment step Δp=0.01, per-prompt submit-probability floors 0.05 (easy) and 0.30 (hard), and a correctness threshold of 0.5 for easy/hard bucketing.

Evaluation. We evaluate on AIME24, AIME25, AMC, MATH500, and Minerva Math. Following the shared protocol in Section A.1, we sample 4 generations per question at temperature 0.6 and report the average pass@1 (avg@4) across the five benchmarks (the y-axis of Figure 8). The x-axis reports the cumulative number of generated rollouts over the 800 training iterations, exposing the rollout-cost / accuracy trade-off across data algorithms.

A.9 Fault-injection Experiment

We evaluate resilience to rollout-service failures and simulated region disconnections using Qwen3-8B. The initial deployment consists of a trainer on four NVIDIA H200 GPUs and four RaaS instances, each using two NVIDIA H200 GPUs. The trainer and dataflow service remain operational throughout the experiment.

At training iteration 34, we terminate one RaaS instance and launch a replacement at iteration 53. At iteration 74, we terminate two instances and disconnect another instance to simulate a region disconnection. We reconnect the disconnected instance at iteration 86 and launch two replacement instances at iteration 105. The run ends at iteration 125. Initial warmup iterations are excluded from the performance summaries, and generation throughput and iteration time are reported separately for each deployment stage.

A.10 Dataflow-service Microbenchmark

We evaluate a single dataflow-service instance in an isolated stress test using trajectory payloads, sequence lengths, and metadata representative of the Qwen3-8B workload. We measure sustained ingestion and serving throughput in sequences per second. We also report the p50 and p99 latency of retrieving a training batch, including data preprocessing and batch assembly. This microbenchmark characterizes the capacity of the dataflow service independently of end-to-end training performance.

A.11 System Prompts for Single- and Multi-Policy Training

This subsection lists the verbatim prompts used in the single-policy and multi-policy collaborative-training experiments of Section 4.1. The math and code paths use different multi-agent workflow structures: math runs use a *symmetric solver–verifier* pair, where both agents share an environment context plus a teammate-output block and are conditioned on role-specific instructions; code runs use an asymmetric *solver + test-case generator* pipeline, where the solver first attempts the problem, the test-case generator synthesizes additional cases against which the solution is executed, and the solver is asked to retry with concrete failure feedback.

Math prompts. For single-policy math, we append a short reasoning suffix to the user's question. For multi-policy math, the solver and verifier are instantiated from the same base model and trained jointly under the role-conditioned templates below; the placeholder {task_description} is filled with the raw problem text and {team_context} with the concatenated outputs of teammates accumulated so far in the rollout.

Single-Agent Math Suffix

Let's think step by step. Please put your final answer within \boxed{}.

Multi-Agent Solver/Actor Prompt (Math)

Task Introduction
You are a member of an expert multi-agent team tasked with solving the math problem. The team's math problem is:
{task_description}
Your Teammates' Outputs
{team_context}
Your Role
You are a "Solver Agent". Your job is to carefully reason through the math problem step by step and derive the correct answer. When reasoning, consider your teammates' outputs if available. Put the final answer in \boxed{}.

Multi-Agent Verifier Prompt (Math)

Task Introduction
You are a member of an expert multi-agent team tasked with solving the math problem. The team's math problem is:
{task_description}
Your Teammates' Outputs
{team_context}
Your Role
You are a "Verifier Agent". Your responsibility is to critically review the most recent solution provided by the "Solver Agent". First, carefully examine each reasoning step, formula, and conclusion for accuracy, completeness, and logical consistency. Explain your analysis in detail. After completing your analysis, at the very end of your output, you MUST provide your final verdict within <verify> </verify> using exactly one of:
(1) <verify>approve</verify> if all steps and the final answer are correct.
(2) <verify>reject</verify> if you detect any issue.
Important: Do NOT output your verdict until you have finished your full analysis.

Code prompts. The Code Solver prompt below is used as the user message for both single-policy and multi-policy code training; it is the only prompt the single-policy solver ever sees. We evaluate two multi-agent code workflows that pair the solver with a second trained model in different ways: (i) *solver + test-case generator*, where the test-case generator synthesizes cases against which the candidate code is executed and, on failure, the solver retries with concrete failing-case feedback; (ii) *solver + selector*, where the solver produces two independent candidates from the same Code Solver prompt and a separate selector LLM is trained to pick the better one. The test-case generator and

selector are each invoked with a system message that fixes their role and output discipline, paired with a user message supplying the problem statement and candidate(s); we list each system+user pair in a single box below, with role markers.

Code Solver Prompt (Single- and Multi-Agent)

Solve the following coding problem in Python 3.
Return only one final “‘python“‘ code block containing the complete solution.
Question:
{question}

Multi-Agent Code Solver Retry Prompt

Solve the following coding problem in Python 3.
Return only one final “‘python“‘ code block containing the complete solution.
Question:
{question}
Your previous solution failed on some test cases.
{feedback}
Now solve the problem and return the code.

Multi-Agent Code Test-Case Generator Prompt (Solver + Test-Case Generator workflow)

[System]
You are a testcase generator for a Python coding problem. Given the problem description (with built-in examples removed) and a candidate solution, produce exactly {generated_case_count} valid test cases that match the original interface. The cases must follow the problem statement, not the candidate solution. First think step by step about edge cases and potential bugs in the candidate inside an <analysis></analysis> block. Then output exactly one “‘json fenced code block containing the final test cases, and nothing after it.
[User]
Problem
{problem_text}
Candidate Solution
{code_text}
Required Output Format
Return JSON with exactly this shape for {generated_case_count} cases:
{
“fn_name”: “{fn_name}”,
“inputs”: [<case1_args>, <case2_args>],
“outputs”: [<case1_expected>, <case2_expected>]
}
Each entry in `inputs` must match the function-call format used by the original problem.

Multi-Agent Code Selector Prompt (Solver + Selector workflow)

[System]
You are a code selector. You will be shown a Python coding problem and two candidate solutions, Candidate A and Candidate B. Analyze both solutions for correctness: check the algorithm, edge cases, off-by-one errors, and whether each matches the problem specification. First think step by step inside an <analysis></analysis> block. Then output exactly one of <final>A</final> or <final>B</final> on its own line and nothing after it.
[User]
Problem
{problem_text}
Candidate A
```python
{code_text_A}
```
Candidate B
```python
{code_text_B}
```
Decide which candidate is more likely to be correct (it is possible that neither is fully correct, or that both are; in those cases pick the one you judge more trustworthy). Reason inside an <analysis></analysis> block, then emit exactly one of <final>A</final> or <final>B</final> on its own line and nothing after it.

A.12 System Prompts for Agentic Tasks

This subsection lists the prompts used by the agentic-task workloads referenced in Sections 4.2 and 4.3: AlfWorld and WebShop (both served via AgentBench task servers) and search-based QA (served via the ASearcher workflow). For AlfWorld and WebShop, the task server injects an instruction message followed by an in-context demonstration trajectory; below we list the system-level instructions verbatim and note where the few-shot / one-shot demos are appended.

AlfWorld. The task server first injects the task instruction below as a user message, then preloads a category-specific few-shot demonstration (one full successful trajectory per task type: `put`, `clean`, `heat`, `cool`, `examine`, `puttwo`) before the actual environment description ("Here is your task. <observation>").

AlfWorld Task Instruction

Interact with a household to solve a task. Imagine you are an intelligent agent in a household environment and your target is to perform actions to complete the task goal. At the beginning of your interactions, you will be given the detailed description of the current environment and your goal to accomplish. For each of your turn, you will be given a list of actions which you can choose one to perform in this turn. You should choose from two actions: "THOUGHT" or "ACTION". If you choose "THOUGHT", you should first think about the current condition and plan for your future actions, and then output your action in this turn. Your output must strictly follow this format: "THOUGHT: your thoughts.
ACTION: your next action
"; If you choose "ACTION", you should directly output the action in this turn. Your output must strictly follow this format: "ACTION: your next action
". After your each turn, the environment will give you immediate feedback based on which you plan your next few steps. If the environment output "Nothing happened", that means the previous action is invalid and you should try more options.
Reminder:
1. The action must be chosen from the given available actions. Any actions except provided available actions will be regarded as illegal.
2. Think when necessary, try to act directly more in the process.

WebShop. The task server injects the system-level prompt below, then pre-loads a one-shot demonstration of a complete shopping trajectory (search → click product → select size → buy) before the agent receives the actual instruction and observation.

WebShop Task Instruction

You are web shopping.
I will give you instructions about what to do.
You have to follow the instructions.
Every round I will give you an observation and a list of available actions, you have to respond an action based on the state and instruction.
You can use search action if search is available.
You can click one of the buttons in clickables.
An action should be of the following structure:
`search[keywords]`
`click[value]`
If the action is not valid, perform nothing.
Keywords in search are up to you, but the value in click must be a value in the list of available actions.
Remember that your keywords in search should be carefully designed.
Your response should use the following format:
Thought:
I think ...
Action:
click[something]

Search-based QA (ASearcher). The ASearcher workflow wraps the user question into a single rendered prompt that defines the search/answer protocol and pre-starts the assistant's response with a `<think>` tag. Our experiments use the search-only variant (the agent has access to a single `<search>` tool); a richer search-and-access variant that additionally exposes an `<access>` tool for page retrieval is also implemented in the codebase but is not used in this paper's evaluation.

Search-Only Prompt

A conversation between User and Assistant. The user asks a question, and the Assistant answers it. The Assistant analyzes the given question and information in the mind, retains important relevant information, calls a search engine to find necessary information, accesses web pages with certain urls, and provides the user with the answer. The Assistant conducts search by <search> query </search> and the top search results will be returned between <information> and </information>. The reasoning processes are enclosed within <think> </think>. Finally, the Assistant provides answer inside <answer> and </answer>, i.e. <answer> answer here </answer>. If there are multiple queries, ensure all answers are enclosed within <answer> </answer>, separated with comma. Note that when the Assistant finds the question is invalid, e.g. no answer could match all information in the question, the Assistant replies with '<answer> the question is invalid. </answer>'.
User:
{question}.
The language of your answer should align with the question. Assistant: <think>

B Additional Results

B.1 Accuracy of Agentic-Task Runs

The Qwen2.5-7B-Instruct training runs introduced for the delta-sparsity measurement (Section A.6) also produce trained policies on three agentic tasks; we report their accuracy here for completeness. For each run we follow the shared evaluation protocol of Section A.1 (4 generations per question at temperature 0.6, mean accuracy) and report the checkpoint with the best benchmark-average

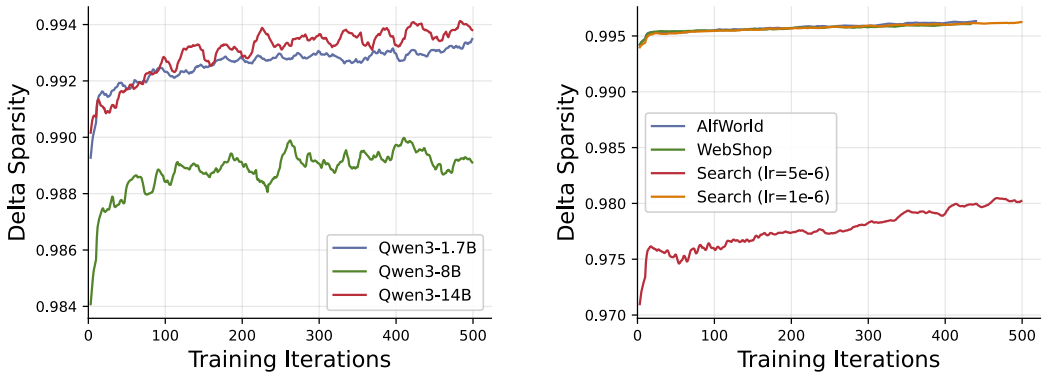
score. Search (ASearcher) is evaluated on four open-domain QA suites and we additionally show a per-benchmark breakdown.

Table 7: Accuracy (pass@1 (avg@4), %) of the Qwen2.5-7B-Instruct M2PO runs from Section A.6, reported at the best-average checkpoint of each run.

Task	**Benchmark**	**Acc. (%)**
AlfWorld	Valid	59.80
WebShop	Valid	93.63
Search (ASearcher)	Bamboogle	69.40
	HotpotQA	75.03
	PopQA	60.53
	TriviaQA	78.25
	Average	**70.80**

B.2 Per-Iteration Delta Sparsity

For completeness we also report the per-iteration sparsity curves underlying the averages in Figure 7. Each curve is averaged step-wise across overlapping sub-runs and then smoothed with a 15-iteration centered moving average; we restrict the x-axis to the first 500 training iterations.



(a) Math reasoning, Qwen3 [41] family.

(b) Tasks, Qwen2.5-7B-Instruct.

Figure 9: Per-iteration delta sparsity vs. training step, smoothed. (a) varies model scale on math (Qwen3-1.7B/8B/14B [41]); (b) varies task on Qwen2.5-7B-Instruct (AlfWorld, WebShop, Search at lr $= 1e{-}6$ and $5e{-}6$). The Search lr=$5e{-}6$ run is a clear outlier in (b); rerunning it at lr=$1e{-}6$ aligns it with AlfWorld and WebShop, confirming that learning-rate magnitude—not task type—drives the gap.

B.3 Implementation Effort and Extension Examples

Table 8 quantifies implementation effort for seven evaluated scenarios. Core-system LoC measures modifications to the trainer, RaaS, dataflow runtime, resource management, and weight-transfer implementation. User-side LoC measures application or algorithm semantics, including workflow interactions, reward logic, and data policies. All seven scenarios require zero core-system modifications; auto-scaling and heterogeneous cross-region deployment require configuration changes only.

Representative extension examples. Replay selection is exposed as a function over the current buffer size and requested batch size. The following excerpt implements selection of the most recent available entries:

Table 8: Implementation effort for the evaluated scenarios. User-side effort is distinguished from modifications to the core system.

Scenario	Core-system LoC modified	User-side effort
Math Solver–Verifier	0	426 LoC
Code Solver–Selector	0	496 LoC
Code Solver–Test-Case Generator	0	893 LoC
Auto-scaling	0	Configuration only
Heterogeneous and cross-region training	0	Configuration only
GRESO	0	230 LoC
Buffer replay	0	7 LoC

```
def select_latest(buffer_size: int, batch_size: int) -> list[int]:
    count = min(batch_size, buffer_size)
    start = max(0, buffer_size - count)
    return list(range(start, buffer_size))
```

The dataflow configuration composes replay, post-rollout filtering, and GRESO prompt selection through separate hooks:

```
dataflow:
  buffer:
    replay_ratio: 0.3
    max_staleness: 8
    filter_function: filter_zero_adv
  curator: "greso"
```

These excerpts illustrate the extension interfaces; the LoC measurements in Table 8 count the complete user-side implementations.

B.4 Cross-family Comparison on Llama Models

We compare AstraFlow and AReaL on Llama 3.2-3B-Instruct and Llama 3-8B-Instruct using the settings in Appendix A.7. Table 9 reports benchmark accuracy and both timing metrics.

Table 9: Cross-family comparison on the Instruct variants of Llama 3.2-3B and Llama 3-8B. Both frameworks use $8\times$H200 per run. Panel (a) reports accuracy in percent; panel (b) reports timing in seconds.

(a) Benchmark accuracy

Llama	Framework	AIME24	AIME25	AMC	MATH500	Minerva	Avg
3.2-3B	AReaL	12.29	1.04	14.58	56.00	21.97	21.18
	AstraFlow	13.33	0.83	14.44	56.00	23.53	21.63
3-8B	AReaL	0.00	0.83	6.67	30.20	14.71	10.48
	AstraFlow	1.04	0.83	10.00	31.70	18.57	12.43

(b) Training efficiency

Llama	Framework	Time/iter (s)	Time/1M tokens (s)
3.2-3B	AReaL	14.03	69.19
	AstraFlow	13.98	76.31
3-8B	AReaL	14.14	119.33
	AstraFlow	13.31	119.17

Average accuracy and iteration time are comparable across the two frameworks in these runs. For the 3B model, AstraFlow takes 76.31 s per million tokens versus 69.19 s for AReaL; for the 8B model, the corresponding times are 119.17 s and 119.33 s.

B.5 Rollout-service Fault-injection Results

Using the injection sequence in Appendix A.9, training continues through RaaS termination, replacement, disconnection, and reconnection without terminating the job. Table 10 reports the stage-level summaries.

Table 10: Rollout-service fault injection with the trainer and dataflow service remaining operational. Initial warmup iterations are excluded from the reported stage summaries.

Stage	RaaS instances	Generation throughput (k tokens/s)	Avg iteration time (s)
A	4	68.1	75.4
B	3	43.6	78.8
C	4	49.5	74.3
D	1	17.6	206.3
E	2	27.4	122.6
F	4	49.2	94.1

The one-instance stage has an iteration time of 206.3 s, compared with 75.4 s in the initial four-instance stage; the final four-instance stage has an iteration time of 94.1 s. These stage summaries span different points in the evolving training workload.

B.6 Dataflow-service Capacity and Recovery

Table 11 reports the isolated single-instance measurements described in Appendix A.10.

Table 11: Single-instance dataflow-service microbenchmark using payloads representative of the Qwen3-8B workload. Retrieval latency includes preprocessing and batch assembly.

Metric	Measurement
Sustained ingestion throughput	930 sequences/s
Sustained serving throughput	557 sequences/s
Training-batch retrieval latency, p50	0.41 s
Training-batch retrieval latency, p99	0.96 s

The service sustains 930 sequences/s for ingestion and 557 sequences/s for serving. The p50 and p99 training-batch retrieval latencies are 0.41 s and 0.96 s, respectively, including data preprocessing and batch assembly.

Implementation and recovery. The dataflow layer is a standalone service with an unsharded in-memory buffer that stores trajectories and metadata such as policy identity and weight version. Model weights are transferred separately from this service. The buffer and metadata are checkpointed with the training job. Recovery restores the trainer and dataflow state jointly from a consistent checkpoint; an isolated hot restart of the trainer is not currently supported.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction accurately reflect the paper's contributions and scope, including the proposed dataflow-oriented RL system and its evaluated capabilities.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper includes a discussion of limitations in the paper.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper provides the experimental details and settings needed to reproduce the main results.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The source code, environment setup, and training recipes are publicly available at `https://github.com/Infini-AI-Lab/astraflow`. The repository documentation describes data preparation and how to run the supported workloads.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: The paper specifies the training and evaluation settings in the main text and provides additional experimental details in the appendix.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: The math Solver–Verifier comparison reports mean and standard deviation over three independent training seeds for each framework (Table 2). These error bars capture training-run variability; avg@4 separately averages four responses per evaluation question. Three-seed error bars are limited to this math comparison.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The appendix reports the compute resources used in the evaluation.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have reviewed the NeurIPS Code of Ethics and confirmed that the paper conforms to it.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: The paper includes a broader-impact discussion covering potential positive and negative societal impacts.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release high-risk data or models requiring additional safeguards.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The paper properly cites the code, datasets, models, and other existing assets used in the experiments.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: AstraFlow is released as software under the Apache-2.0 license at `https://github.com/Infini-AI-Lab/astraflow`. The repository includes installation instructions, environment specifications, training recipes, and links to the documentation.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or research with human subjects, so IRB approval is not applicable.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: Large language models were used for paper editing and code assistance. In addition, Claude Code was used as the agentic maintainer in the rollout auto-scaling experiment, where it followed structured utilization reports to launch or retire RaaS instances as described in the paper.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.