# ALOER: ACTION-LEVEL ORCHESTRATION OF EXTERNAL RESOURCES FOR AGENTIC REINFORCEMENT LEARNING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Agentic reinforcement learning (RL) trains large language models (LLMs) through environment interactions and requires resources beyond the training cluster, such as CPUs for code execution and GPUs for reward models. Existing service systems for agentic RL typically rely on static resource provisioning, which mismatches the multi-turn and intermittent pattern of external resource invocations, leading to severe inefficiency. We present ALOER, a unified system for elastic, action-level orchestration of external resources in agentic RL. ALOER utilizes a unified action-level formulation and an elastic scheduling algorithm to minimize action completion time (ACT) while satisfying heterogeneous resource constraints. Further, heterogeneous resource managers are tailored to efficiently support the action-level execution with elastic degree-of-parallelism on heterogeneous resources. Evaluation on well-established agentic RL tasks demonstrates that ALOER improves average ACT by up to 4.3×, speeds up RL training steps by up to 1.5×, and reduces external resource usage by up to 71.2%.

## 1 INTRODUCTION

Agentic reinforcement learning (RL) trains large language models (LLMs) to solve tasks through interactions with external environments, using tools such as shell commands, APIs, and device operations (Cao et al., 2025). It has demonstrated significant performance improvements on complex real-world tasks such as AI coding (Le et al., 2022; Dou et al., 2024), deep search (Zheng et al., 2025; Qi et al., 2025), and embodied AI (Zitkovich et al., 2023; Wang et al., 2023). Agentic RL has thus become a critical component of LLM post-training.

Agentic RL demands substantial external resources, e.g., CPUs for AI coding, GPUs for reward models, and API quotas for web browsing, which introduce additional management requirements, paramount for two reasons. First, the latency of invocations is tightly coupled with the RL efficiency. Without effective external resource management, invocations can become slow or even blocked, hindering the training speed. Second, the additional cost of external resources is non-negligible. Maintaining many execution environments and hosting multiple reward models can require substantial CPU and GPU resources beyond those used for LLM training and inference. Thus, improving the utilization of these external resources and reducing the associated cost is undoubtedly important.

Notably, agentic RL's interaction with external resources is inherently *multi-turn* and *intermittent* (Figure 1): within each trajectory, the agent alternates between LLM generation and tool invocations over multiple turns; at the task level, the RL process alternates between rollout and training phases. Existing resource management and serving systems (kub, 2026; Zheng et al., 2024) typically provision resources for long-lived environments and services. These allocations can persist across idle intervals in agentic RL, leading to over-provisioning at two levels (Figure 3(b)): (1) At the trajectory level, systems reserve dedicated resources for the entire lifecycle of a trajectory, even though external resources are only accessed intermittently between LLM generation turns. For example, in AI coding, the environment is accessed only for 47% of its lifetime on average, leaving the allocated CPUs idle for the remaining time. (2) At the RL task level, tasks usually require specific external services and these services are deployed on isolated resources (Jiang et al., 2025; Microsoft, 2025). However, due to the fluctuating pattern of external invocations, these resources are under-



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

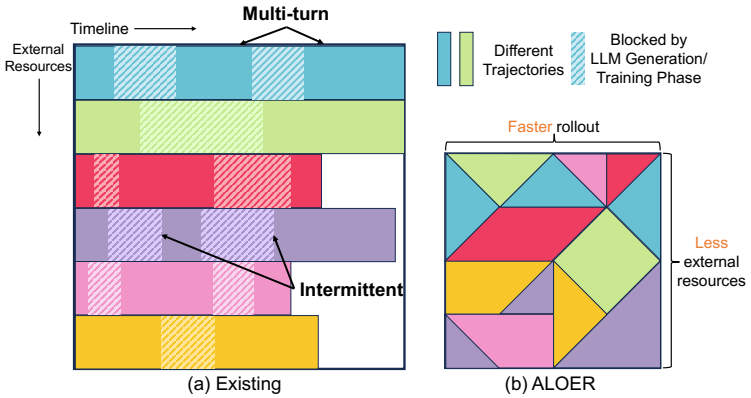



Figure 1: The multi-turn and intermittent pattern of external resource invocations in agentic RL, and comparison of existing approaches with ALOER.

utilized and wasted. In conclusion, previous static management limits system concurrency, leading to queuing delays and slower RL training.

To address these inefficiencies, we propose the **action-level** orchestration that shifts external resource management to the granularity of individual atomic invocations. We reclaim resources from long-lived environments or services and reallocate them to other actions requiring the same resource type. In addition, our fine-grained resource management enables elastic degree-of-parallelism (DoP), reducing execution latency. As illustrated in Figure 1, the action-level orchestration requires fewer external resources by mitigating the over-provisioning, and achieves faster rollout through elastic resource allocation.

We design ALOER, an action-level resource management system to uniformly orchestrate external resource invocations. First, it utilizes a unified action formulation to normalize various types of actions into a common format for scheduling. This formulation represents each action with a vectorized resource cost covering CPU, GPU, memory, and API quotas, and models elasticity and execution duration to estimate the marginal benefit of increasing its DoP. Also, we propose a heuristic scheduling algorithm to minimize the action completion time (ACT), while maintaining low overhead. The greedy eviction algorithm can orchestrate elastic actions online with one-shot DoP decisions, avoiding overly aggressive/conservative allocation strategies that lead to suboptimal ACT. ALOER tailors resource managers for heterogeneous resources, each with specialized mechanisms to release the resources after each action and restore the states of environments or services when invoked, and to allocate resources to mitigate fragmentation and improve parallel efficiency.

We evaluate ALOER on real-world agentic RL training tasks, and the experimental results show that ALOER improves average ACT by up to 4.3×, speeds up each RL step by up to 1.5×, and saves up to 71.2% of external resources.

In summary, this paper makes the following contributions:

- We identify external resource over-provisioning at the trajectory and RL-task levels in agentic RL.
- We design ALOER, an action-level resource management system combining a unified action formulation, elastic scheduling, and heterogeneous resource managers.
- We demonstrate improvements in ACT, end-to-end training efficiency, and external resource cost on real-world agentic RL tasks.

## 2 Background

### 2.1 Agentic RL Training

LLMs with agentic capacities can interact with the real world through external tools, e.g., shell commands, APIs, or device operations. As illustrated in Figure 2, the RL framework first uses an agentic LLM to process the samples and collects their execution trajectories, a phase known as *rollout* in RL. During the rollout phase, the framework typically follows the ReAct pattern (Yao et al., 2022), where the LLM decides how to use tools, and the RL framework accordingly performs

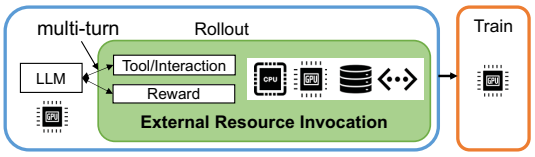



Figure 2: One training step of agentic RL.

tool invocations and gets the observation of the environment. Once a trajectory is completed, its reward is calculated and subsequently used to train the LLM.

While the LLM primarily consumes GPU resources, tool invocations rely on heterogeneous resource types including CPU, GPU, storage, and network, termed **external resources** from the perspective of the RL framework. Instead of invoking tools only once, agentic RL usually repeats the ReAct pattern over multiple turns, so frequent invocations of external resources are a pronounced characteristic of agentic RL. For example, AI coding repeatedly executes shell commands or edits files using CPUs, while DeepSearch accesses multiple websites and consumes API quotas. In addition, reward computation at the end of trajectories typically involves external resources as well: LLM-as-a-judge (Bai et al., 2022) is commonly deployed as an independent model service, and AI coding evaluates rewards by running a large number of test cases. All such operations that invoke resources outside the RL framework can be collectively referred to as **external resource invocations**.

External resource management is critical to both RL training efficiency and resource cost. First, external invocations lie on the critical path of each sample's rollout, so longer invocations prolong RL training. For the same RL task, reducing external resources from $1\times$ to $0.5\times$ increases both average ACT and step duration (Figure 3(a), Left). Second, external services can require substantial resources yet remain poorly utilized. For example, MOPD (Xiao et al., 2026; Ma et al., 2026) deploys dozens of domain-specific fine-tuned teacher models on hundreds of GPUs for reward computation. Figure 3(a) (Right) shows GPU SM activity across 12 teacher models: it fluctuates over time and across models, averaging below 3% and indicating substantial resource waste.

### 2.2 Over-Provisioning

The over-provisioning of external resources during agentic RL training can be categorized into two levels, both of which are derived from the intermittent patterns of agentic RL.

**Over-provisioning within trajectories.** To guarantee the continuity and independence of multi-turn invocations, some agentic RL tasks require an isolated and stateful environment for each trajectory to interact with. Existing systems (kub, 2026) statically allocate external resources for the entire lifecycle of a trajectory, although the trajectory only interacts with these resources within some time windows. We use a typical RL task in production, AI coding, to evaluate the action duration (including tool calling and reward calculation) of each trajectory. As shown in Figure 3(b) (Left), external invocations account for only 47% of each trajectory's rollout time on average.

**Over-provisioning within RL tasks.** Different RL tasks require task-specific external services, such as reward models with different architectures or parameters (Li et al., 2025; Jin et al., 2025a; Xiao et al., 2026). These services typically occupy separate resources (Microsoft, 2025; Jiang et al., 2025), which remain underutilized as invocation rates fluctuate during RL, and even fall idle during training. We profile the number of external invocations of two real agentic RL tasks, DeepSearch

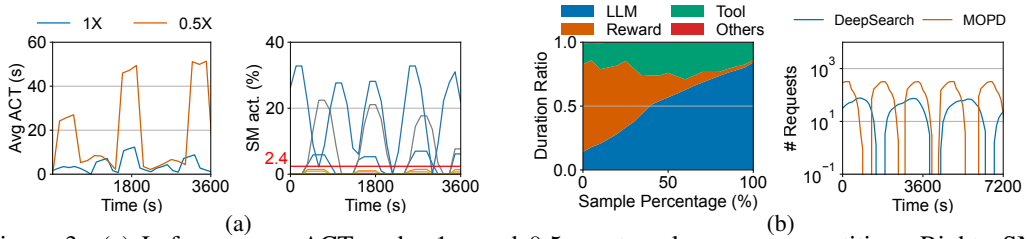



Figure 3: (a) Left: average ACT under $1\times$ and $0.5\times$ external resource quantities. Right: SM activity of 12 different reward services in MOPD. (b) Left: Code agent rollout duration ratio. Right: # External invocations of two agentic RL tasks.

and MOPD, as shown in Figure 3(b) (Right), where the number of invocations varies by up to three orders of magnitude.

### 2.3 Opportunities and Challenges

**Opportunity in Action-level Orchestration.** To address the inefficiency caused by over-provisioning, we propose to manage external resources at the **action level**. Here, an *action* refers to an atomic invocation of external resources that is not interleaved with LLM generation or other actions within the same trajectory. At this granularity, managed resources are allocated to actions from a unified pool and reclaimed after execution, rather than reserved for the lifetime of environments or services. Resources that would otherwise remain idle during LLM generation or training can thus serve other actions, thereby improving the utilization of external resources. Beyond reusing resources between actions, action-level allocation can elastically increase the degree of parallelism (DoP) of actions when spare capacity is available, shortening their execution.

**Challenges.** First, unified scheduling must accommodate heterogeneous resource requirements and execution behavior (§3.1). An action may consume several resource types, but satisfying its resource requirements alone does not reveal whether additional allocation will shorten execution. For example, parallel test execution can benefit from more CPUs, whereas API quotas constrain admission rather than execution speed. The challenge is to account for these differences within a common scheduler rather than requiring tool-specific scheduling logic.

Second, some existing approaches multiplex or scale service instances rather than accelerate individual actions through elastic DoP (Lorido-Botran et al., 2014; Wu et al., 2023). Methods that optimize per-job allocations may require costly decisions or iterative adjustment during execution (Peng et al., 2018; Bhardwaj et al., 2023; Qiao et al., 2021). Action durations follow a long-tailed distribution (e.g., actions in AI coding range from 0.01 s to 1200 s), with many actions leaving little time to amortize these costs and thus requiring fast, one-shot allocation decisions before execution (§3.2).

Third, action-level orchestration must decouple execution resources from persistent environment and service states (§4). Retaining allocations between invocations wastes resources, whereas destroying and recreating environments or services risks losing state or incurring substantial initialization overhead. Differences in resource characteristics and topology further complicate frequent, elastic allocation without excessive reconfiguration costs.

## 3 Unified Action-Level Formulation and Scheduling

We first formulate actions with heterogeneous external resources and job patterns in §3.1. Then we introduce our unified elastic resource scheduling algorithm in §3.2.

### 3.1 Action Formulation

**Vectorized Resource Cost Modeling.** Given an action, $a_i$, it may consume multiple types of resource simultaneously. For example, an action of DeepSearch may involve parallel or subsequent queries to multiple websites, where each website has a unique API quota and is considered as a different resource type. An action of the AI coding task involves not only CPUs but also memory. Thus, the cost of an action is represented as a multi-dimensional vector, denoted as $C_i = (c_{i,0}, c_{i,1}, ..., c_{i,k-1})$, where $k$ refers to the number of resource types managed by ALOER. The vector enables joint consideration of multiple resource constraints during scheduling and allocation. Each dimension of the vector corresponds to the consumption of one type of resource managed by ALOER (including CPU, GPU, memory, network bandwidth, and external API quotas).

Importantly, the cost for each resource type is not a static value. For actions that are elastic to the resource quantity, the cost is specified as a range between the minimum and maximum feasible resource usage. Moreover, for some tasks, such as reward services using multiple GPUs, the allowed resource quantities are discrete (e.g., 1, 2, 4, or 8 GPUs). Therefore, each component $c_{i,j}$ in $C_i$ specifies the feasible quantities of resource type $j$. This formulation allows the scheduler to flexibly adjust resource allocation without violating feasibility constraints, or to introduce some priors to narrow the search space during scheduling.

**Algorithm 1** Elastic Resource Scheduling Algorithm.

**Require:** Actions in the waiting queue $W$, vectorized resource status $R = (R_0, ..., R_{k-1})$

1: $Selected \leftarrow \varnothing$
2: $C \leftarrow W[:n],\ n = \max\{i \mid R.\text{accommodate}(W[:i])\}$ // Get candidates
3: **for** $j = 0 \rightarrow k-1$ **do** // Process actions sharing key elasticity resource $R_j$
4: $C_j \leftarrow C.\text{split}(R_j)$
5: **if** $\forall a_i \in C_j$: ($E_i$ is None or $S_i \equiv 0$) **then** // if elasticity unknown
6: $Selected += C_j$; **continue** // or zero, directly continue
7: $obj \leftarrow \text{getApproximatedObjective}(C_j, R_j)$ // Initialize objectives
8: **for** $t = 1 \rightarrow |C_j|$ **do** // Greedily evict the last action and compare
9: $newObj \leftarrow \text{getApproximatedObjective}(C_j[:-t], R_j)$
10: **if** $newObj >= obj$ **then break**
11: **else** $obj \leftarrow newObj$
12: $Selected += C_j[:-t+1]$; **continue**
**return** $Selected$

**Elasticity Modeling.** For elastic actions, an additional argument is introduced to describe their elastic behavior. In terms of a specific resource, its elasticity is described by a mapping from the $m$ units of the resources to an elastic ratio between 0 and 1. The function satisfies the following:

$$a.\text{getDur}(m) = \frac{T_{ori}}{E(m) * m} \quad (0 < E(m) \leq 1), \tag{1}$$

where $T_{ori}$ is the execution duration with a single resource unit. To simplify modeling and optimization, we assume that among all resource types consumed by an action, only one resource type is the key elasticity resource. For instance, GPU-based actions require sufficient GPU memory for execution, but their execution latency is primarily determined by the degree of parallelism. We evaluate $E(m)$ over feasible allocations of the key resource, $m \in c_{i,key}$ ($0 \leq key < k$).

### 3.2 Elastic Resource Scheduling

We first formulate the action completion time (ACT) as the optimization objective. Analogous to the job completion time (JCT), ACT is decomposed into: (1) the queuing duration before the action is scheduled; (2) the execution duration of the action. The scheduling objective is therefore to find the resource allocation among several actions that minimizes the sum of ACTs:

$$\text{Objective:} \quad \text{minimize } ACTs = \sum_{0 \leq i < N} (T_i^q + T_i), \tag{2}$$

where $N$ denotes the number of actions in the waiting queue at the scheduling time, $T_i$ denotes the actual execution duration of action $a_i$, and $T_i^q$ denotes its queuing duration. Then, we focus on the core problem: allocating finite units of resources among several elastic jobs and minimizing the total JCTs, i.e., determining the DoP for each action to acquire maximized efficiency, which is NP-hard (Bruno et al., 1974). According to the requirements discussed in § 2.2, we propose a unified heuristic scheduling algorithm, primarily based on a greedy eviction algorithm with approximated objective, as shown in Algorithm 1.

At each scheduling invocation, we select the longest prefix of the waiting queue as candidates $C$ whose minimum resource requirements jointly satisfy the capacity and topology constraints of all resource types (Lines 1–2). We then split the candidates into several groups according to the key elasticity resource (§ 3.1), and respectively apply the elastic allocation algorithm on each resource type (Line 3-4). Under the assumption that an action's elasticity is dominated by a single resource type, scaling an action along this resource does not affect the allocation of other resources. If all actions in a group are non-elastic or have unknown elasticity, the algorithm directly admits these actions (Lines 5–6). For elastic actions with known elasticity and execution durations, we initialize the objective with all candidates $C_j$ allocated their minimum required resources. We then iteratively evict the last candidate and reassign its resources to the remaining candidates. For each reduced candidate set, we compute the optimal allocation and evaluate the objective, stopping when further eviction no longer reduces it (Line 8-11).

We decompose objective approximation into two parts, the ACTs of the candidates and the ACTs of the remaining actions in the waiting queue; the former are determined exactly using a dynamic pro-

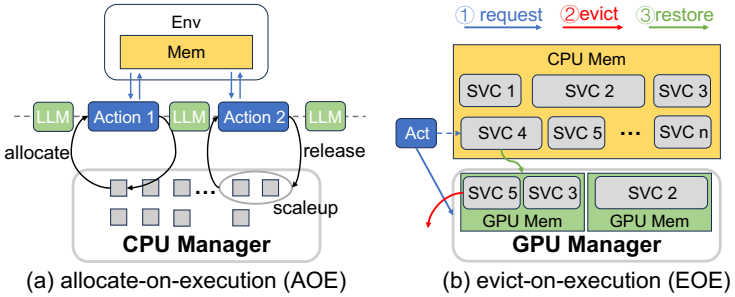



Figure 4: Heterogeneous resource management.

gramming (DP) algorithm (§B.2), whereas the latter are approximated (detailed in Appendix B.1). The computational complexity depends on the resource topology through the DP implementation. For a naive topology, the overall complexity is $O(kn^2m^2)$, where $k$ is the number of resource types, $n$ is the number of candidates, and $m$ is the number of remaining units of a resource type.

## 4 Heterogeneous Resource Managers

Because heterogeneous resources differ significantly in both characteristics and topologies, fast context switching and efficient resource pooling require resource-specific management policies. We design specialized resource managers for different resource types, including CPUs, GPUs, and other basic resources. For external resources that cannot be scaled up, such as website quotas and request QPS limits, we introduce *Basic Resource Manager* to enforce resource constraints. This manager also defines a set of basic interfaces that can be overridden to support action-level scheduling and unified resource management. Two representative resource managers are detailed as follows.

### 4.1 CPU Manager via AOE

The multi-turn, intermittent nature of agentic RL provides explicit action execution boundaries, which we exploit through a CPU management mechanism termed *allocate-on-execution (AOE)*. AOE modifies a container's cgroup configuration on demand. Before executing each action, CPU Manager updates the container's cgroup according to the resource units assigned by the scheduling algorithm. The container then executes the action under the updated cgroup constraints, which also apply to child processes forked by the entry process. After execution, CPU Manager reclaims the allocated CPU resources. To preserve the long-lived state of each environment, we retain the memory allocated to each container until its trajectory completes. This is feasible in modern clusters with abundant memory capacity; our evaluation confirms that memory does not become a bottleneck under ALOER's management.

Each trajectory is placed on a node with sufficient CPU and memory resources, and its actions remain on that node to preserve environment state. For each action, the manager explicitly assigns a set of CPU cores through `cpuset` to prevent interference across actions. To accommodate background processes within containers, we reserve a separate shared CPU pool on each node. Background processes present at container startup or remaining after an action completes are assigned to this pool. Although most background processes consume little CPU time, we cap their CPU usage to prevent excessive consumption of shared-pool resources.

Elastic DoP also requires action implementations to exploit the allocated resources. For example, we adapt reward-calculation actions to execute test cases in parallel according to the assigned DoP. CPU placement and action-adaptation details are provided in Appendix D.1.

### 4.2 GPU Manager via EOE

Compared with Docker-based CPU environments, long-lived services on GPU clusters incur significantly higher setup overhead. For example, a reward model service must compile kernels, establish communication groups, and, most critically, load model parameters into GPU memory (Zhang et al., 2025; Lou et al., 2026). Moreover, preserving service states exclusively in either GPU memory or

CPU memory is impractical: (1) GPU memory is both scarce and heavily contended, preventing all services from being persistently cached; (2) restoring services entirely from CPU memory incurs prohibitive model-parameter transfer overhead. We adopt a *hierarchical memory structure* combined with an *evict-on-execution (EOE)* mechanism to reduce context-switch overhead, along with resource allocation strategies to mitigate resource fragmentation and service cache thrashing.

During initialization, GPU Manager prepares all required services in turn by deploying them on GPUs and backing up their states in CPU memory. When an action requests a service, GPU Manager allocates a set of available GPUs and checks whether the requested service is already resident in GPU memory. If so, the action is executed immediately. Otherwise, GPU Manager restores the service from CPU memory, after evicting cached services from GPU memory as necessary until sufficient memory is available for exclusive execution. After the action completes, the service remains cached in GPU memory until a subsequent eviction. Prior work on reducing the overhead of restoring services from CPU memory (Zhang et al., 2025; Xiang et al., 2025) is orthogonal to our EOE mechanism and can be readily integrated. EOE naturally supports elastic DoP by treating different DoP configurations of a service as distinct services.

To mitigate GPU fragmentation, we adopt a multi-level cell structure (Zhao et al., 2020) that organizes GPUs into aligned, power-of-two *chunks*. GPU Manager selects the smallest available chunk that can accommodate a request, splitting larger chunks when necessary. Among same-sized chunks, it prefers one that caches the requested service and uses least-recently-used (LRU) eviction to reduce cache thrashing. Chunk definitions and runtime allocation details are provided in Appendix D.2; the GPU-specific DP operator is described in Appendix C.2.

# 5 Evaluation

We first evaluate ALOER using various agentic RL tasks to show its ACT efficiency (§5.2), and then highlight its scalability over RL batch size and external resource (§5.3). Finally, we investigate the design of ALOER scheduling algorithms (§5.4); analyses of RL convergence and system overhead are provided in Appendices E.2 and E.3, respectively.

## 5.1 Experiment Setup

**Testbed.** The testbed consists of two components: a 48-node GPU cluster running the agentic RL framework, and a set of heterogeneous external resources managed by ALOER. Implementation details and full experimental configurations are provided in Appendices D and E.1, respectively.

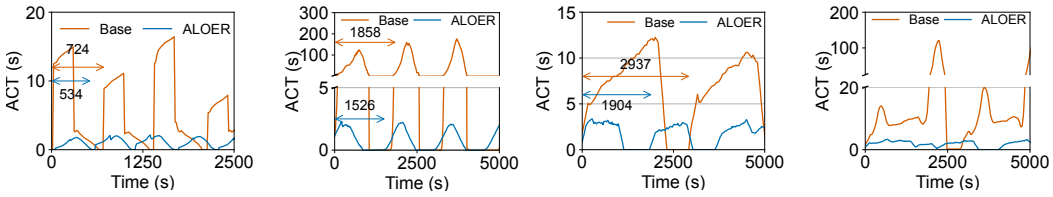
**Workloads.** We evaluate three representative workloads. **AI Coding** uses isolated CPU environments for intermittent code execution and reward computation. **DeepSearch** consumes API quotas for web-based question answering and uses a GPU-hosted LLM judge for rewards. Both use GRPO (Shao et al., 2024). **MOPD** (Xiao et al., 2026; Ma et al., 2026) integrates multiple RL tasks, including agentic tasks, and computes trajectory log-probabilities with GPU-hosted teacher models for alignment. We also run MOPD and DeepSearch concurrently (**MOPD+Search**) to evaluate GPU sharing across external services and mitigation of task-level over-provisioning.

**Baselines.** We use workload-specific baselines, as no existing system uniformly supports all three workloads. For **AI Coding**, Kubernetes creates one pod per trajectory with a 0.5-CPU request and a four-CPU limit, allowing multiplexing. For **MOPD** and **DeepSearch** rewards, SGLang (Zheng et al., 2024) hosts statically provisioned model services. DeepSearch trajectories independently issue API calls and retry failed or timed-out requests.

**Metrics.** We report average ACT and the durations of LLM generation, tool invocation, and reward computation. Stage durations are normalized by ALOER's total trajectory duration.

## 5.2 End-to-End Performance

Figure 5 reports average ACTs over consecutive time windows. With the same external resources, ALOER consistently lowers ACTs across workloads by mitigating over-provisioning. For DeepSearch, whose API calls are non-scalable, quota and concurrency control reduce rate-limit errors, timeouts, and retries.




(a) AI Coding. (b) MOPD. (c) DeepSearch. (d) MOPD+DeepSearch.

Figure 5: ACT under different RL tasks. The numbers and arrows indicate the step durations of RL tasks.

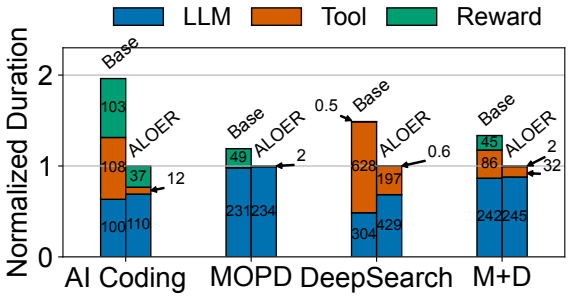



Figure 6: End-to-end performance for different RL tasks. Numbers indicate absolute durations (s).

We report **step duration** as the average duration of ten RL training steps. ALOER speeds up AI Coding and DeepSearch by $1.4\times$ and $1.5\times$, respectively. AI Coding benefits from action-level sharing that improves external CPU utilization. For DeepSearch, the speedup comes from traffic control rather than faster individual API execution: limiting concurrency reduces API failures, which otherwise incur retries and invalidate trajectories. MOPD gains are smaller because its pipelined *rollout* duration is dominated by long-tail trajectories rather than average ACT.

Figure 6 breaks down trajectory durations. ALOER reduces AI Coding's environment-interaction and reward-computation durations by $9.0\times$ and $2.8\times$, respectively ($4.3\times$ for external invocations overall). The former benefits from reduced queuing through action-level sharing, while the latter benefits from elastic DoP. MOPD benefits from pooling GPUs across teacher models serving partially interleaved RL tasks, unlike the baseline's fixed per-model allocation. DeepSearch's reward stage is slightly slower under ALOER because of restoration overhead when serving a single model. In the combined MOPD+Search workload, sharing again provides substantial gains.

### 5.3 System Scalability

We evaluate CPU and GPU scalability by varying **batch size** and **resource capacity**, measuring mean total ACT per trajectory. Here, RL batch size denotes the number of trajectories per batch.

**Scalability of CPU.** We also compare against K8S+VPA (Vertical Pod Autoscaler), which applies ALOER's scheduling hints through Kubernetes' in-place pod vertical scaling. With 1280 CPU cores across five nodes (Figure 7, left), ALOER reduces ACT by $3.1\times$–$27.7\times$ relative to K8S across batch sizes. At batch size 1536, K8S suffers frequent control-plane queuing timeouts, whereas ALOER continues serving with increased ACT. Varying CPU capacity at batch size 1280 (right), ALOER reduces ACT by $1.89\times$–$4.33\times$. K8S+VPA performs worse than K8S because per-pod CPU reconfiguration is too costly for frequent action-level adjustments.

**Scalability of GPU.** We also compare against ServerlessLLM (Fu et al., 2024), which scales service replicas with per-model request volume. On five external nodes with eight GPUs each (Figure 8, left), ALOER reduces ACT by $18.1\times$ relative to SGLang at batch size 2048, where ServerlessLLM fails to serve requests. At batch size 1024, the reductions relative to SGLang and ServerlessLLM are $3.4\times$ and $101.8\times$, respectively. At low concurrency, SGLang is slightly faster because ALOER incurs restoration overhead. At batch size 1024 (Figure 8, right), ALOER serves 10 reward services with only 29% of SGLang's GPUs at the same ACT. ServerlessLLM also reduces GPU usage, but service re-initialization during replica scaling or switching incurs greater overhead than ALOER's EOE mechanism; it also lacks per-request elastic DoP.

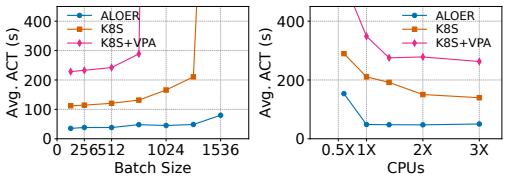



Figure 7: Scalability of CPU.

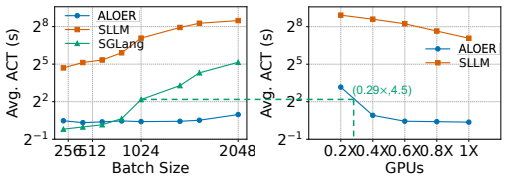



Figure 8: Scalability of GPU.

### 5.4 Analysis of ALOER

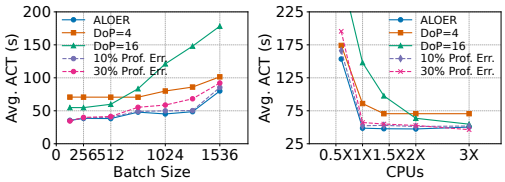



Figure 9: Impact of ALOER scheduling.

**Impact of ALOER Scheduling.** To control variability from LLM outputs, we compare schedulers using a benchmark built from a real AI Coding RL trace. Only reward-calculation actions are CPU-elastic, with long-tailed durations and parallelizable execution. We compare elastic DoP (1–32) against fixed DoP=4 and DoP=16, chosen empirically. In Figure 9, elastic scheduling consistently lowers ACT: by 2.0× versus DoP=4 at batch size 256, 3.0× versus DoP=16 at batch size 1280, and 1.8× versus DoP=4 with 1× CPU cores.

**Profiling Error Analysis.** We inject additional profiling errors of 10% and 30% to test scheduler robustness (Figure 9). ACT generally increases slightly, but even with 30% error, ALOER consistently outperforms both fixed-DoP baselines.

## 6 Related Work

**RL framework optimization.** RL frameworks improve training and rollout efficiency through colocated or disaggregated execution (Sheng et al., 2025; Hu et al., 2024), pipelining and streaming (Zhong et al., 2025b;a), and asynchronous execution (Fu et al., 2025). Further optimizations target request scheduling, load balancing, and speculative decoding during rollout (Gao et al., 2025; Jin et al., 2025b). These optimizations are complementary to ALOER, which manages the external resources consumed by tool invocations and reward computation.

**Agent resource management.** Mars (Zhu et al., 2026) exploits batch-level latency slack to scale reward workers, whereas interactive tool calls block subsequent LLM turns. MegaFlow (Zhang et al., 2026), SkyRL-Agent (Cao et al., 2025), and ROCK (Wang et al., 2025) emphasize task-level instance management, stage-level pipelining, and sandbox orchestration, respectively. In contrast, ALOER targets resource allocation for individual actions under the multi-turn and intermittent pattern of agentic RL, combining elasticity-aware scheduling with resource reclamation between invocations while preserving persistent state.

**Resource elastification.** Elastic resource management broadly falls into two categories. The first is *instance autoscaling*, which adjusts service instance counts to match demand in FaaS platforms (Agache et al., 2020; Knative Authors, 2022), other cloud services (Lorido-Botran et al., 2014; Wu et al., 2023; Wang et al., 2024; Wu et al., 2024; Zhang et al., 2024), and LLM serving (Zhang et al., 2025; Yu et al., 2025; Xiang et al., 2025; Zhao et al., 2025). Such scaling increases service capacity but does not accelerate individual action execution. The second is *elastic DoP*, which adjusts per-job resources to shorten execution, as in Pollux (Qiao et al., 2021), Optimus (Peng et al., 2018), Cilantro (Bhardwaj et al., 2023), and ElasticFlow (Gu et al., 2023). However, these systems incur costly allocation decisions or rely on iterative adjustment during long-lived jobs, making them ill-suited to the many short actions within agentic RL's long-tailed duration distribution.

## 7 Conclusion

ALOER orchestrates heterogeneous external resources for agentic RL, by formulating external interactions at the action level, scheduling them elastically, and devising heterogeneous resource managers, to enhance overall training efficiency while improving external resource utilization. Extensive evaluation shows that ALOER significantly shortens training step duration, lowers average ACT, and substantially reduces external resource cost, making agentic RL more efficient.

## 8 AI USE STATEMENT

In this work, we used generative AI tools to help implement methods, assist with translation, support qualitative and thematic data analysis, and interpret results. We did not use generative AI tools to generate synthetic datasets, help develop theoretical models or conceptual frameworks, formulate mathematical claims, propose or refine hypotheses, design or provide feedback on research methodology or experiments, or clean and reformat datasets. The tasks of providing critical ingredients for mathematical proofs and assisting with proof writing are not applicable to this work. Additionally, we used generative AI tools to create or modify scientific figures or images, create or edit software code, draft parts of a research paper, summarize or analyze existing literature, and edit the paper to improve readability. We have reviewed all AI-assisted work. LLM-generated code was reviewed and verified for correctness by 2 authors and LLM-assisted interpretations of the results have been reviewed for consistency with actual experimental results. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

## REFERENCES


Gojudge. https://docs.goj.ac/, 2026.

Kubernetes. https://kubernetes.io/, 2026.

Alexandru Agache, Marc Brooker, Andreea Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020*, pp. 419–434. USENIX Association, 2020.

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Romil Bhardwaj, Kirthevasan Kandasamy, Asher Addanki, Maolin Wu, Ilya Stober, Ion Stoica, and Joseph E. Gonzalez. Cilantro: Performance-aware resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023*, pp. 623–643. USENIX Association, 2023.

James Bruno, Edward G Coffman Jr, and Ravi Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 1974.

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.

Shihan Dou, Yan Liu, Haoxiang Jia, Limao Xiong, Enyu Zhou, Wei Shen, Junjie Shan, Caishuang Huang, Xiao Wang, Xiaoran Fan, et al. Stepcoder: Improve code generation with reinforcement learning from compiler feedback. *arXiv preprint arXiv:2402.01391*, 2024.

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.

Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-Latency serverless inference for large language models. In *USENIX OSDI*, 2024.

Wei Gao, Yuheng Zhao, Dakai An, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Ju Huang, Weixun Wang, Siran Yang, Wenbo Su, et al. Rollpacker: Mitigating long-tail rollouts for fast, synchronous rl post-training. *arXiv preprint arXiv:2509.21009*, 2025.

Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2023*, pp. 266–280. ACM, 2023.

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*, 2024.

Dongfu Jiang, Yi Lu, Zhuofeng Li, Zhiheng Lyu, Ping Nie, Haozhe Wang, Alex Su, Hui Chen, Kai Zou, Chao Du, et al. Verltool: Towards holistic agentic reinforcement learning with tool use. *arXiv preprint arXiv:2509.01055*, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The International Conference on Learning Representations (ICLR)*, 2024.

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*, 2025a.

Chao Jin, Yinmin Zhong, Zili Zhang, Yimin Jiang, and Yibo Zhu. Efficient rl for llms with dynamic and online speculative decoding. In *Frontier AI Systems Workshop (FAISys)*, 2025b.

Knative Authors. Knative serving: Autoscaling, 2022. URL https://knative.dev/docs/serving/autoscaling/.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, 2022.

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

Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 2014.

Chiheng Lou, Sheng Qi, Chao Jin, Dapeng Nie, Haoran Yang, Yu Ding, Xuanzhe Liu, and Xin Jin. HydraServe: Minimizing cold start latency for serverless LLM serving in public clouds. In *23rd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2026*. USENIX Association, 2026.

Wenhan Ma, Jianyu Wei, Liang Zhao, Hailin Zhang, Bangjun Xiao, Lei Li, Qibin Yang, Bofei Gao, Yudong Wang, Rang Li, Jinhao Dong, Zhifang Sui, and Fuli Luo. MOPD: Multi-teacher on-policy distillation for capability integration in LLM post-training, 2026. URL https://arxiv.org/abs/2606.30406.

Microsoft. Multi-agent reference architecture. https://github.com/microsoft/multi-agent-reference-architecture, 2025.

Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018*, pp. 3:1–3:14. ACM, 2018.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, et al. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. In *The International Conference on Learning Representations (ICLR)*, 2025.

Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021*, pp. 1–18. USENIX Association, 2021.

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

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. In *EuroSys*, 2025.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Weixun Wang, XiaoXiao Xu, Wanhe An, Fangwen Dai, Wei Gao, Yancheng He, Ju Huang, Qiang Ji, Hanqi Jin, Xiaoyang Li, et al. Let it flow: Agentic crafting on rock and roll, building the rome model within an open agentic learning ecosystem. *arXiv preprint arXiv:2512.24873*, 2025.

Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y Yan. Autothrottle: A practical bi-level approach to resource management for SLO-targeted microservices. In *USENIX NSDI*, 2024.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*, 2025.

Bingyang Wu, Kun Qian, Bo Li, Yunfei Ma, Qi Zhang, Zhigang Jiang, Jiayu Zhao, Dennis Cai, Ennan Zhai, Xuanzhe Liu, et al. Xron: A hybrid elastic cloud overlay network for video conferencing at planetary scale. In *ACM SIGCOMM*, 2023.

Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric Friedman, Scott Shenker, and Ion Stoica. Can't be late: optimizing spot instance savings under deadlines. In *USENIX NSDI*, 2024.

Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. Aegaeon: Effective gpu pooling for concurrent llm serving on the market. In *ACM SOSP*, 2025.

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

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.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International conference on learning representations (ICLR)*, 2022.

Minchen Yu, Rui Yang, Chaobo Jia, Zhaoyuan Su, Sheng Yao, Tingfeng Lan, Yuchen Yang, Yue Cheng, Wei Wang, Ao Wang, et al. $\lambda$ Scale: Enabling fast scaling for serverless large language model inference. *arXiv preprint arXiv:2502.09922*, 2025.

Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. BlitzScale: Fast and live large model autoscaling with O (1) host caching. In *USENIX OSDI*, 2025.

Lei Zhang, Mouxiang Chen, Ruisheng Cao, Jiawei Chen, Fan Zhou, Yiheng Xu, Jiaxi Yang, Liang Chen, Changwei Luo, Kai Zhang, et al. Megaflow: Large-scale distributed orchestration system for the agentic era. *arXiv preprint arXiv:2601.07526*, 2026.

Zili Zhang, Chao Jin, and Xin Jin. Jolteon: Unleashing the promise of serverless for serverless workflows. In *USENIX NSDI*, 2024.

Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis CM Lau, Yuqi Wang, Yifan Xiong, et al. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *USENIX OSDI*, 2020.

Yihao Zhao, Jiadun Chen, Peng Sun, Lei Li, Xuanzhe Liu, and Xin Jin. Seallm: Service-aware and latency-optimized resource sharing for large language model inference. *arXiv preprint arXiv:2504.15720*, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. In *Advances in neural information processing systems*, 2024.

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. *arXiv preprint arXiv:2504.03160*, 2025.

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*, 2025a.

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 *USENIX NSDI*, 2025b.

Ruidong Zhu, Mingcong Han, Yinmin Zhong, Wencong Xiao, Xuanzhe Liu, and Xin Jin. Towards efficient reward service for rlvr with request-level flexibility and batch-level constraint. In *USENIX NSDI*, 2026.

Brianna Zitkovich, Tianhe Yu, Sichun Xu, Peng Xu, Ted Xiao, Fei Xia, Jialin Wu, Paul Wohlhart, Stefan Welker, Ayzaan Wahid, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning*, 2023.

# A System Architecture

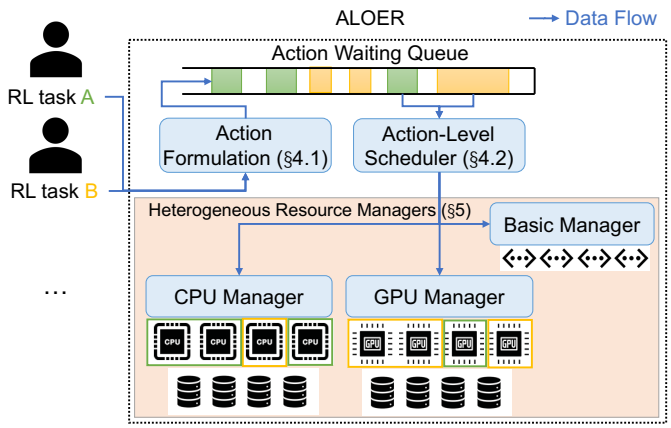



Figure A.1: System overview of ALOER.

To address the multifaceted over-provisioning issues inherent in current agentic RL training—within trajectories and within RL tasks—we propose a unified action-level platform, ALOER. Our architecture decouples external resource management from the primary RL training cluster, shifting the granularity of control from long-lived trajectories to individual atomic interactions, or actions.

As shown in Figure A.1, the system serves as a centralized intermediary between diverse RL Tasks and heterogeneous external resources. The workflow follows a standardized execution cycle: (1) Action Submission: During the rollout phase, when an agentic LLM invokes a tool, or reward calculation with external resources, it submits a request to ALOER. (2) Unified Formulation & Queuing: Through formulation, ALOER translates various external resource invocations into unified actions. Hence, these requests are submitted to a Unified Action Queue. (3) Elastic Scheduling: The scheduler schedules the actions in the waiting queue under the constraints of external resources, and coordinates with specialized Resource Managers (e.g., CPU, GPU, API) to elastically allocate the resources based on real-time system state and action formulation. (4) Action Execution: After the units of resources are decided, actions are independently executed on the resources allocated by heterogeneous resource managers. (5) Transmit & Observation: Once an execution is complete, the results are transmitted back to the RL framework, allowing it to proceed with the next LLM generation.

There are three main components in ALOER—the action formulation module(§3.1), the action-level scheduler(§3.2), and heterogeneous resource managers(§4). The foundation of our architecture treats every atomic invocation as an action. Thus, the formulation module models the costs and performance elasticity of various actions in a unified way. Then, the scheduler decides whether an action can be scheduled under the resource constraints, and determines DoP elastically to minimize the action completion time (ACT). In this way, ALOER reduces the time LLMs spend waiting for external resource invocations, further accelerating the RL training. With allocation decisions, heterogeneous resource managers are responsible for efficient resource allocation and sharing. Using a unified resource interface, our resource managers, including Basic Manager, CPU Manager and GPU Manager, employ domain-specific techniques to improve system robustness, resource utilization, and action speed.

# B ELASTIC SCHEDULING DETAILS

## B.1 ACT APPROXIMATION

Following the scheduling procedure in Section 3.2, Algorithm 2 computes the objective used by Algorithm 1. It combines the exact ACTs of the selected candidates with the estimated ACTs of the remaining actions.

Because the candidates are to be scheduled at once, their ACTs are computed exactly using their determined allocations. We employ a topology-agnostic dynamic programming (DP) algorithm that is applicable to heterogeneous resource topologies, termed as DPArrange in the pseudo code. This algorithm, provided in Appendix B.2, resolves the optimal discrete allocation among these elastic candidates and updates the completion heap according to the allocation. However, ACTs of the remaining actions are influenced by the allocation strategy ahead of them. It is intractable to recursively compute the best allocation and attain the optimal objective for these actions. Instead, we simplify the process and estimate their ACTs by assigning minimum resources and sequentially inserting them into the completion heap.

Notably, to approximate the objective more precisely, we introduce a parameter $depth$, which expands the search space by allowing the first remaining action to explore multiple allocation choices. In practice, setting $depth = 2$ or 3 is sufficient.

Moreover, the conditions require known execution durations only for elastic actions, rather than all actions. This is because, during greedy eviction of candidates, the durations of non-elastic actions remain constant, and do not affect the relative comparison of objectives. Although the exact execution durations of non-elastic actions affect the completion heap, we can approximate them using historical averages because elastic actions typically last much longer and dominate the evolution of the completion heap.

**Algorithm 2** ACTs Approximation.

**Require:** The waiting queue of actions $W$, the executing queue $E$, the $depth$ of performing approximation of objective

**function** GETAPPROXIMATEDOBJECTIVE($C_j, R_j$)
  $AC_j \leftarrow W$.split($R_j$) $- C_j$ // Actions to be scheduled the next step
  $exactObj,\ newHeap \leftarrow$ DPArrange($R_j, C_j$)
  $approxObj \leftarrow newHeap$.estimate($AC_j$)
  **return** $exactObj + approxObj$

**function** ESTIMATE($heap, C$)
  $obj \leftarrow \inf,\ heap \leftarrow$ copy($heap$)
  **for** $d = 1 \rightarrow depth$ **do**
    $T_0 \leftarrow C[0]$.getDur($d$), $ts \leftarrow heap$.pop()
    $tmpObj \leftarrow 0,\ tmpHeap \leftarrow$ copy($heap$)
    $tmpObj += ts + T_0,\ tmpHeap$.push($ts + T_0$)
    **for** $i = 1 \rightarrow |C|$ **do**
      $T_i \leftarrow C[i].T_{ori},\ ts \leftarrow heap$.pop()
      $tmpObj += ts + T_i,\ tmpHeap$.push($ts + T_i$)
    **if** $tmpObj < obj$ **then** $obj \leftarrow tmpObj$
  **return** $obj$

## B.2 TOPOLOGY-AGNOSTIC DPARRANGE

Solving the optimal discrete allocation among several scalable candidates and available units is a classical variant of the multi-dimensional bin packing problem. However, heterogeneous resource topologies require topology-specific allocation primitives. Algorithm 3 provides the unified, topology-agnostic DPArrange routine used in Section 3.2, with resource-specific behavior supplied by a DP operator. The corresponding basic/CPU and GPU operators are presented in Appendix C, in the same order as the resource managers in the main text.

**Algorithm 3** Topology-Agnostic DPArrange

**Require:** A set of scalable tasks $C = \{c_1, \ldots, c_m\}$, available resource units $U$, supported unit sets $\mathcal{S} = \{S_1, \ldots, S_m\}$, resource manager $\mathcal{M}$, and its DP operator $\mathcal{O}$
**Ensure:** Optimal execution time $T^\star$, allocation $\{k_1, \ldots, k_m\}$

$n \leftarrow \min\{\text{END}(U),\ \text{END}(\mathcal{S})\}$
**if** $n < m$ **then**
  **return error** // Each task requires at least one unit
Initialize DP table $dp[i][j] \leftarrow \infty$ for $0 \le i \le m,\ 0 \le j \le n$
Initialize choice table $choice[i][j] \leftarrow 0$
$dp[0][0] \leftarrow 0$
$start_{prev} \leftarrow 0$
$start_{cur} \leftarrow \mathcal{O}.\text{START}(S_1)$
**for** $i = 1 \to m$ **do**
  **for** $j = start_{cur} \to n$ **do**
    **for all** $k \in S_i$ **do** // Units assigned to task $c_i$
      $j' \leftarrow \mathcal{O}.\text{PREV}(j, k)$
      **if** $j' < start_{prev}$ **or** $\neg\mathcal{O}.\text{ISVALID}(j', S_{1:i-1})$ **then**
        **continue**
      $T_i(k) \leftarrow \mathcal{O}.\text{GETDURATION}(c_i, k)$
      **if** $dp[i-1][j'] + T_i(k) < dp[i][j]$ **then**
        $dp[i][j] \leftarrow dp[i-1][j'] + T_i(k)$
        $choice[i][j] \leftarrow k$
  $start_{prev} \leftarrow start_{cur}$
  $start_{cur} \leftarrow \mathcal{O}.\text{START}(S_{1:i+1})$
$j \leftarrow n$
**for** $i = m \to 1$ **do**
  $k_i \leftarrow choice[i][j]$
  $j \leftarrow \mathcal{O}.\text{PREV}(j, k_i)$
**return** $dp[m][n],\ \{k_1, \ldots, k_m\}$

# C Resource-Specific DP Operators

## C.1 Basic and CPU DP Operator

With the topology-agnostic DPArrange algorithm, each resource manager only needs to implement the primitives of its DP operator. Algorithm 4 defines the basic operator applicable to the CPU manager in Section 4.

**Algorithm 4** Basic DP Operator for CPU Resources

**function** START($\mathcal{S}$)
  **return** $\sum_i \min(S_i)$
**function** END($X$)
  **if** $X$ is a unit list **then**
    **return** $|X|$
  **else**
    **return** $\sum_i \max(S_i)$
**function** PREV($j, k$)
  **return** $j - k$
**function** ISVALID($r, X$)
  **if** $r < 0$ **then**
    **return false**
  **if** $X = \emptyset$ **then**
    **return** $(r = 0)$
  **for all** $u \in X_1$ **do**
    **if** $u \le r$ **and** ISVALID$(r - u, X_{2:})$ **then**
      **return true**
  **return false**

### C.2 GPU-TOPOLOGY-AWARE DP OPERATOR

Algorithm 5 represents GPU states by counts of chunks of sizes $\{1, 2, 4, 8\}$. For each size, $count_{size}$ and $limit_{size}$ denote the available and maximum chunk counts. Mixed-radix encoding maps each count vector to a unique scalar DP index. START encodes the minimum required chunk counts for the supported allocations $\mathcal{S}$. PREV greedily consumes existing chunks from largest to smallest, returning the encoded residual state or $\perp$ if the request cannot be met. This routine does not perform the runtime chunk splitting described in Appendix D.2.

**Algorithm 5** GPU-Topology-Aware DP Operator

**Require:** Maximum chunk counts $\mathbf{limit}$ for $sizes = (1, 2, 4, 8)$
State: $\mathbf{count} = (count_1, count_2, count_4, count_8)$, with $0 \le count_{size} \le limit_{size}$
Precompute $stride_{size} = \prod_{smaller \in sizes,\ smaller < size} (limit_{smaller} + 1)$; the empty product is 1.

**function** ENCODE($\mathbf{count}$)
  **return** $\sum_{size \in sizes} count_{size} \cdot stride_{size}$
**function** DECODE($index$)
  **for** $size \in (1, 2, 4)$ **do**
    $count_{size} \leftarrow index \bmod (limit_{size} + 1)$
    $index \leftarrow \lfloor index / (limit_{size} + 1) \rfloor$
  $count_8 \leftarrow index$
  **return count**
**function** START($\mathcal{S}$)
  $\mathbf{count} \leftarrow$ minimum required chunk counts inferred from supported allocations $\mathcal{S}$
  **return** ENCODE($\mathbf{count}$)
**function** ISVALID($index$)
  **return** $0 \le index \le$ ENCODE($\mathbf{limit}$)
**function** PREV($index, request$)
  $\mathbf{count} \leftarrow$ DECODE($index$), $\quad need \leftarrow request$
  **for** $size \in (8, 4, 2, 1)$ **do**
    $take \leftarrow \min(count_{size}, \lfloor need / size \rfloor)$
    $count_{size} \leftarrow count_{size} - take$
    $need \leftarrow need - size \cdot take$
  **if** $need > 0$ **then**
    **return** $\perp$
  **return** ENCODE($\mathbf{count}$)

# D IMPLEMENTATION

ALOER is a standalone service platform designed to orchestrate external resource invocations of agentic RL training and applicable to all RL training frameworks. Through collaboration with agentic RL frameworks, ALOER supports standard interaction protocols, including the Model Context Protocol (MCP) and OpenAI function calling, enabling seamless integration with existing agent pipelines. To facilitate rapid development, ALOER provides an *Action Development Kit (ADK)*. The ADK defines a set of extensible interfaces that support the introduction and evolution of *Tools*, *Interactions*, and *Reward Services*. By extending the OpenAI function schema, developers can optionally specify elasticity, resource requirements, and other properties for user-defined actions, demonstrating the generality and flexibility of ALOER.

ALOER is implemented as a lightweight and developer-friendly system, consisting of over 8,000 lines of code, with the convenience of RL developers and researchers in mind. To support the high-concurrency service of actions, ALOER leverages Ray as its runtime to achieve multi-process execution and distributed coordination.

Users can flexibly configure resource capacities and topology for different resource types, as well as customize scheduling policies and parameters, allowing ALOER to adapt to diverse cluster environments and workload characteristics.

## D.1 CPU MANAGER IMPLEMENTATION

The CPU Manager is implemented on top of a Docker-like virtualization mechanism, GoJudge (goj, 2026), enabling isolation and fine-grained CPU resource control.

When the first action of a trajectory is invoked, CPU Manager identifies nodes with sufficient CPU cores for the action and sufficient memory, then selects a node according to a memory load-balancing policy. All subsequent actions in the same trajectory execute on the selected node. Modern cloud CPU nodes typically provide at least 128 cores, substantially more than the number allocated to a single action. CPU Manager therefore schedules actions independently on each node. In addition, parallel efficiency depends on inter-core communication overhead, which is affected by the physical distances between cores on the chip. When allocating CPU cores to elastic actions, CPU Manager therefore prioritizes cores within the same non-uniform memory access (NUMA) node.

Beyond supporting elastic DoP at the operating-system level, we also ensure that action implementations support variable degrees of parallelism when constructing RL datasets and submitting actions. For example, in code generation tasks, test cases can be efficiently executed in parallel, yet the testing commands provided in existing datasets typically support only single-process execution. For test commands that use `pytest`, we replace `pytest` with `pytest -n ${parallel_num}` and dynamically set `parallel_num` at runtime to enable parallel execution.

## D.2 GPU MANAGER IMPLEMENTATION

GPU Manager adopts the SGLang (Zheng et al., 2024) engine to deploy reward services and utilizes `TorchMemorySaver` to back up GPU-resident states into CPU memory, supporting efficient eviction and restoration.

To accommodate common DoPs, we define a valid allocation unit, termed a *chunk*, as a contiguous GPU interval $(start, end)$ satisfying $end - start = 2^a$ and $2^a \mid start$, where $a \in \{0, 1, 2, 3\}$ denotes the chunk's level. Initially, each GPU node is represented by a single available level-3 chunk $(0, 8)$. To allocate $m$ GPUs, where $2^{a-1} < m \leq 2^a$, GPU Manager examines available chunks in ascending order of level and selects the smallest chunk whose level $b$ satisfies $b \geq a$. If $b > a$, it splits the selected chunk into valid chunks and returns a resulting level-$a$ chunk.

# E EVALUATION

## E.1 EXPERIMENTAL SETUP DETAILS

**Testbed.** We use VeRL (Sheng et al., 2025) with external resource invocation and asynchronous, sequence-level *rollout*. The RL framework runs on a production cluster of 48 GPU nodes, each equipped with eight NVIDIA Hopper-architecture GPUs and interconnected via high-bandwidth NVLink and RDMA networks. The external CPU cluster contains 15 nodes, each with 256 AMD CPU cores and 2.4 TB of memory. The external GPU cluster consists of five nodes, each equipped with eight high-end NVIDIA GPUs and 3 TB of CPU memory. External APIs include Google Search and PDF parsing services with enforced quotas.

**Workload configurations.** AI Coding uses an in-house dataset following the scaffold of SWEBench (Jimenez et al., 2024). DeepSearch uses a dataset based on BrowseComp (Wei et al., 2025), with a finetuned GPT-OSS (Agarwal et al., 2025) judge computing rewards over generated trajectories. For each workload in the end-to-end experiments, ALOER and the baseline use the same fixed batch size and external resource capacity. Batch sizes are 1280, 2048, and 2048 for AI Coding, MOPD, and DeepSearch, respectively. We use Qwen3-32B (Yang et al., 2025) for the RL training models and the teacher models in MOPD.

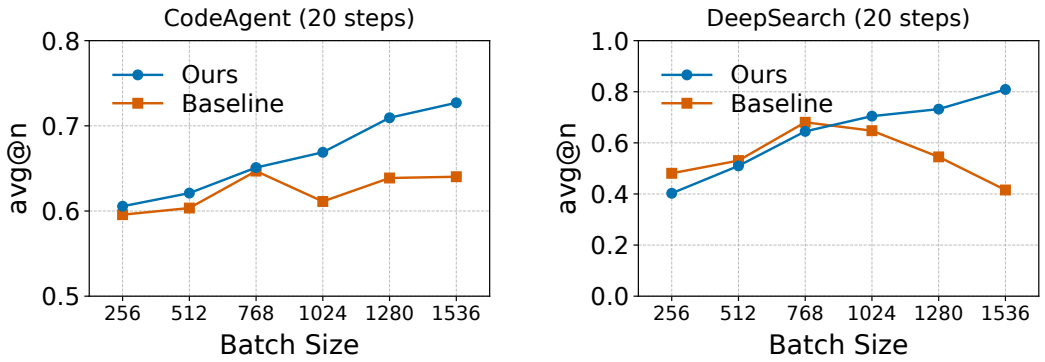
**Baseline deployment.** For AI Coding, Kubernetes creates a pod at the beginning of each trajectory with a 0.5-CPU request and a four-CPU limit. For MOPD, nine teacher models are deployed using SGLang (Zheng et al., 2024), each allocated four GPUs with tensor parallelism (TP). For DeepSearch, each trajectory independently performs API calls, retrying at most three times after errors or a timeout of 600 s. Its reward service uses five replicas with TP degree 8. For MOPD+Search, ten reward services are deployed, each with four GPUs and TP degree 4.

**Scheduling configuration.** We set $depth = 2$ and profile elasticity and execution durations in advance only for CPU reward calculation and GPU reward-model inference.

## E.2 RL CONVERGENCE

Since ALOER does not affect MOPD's inference results, we evaluate RL convergence on the other two workloads, AI Coding and DeepSearch.

**Experimental setup.** We evaluate AI Coding (labeled CodeAgent in the figure) and DeepSearch using the workloads and corresponding baselines described in Appendix E.1. The AI Coding baseline uses Kubernetes-managed per-trajectory environments; the DeepSearch baseline issues API requests independently and uses SGLang for reward-model serving. We vary batch size from 256 to 1536 and run both ALOER and the baseline for 20 RL training steps at each batch size. We report the final step's `avg@n`, defined as the average reward across all $n$ repeats. With the number of steps fixed, larger batches process more training samples and are therefore expected to improve learning, provided that tool execution and reward computation remain reliable.




(a) AI Coding.

(b) DeepSearch.

Figure E.2: Final-step `avg@n` after 20 RL training steps under different batch sizes. Each point averages rewards across all repeats.

**AI Coding.** Figure E.2(a) shows that ALOER already achieves higher reward at small batch sizes. In a small number of environments, test execution during reward calculation takes so long that it exceeds the timeout and is recorded as a failure. By accelerating reward calculation, ALOER allows these tests to finish within the timeout rather than being counted as failures. At larger batch sizes, the gap widens because excessive queuing in the baseline causes many trajectories to become invalid, reducing the effective training data despite the larger nominal batch size.

**DeepSearch.** The two systems achieve comparable rewards at smaller batch sizes (Figure E.2(b)), but the baseline's reward drops sharply as concurrency increases. Without concurrency control, the baseline incurs widespread API request failures, disrupting environment interactions and severely impairing RL convergence. In contrast, ALOER regulates API access through quota and concurrency control, allowing reward to continue improving as batch size increases.

### E.3 Overhead Breakdown

We further analyze the system overhead by breaking down ACTs for AI Coding (CPU-intensive) and MOPD (GPU-intensive), as reported in Table 1. Even when ALOER is heavily congested at an AI Coding batch size of 1536, system overhead accounts for less than 3% of execution duration, compared with only 2% at batch size 1280. For MOPD on GPUs, overhead—primarily due to restoration—represents roughly 25% of execution time, which cannot be overlooked. Fortunately, this overhead does not increase explosively with higher concurrency or more severe queueing, ensuring efficiency and stability under high-concurrency agentic RL training.

| **Workloads** (bsz) | **Coding** (1280) | **Coding** (1536) | **MOPD** (2048) | **MOPD** (3072) |
|---|---|---|---|---|
| Exec Dur. | 0.975 | 1.206 | 0.621 | 0.705 |
| Queue Dur. | 0.015 | 0.428 | 0.081 | 15.05 |
| Sys. Overhead | 0.024 | 0.036 | 0.201 | 0.240 |

Table 1: ACTs Breakdown.