Artos: Buffer-Centric Data Management for Elastic Reinforcement Learning

Yuheng Zhao* HKUST
Suyi Li* HKUST
Tianyuan Wu HKUST
Wei Gao HKUST
Guangzhen Chen WeBank
Jun Yang WeBank
Daohe Lu WeBank
Si Luo WeBank
Chengbo Li WeBank
Minchen Yu CUHK-ShenZhen
Ruichuan Chen Nokia Bell Labs
Wei Wang HKUST

ABSTRACT

Distributed reinforcement learning (RL) has gained significant traction in the field of artificial intelligence. However, existing distributed RL frameworks provide fixed-scaled resource management and have limited expressiveness of different on-policy and off-policy distributed RL algorithms, thus suffering from poor end-to-end performance and low resource utilization.

Therefore, we introduce an abstraction of distributed RL training workflows that enables the flexible and seamless definition of various RL algorithms. Central to this abstraction is the buffer, which acts as a communication channel between distinct stages of RL training. We propose to leverage a buffer-centric approach to design and implement Artos, an elastic and resource-efficient framework for distributed RL training. First, Artos is equipped with a rich set of trigger primitives bound to the buffer, meeting heterogeneous buffer management requirements and supporting workflow patterns of different on/off-policy RL algorithms. Second, Artos exploits the buffer to decouple the stage dependencies and address the resource demand heterogeneity between stages in RL training, which provides resource elasticity at each stage to achieve high resource utilization. Additionally, Artos consists of a high-performance data transfer architecture inside the buffer, supporting efficient data pipelining to alleviate the significant transfer overhead. We prototyped Artos on an AWS EC2 cluster and our evaluation shows that for different RL algorithms, without sacrificing accuracy, Artos either reduces CPU and GPU time by up to 49.89% and 57.41% or reduces end-to-end latency by up to 57.47% compared with well-established open source solutions. Artos further reduces end-to-end latency by 15.22% when batch size changes dynamically, due to its zero-cost scaling ability.

1 INTRODUCTION

Distributed reinforcement learning (RL) has become increasingly popular over the years within the artificial intelligence landscape. RL workflows focus on facilitating agents to learn to make decisions in complex environments, enabling breakthroughs in various applications. Ever since the immense success of AlphaGo [44], which employed RL to surpass human expertise in the intricate game of Go, a proliferation of RL applications has been catalyzed across diverse domains such as resource management [31, 32], robotic manipulation [50, 65], AI gaming agents [47, 51], DBMS [24, 54, 63], and recommendation systems[48, 64, 66], etc.

*Equal contribution

RL training generates experience data by interacting with the environment using a *policy* model, then it stores the experience data in a data buffer and optimizes the *policy* model using the experience data. RL algorithm consists of on-policy algorithms [34, 38, 42, 43] and off-policy algorithms [8, 12, 16, 35]. This classification is generated from the different synchronization patterns of the policy model. A typical distributed RL training workload features an actor-learner architecture, in which multiple distributed actors are allocated on worker nodes with CPU cores to collect experience data and learner process on a head node assigned with GPUs, to update the model with the collected data. Various distributed RL frameworks [10, 11, 17, 23, 37, 40, 58, 59] have been proposed to improve the resource utilization and expedite the RL training speed.

We characterize the distributed RL workloads to facilitate the efficient design of the distributed RL framework in three aspects. (***c1***) Heterogeneous buffer management requirements: The data buffer in RL training stores experience data and model parameters, and manages them with various strategies of different RL algorithms, necessitating the efficient management of both types of data. (***c2***) Resource demand heterogeneity: The actor is a CPU-intensive workload due to the frequent interaction with the environment to collect the experience data. The learner is a GPU-intensive workload to expedite the RL model update. This heterogeneous resource demand benefits from elasticity. Particularly, the learner can benefit from a large batch size [13, 33, 57] and adaptive batch size[15, 30, 45, 58, 59] (***c3***) Substantial data transfer overhead: distributed RL workloads need to frequently exchange the experience data and model parameters between the learner node and worker nodes, accounting for up to more than 49.97% in the end-to-end training time.

Existing distributed RL frameworks do not fully consider these characteristics, leading to unsatisfactory training performance and resource utilization. First, many frameworks [10, 11, 17, 23, 37, 40] opt for fixing resource allocation for distributed RL training workloads. The actors are long-running CPU processes that generate experience data for the learner. Meanwhile, the learner remains idle to wait for the data, leading to underutilized GPU resources, and vice versa, CPU resources also suffer from idleness(***c2***). Second, a few frameworks [58, 59] support resource elasticity for distributed RL training. Despite elasticity, these frameworks employ inefficient communication channels [1, 3] and lack optimizations to mitigate the significant transfer overhead(***c3***). While resource elasticity reduces computational overhead, it exposes data transfer as a performance bottleneck, leading to increased end-to-end latency.

Additionally, none of the existing frameworks are able to seamlessly support the heterogeneous buffer managements(***c1***). Frameworks like Ray and CleanRL fall short of efficient training of off-policy RL algorithms. They enforce the synchronization of model parameters and the experience data between the actors and learner in off-policy RL algorithms for implementation simplicity, incurring unnecessary synchronization overhead. Elastic frameworks either dedicate themselves to a specific RL training algorithm [59] or only achieve resource elasticity for actors with specific optimization strategies [58], limiting their adoption of various RL algorithms.

In this paper, in order to overcome the low resource utilization and the high end-to-end latency of existing distributed RL frameworks, we propose a novel abstraction of distributed RL training workflows. We define a distributed RL workflow as three separate stages: data collection, buffer data management, and data consumption, where buffer data management serves as a communication channel between the two stages. We propose a novel *buffer-centric approach* for an efficient framework, which provides a set of trigger primitives to support heterogeneous buffer management strategies, enables elasticity across stages with varying resource demands, and implements a high-performance underlying data transfer architecture. With this approach, developers can orchestrate sophisticated workflow patterns with a simple configuration of the trigger primitives through a unified interface. Elasticity enables the decoupled stages with different resource demands to be scaled independently to reduce resource idleness. The underlying data transfer architecture is transparent to developers and provides minimal data transfer overhead to accelerate training.

We propose Artos, an elastic and resource-efficient distributed RL framework that provides three key designs to enable high resource utilization and low end-to-end training latency.

First, Artos explicitly manages the utilization of buffer data during distributed RL training. The heterogeneous buffer management requirements in distributed RL training indicate the diverse data exchange patterns in distributed RL training and the need to support various distributed RL algorithms, in which existing frameworks fall short. The buffer-centric approach decouples the stages of data collection and consumption, enabling Artos to flexibly manage diverse workflow orchestrations of distributed RL algorithms rather than being confined to a fixed pattern. Artos offers a rich set of trigger primitives that seamlessly support various data consumption patterns. These trigger primitives are bounded to the data buffer and executed based on various data and conditions, e.g., experience data, model-related data, time periods, transfer conditions, etc. With configurable trigger conditions, developers can employ these primitives to construct different workflows of various on/off-policy distributed RL algorithms, including experience data fetching, model parameter synchronization, scaling, and pipeline data transfer without incurring additional synchronization overhead. We illustrate how to utilize these trigger primitives to meet heterogeneous buffer data management requirements in § 4.2.

Second, Artos provides a lightweight elastic architecture to improve resource utilization and decouples the stage dependencies of components with heterogeneous resource requirements. By implementing an elastic-cloud-process-based framework, Artos deploys actors as elastic cloud processes and offers on-demand GPU services

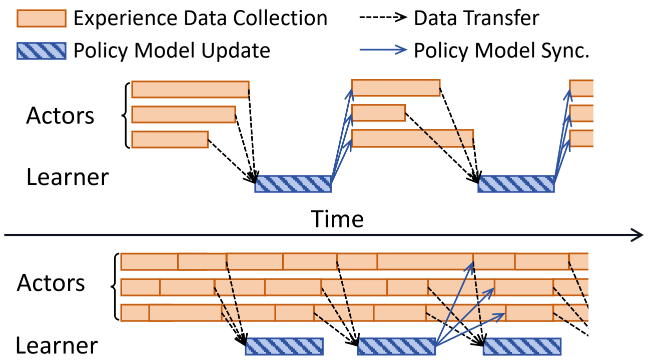


Figure 1: Distributed RL training typically uses an actor-learner architecture, which includes on-policy algorithms (up) and off-policy algorithms (down).

for learners. This configuration allows for efficient resource allocation, facilitating scalability based on real-time demand. Additionally, Artos employs an analytical workload monitor that dynamically collects performance metrics from workloads, thereby enabling pre-warming of actor instances and reducing start-up latency. This lightweight framework not only permits Artos to respond quickly to variations in resource usage but also enables the reclaim of idle resources with minimal overhead. The efficiency improvements observed in § 6.2 indicate a significant reduction in CPU and GPU time, demonstrating the practicality and effectiveness of this approach. In § 6.3 we show that Artos has zero-cost scalability, facilitating optimizations e.g., adaptive batch size training.

Third, Artos implements a high-performance data transfer architecture designed to optimize data transmission within distributed RL training environments. It offers built-in data pipelining strategies, including a mini-batch data pipeline and a granular dual-stage pipeline, which allow the learner to begin updating the policy model prior to receiving all experience data. Additionally, since pipelining has been widely discussed in accelerating ML workloads [9], Artos supports customized data pipelining options to minimize overhead. By abstracting the implementation details of the data transfer layer from developers, Artos relieves the burden of managing complex system implementations. This design enables developers to focus on RL-specific algorithm design without diverting their attention to underlying infrastructure issues. Experiment result in § 6.3 suggests that these optimizations lead to an 11.76% reduction of average latency, further underscoring the efficiency of Artos's architecture.

We implement a prototype framework of Artos from scratch and evaluate Artos against a well-established open-source distributed RL framework, Ray, on popular RL environments with different types of RL algorithms. Our evaluation results show that for different RL algorithms, Artos can either reduce CPU and GPU time by 49.89% and 57.41% without compromising training latency or reduce end-to-end latency by up to 57.47% without provisioning additional resources. Artos shows zero-cost scaling ability and would further reduce end-to-end latency by 15.22% with adaptive batch size training requiring elasticity. Artos also provides a unified programming interface, and developers can easily support their specific algorithms with minimum coding efforts.

2 BACKGROUND AND MOTIVATION

In this section, we give a primer on distributed RL training. We then characterize specific features of distributed RL workflows and illustrate the challenges of an efficient distributed RL framework. Last, we validate the design of the existing RL frameworks and highlight why they fall short in offering high resource utilization and optimized end-to-end latency.

2.1 RL and Distributed RL Training

Reinforcement Learning (RL) is a machine learning paradigm that trains a *policy* neural network to interact with an environment by making actions based on environment states. The environment provides a *reward* for each *action*, and the goal of RL is to learn a *policy* that maximizes the cumulative reward over time. RL has been widely applied in various domains, such as games [36, 44], datacenter management [26, 31, 32, 49], and robotic manipulation [50, 65].

In contrast to the classical machine learning paradigm, where models are trained on pre-existing, fixed datasets, RL follows a trial-and-error approach: It generates experience data by interacting with the environment using a *policy* and optimizes the *policy* based on the experience data. The RL training workflow typically involves two key components: the actor and the learner. During the training process, an actor interacts with an environment, generating trajectories of experience data tuples, i.e., (*state*, *action*, *reward*, *next state*). These tuples capture the current environment *state*, the *action* taken by the *policy*, the resulting *reward*, and the subsequent state transition within the environment. The learner processes these experience tuples collected by the actor to optimize the *policy* neural network. This training process continues until the policy achieves a satisfactory performance or meets a predefined termination criterion.

Distributed RL Training. To improve training efficiency and performance, RL training can be scaled using distributed computing infrastructure, where multiple actors run in parallel to generate experience data, significantly accelerating the data collection process. The collected experience data from multiple actors is stored in a centralized data buffer and the learner aggregates the experience data with different strategies to update the policy model.

Figure 1 illustrates the distributed RL training workflow with a single learner and multiple actors. The workflow pattern between the learner and actors depends on how the learner utilizes the actors' experience data and synchronizes the model parameters. RL algorithms can be divided into on-policy algorithms and off-policy algorithms. For *on-policy* RL algorithms [34, 38, 42, 43], the learner must update the policy using experience data generated by actions taken according to the same policy and synchronize the updated model parameters in each iteration. With this tight restriction, policy model update is more stable [14]. For *off-policy* algorithms [8, 12, 16, 35], the learner can optimize the policy model using experience data generated by a stale policy. This flexibility reduces the synchronization overhead by permitting a divergence between the policy used by actors and the one being optimized.

2.2 Characterizations of Distributed RL

We make characterizations of distributed RL and give design requirements to implement an efficient distributed RL framework.

Table 1: Distributed RL algorithms have heterogeneous strategies to manage the data buffer and to pick the experiment data used for policy model updates.

Pattern	Experience Data Used	Algorithms
Full Batch	Entire buffer's data	PPO, TRPO
Uniformly Random	Randomly pick data	SAC, DDPG, DQN
PER	High TD-error data	Ape-X, PER-DQN
N-Step	Contiguous reward sequence	SAC, Ape-X
FIFO	Data from each actor	IMPALA

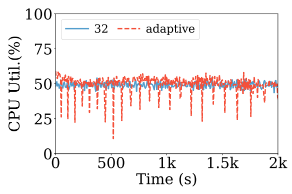
Heterogeneous Buffer Management. A typical distributed RL training workflow involves two types of data: the policy model parameters and the experience data collected by the actors, both of which are stored in the data buffer. Distributed RL training shows a large variance of patterns to utilize the experience data and synchronize the policy model parameters, thus leading to heterogeneous buffer management and control strategies.

On one hand, distributed RL training often requires synchronizing policy model parameters between the learner and actors, and different algorithms vary in synchronization requirements. Figure 1(up) shows that the learner needs to synchronize with the learner during each training iteration in on-policy algorithms. Thus, the actors always use the latest policy model to collect the experience data. Figure 1(down) illustrates that off-policy algorithms allow the learner and actors to work concurrently. Specifically, the learner can update the policy model multiple times and then synchronize the policy model parameters with the actors. Before receiving the synchronized parameters from the learner, the actors use the stale policy model to generate the experience data.

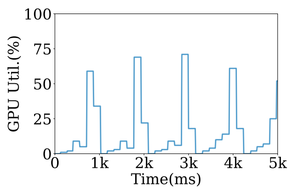
On the other hand, RL algorithms exhibit significant diversity in how they utilize the collected experience data for training, particularly in distributed settings. On-policy algorithms (e.g., PPO [43], TRPO [42]) rely on fresh experience data collected by the current policy and discarding old data after each iteration. Conversely, off-policy algorithms (e.g., DQN [35], SAC [16] and D4PG [8]), leverage replay buffers to store and reuse past experiences, which is called experience replay [29], yet the replay strategies vary widely: DQN employs uniform sampling from a fixed-size buffer, while algorithms like SAC could choose to prioritize entropy-maximizing transitions [19, 41] to focus on high TD-error samples, or N-Step sampling to enhance valuable samples. Table 1 lists the diversified experience data utilization pattern with different distributed RL algorithms. These heterogeneous data management strategies of experience data and policy model parameters constitute the different buffer management strategies of distributed RL algorithms.

Challenge #1: To enable an expressive design pattern, a distributed RL framework should focus on seamlessly supporting the heterogeneous buffer management requirements of distributed RL training workflows.

Elasticity benefits distributed RL Training. When running distributed RL workloads, elasticity is important to increase performance, from two aspects:

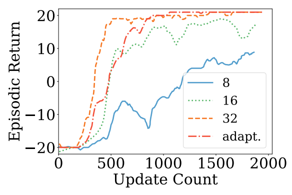


(a) Average CPU Utilization

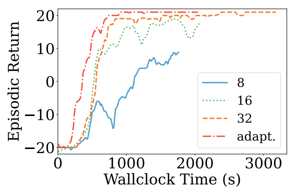


(b) GPU Utilization

Figure 2: Resource utilization of PPO workload on Ray. Blue lines show utilization with 32 actors and red dashed line shows adaptive batch size training. (a) is calculated with the fraction of CPU usage time and total CPU time. (b) shows real-time GPU utilization monitored by `nvitop`.

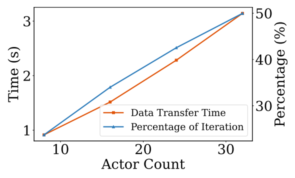


(a)

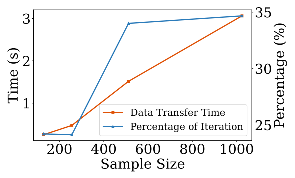


(b)

Figure 3: Profiling results of RL training acceleration approaches. We use OpenAI Atari PongNoFrameskip-v4 workloads as a benchmark. (a) shows the return value of each episode with training updates. (b) shows the return value of each episode with wall-clock time. Each line represents a different actor count with a fixed step size of 128.



(a) Different Actor Count



(b) Different Step Size

Figure 4: Data transfer overhead of PPO on Ray, running Atari PongNoFrameSkip-v4 environment. (a) illustrates that with a fixed step size of 512, data transfer overhead increases linearly with the increase of actor count, and takes up to 49.97% of total iteration time with 32 actors. (b) shows that with a fixed actor count of 16, data transfer overhead increases with the increase of sampling step size per actor, and takes for up to 34.67% with 1024 steps.

Improving Resource Efficiency. Distributed RL training workflows have two different stages, as illustrated in Figure 1, which are experience data collection and policy model update (i.e., data consumption). However, the two stages have heterogeneous resource demands: experience data collection is CPU-intensive and requires CPU for interaction with the environment, while policy model update is GPU-intensive and benefits from the acceleration of GPU. This heterogeneity of resource demand in different stages causes resource under-utilization when switching between the two stages without the ability to scale resources on demand, especially affecting workloads with repeated synchronization and stage dependencies as illustrated in Figure 1(up). Figure 2 shows the resource utilization of training PPO workloads with a fixed step size with Ray. When monitoring the real-time metric with `nvitop`, the GPU utilization pattern in Figure 2(b) shows a correspondence to this repeated synchronization. CPU utilization follows a similar pattern, Figure 2(a) also shows the average CPU utilization within a whole training process. When batch size is fixed with 32 actors, the average CPU utilization is only around 50%. In this case, an elastic framework would enable workflows to release resources after use, thus improving resource efficiency.

Reducing End-to-End Latency. It is widely discovered that in distributed RL dynamically scaling the number of workers would accelerate training. KungFu [30] and Hydrozoa [15] scale up the number of workers in distributed ML training. We empirically validate the effectiveness of adaptive batch size training according to Hydrozoa with large batch training [13, 56]. We adjust the batch size by configuring the number of concurrent actors. Figure 3 shows that adaptive batch size training outperforms other fixed batch size settings. A few research works try to benefit distributed RL training with elasticity: MinionsRL [59] scales the number of distributed RL actors with another RL policy, and Nitro [58] boosts the number of actors in the next iteration when a specific condition is met.

> ***Challenge #2***: Elasticity is important to implement an efficient and effective distributed RL RL framework, lack of elasticity would cause either resource under-utilization or increased end-to-end latency.

Substantial Data Transfer Overhead. Data transfer would introduce additional overhead for distributed RL training, especially for on-policy algorithms, the actors and the learner would need to repeatedly transfer experience data in each iteration. We test the data transfer performance of the PPO algorithm on Ray [37], the state-of-the-art open-source RL framework. The data transfer latency is examined with different batch sizes, by changing either number of actors N or sample step size T of each actor. Experimental results are shown in Figure 4, it is clear that with the increase of actor count, data transfer time and percentage get larger and in our setting with 512 steps per iteration, the overhead can be up to 49.97% of the total iteration latency as illustrated in Figure 4(a). Figure 4(b) shows that with the increase of sample size, the transfer time and percentage also increase, considering that sampling time also grows when T gets larger, the percentage does not increase linearly. The significant overhead motivates us to support various optimizations to reduce this overhead for algorithms, such as data pipelining strategies, to avoid potential performance bottlenecks.

> ***Challenge #3***: An efficient distributed RL framework should carefully manage data transfer to reduce its substantial transfer overhead.

2.3 Limitations of Current Platforms

Based on these challenges, it is clear that a general distributed RL framework should be able to abstract heterogeneous buffer

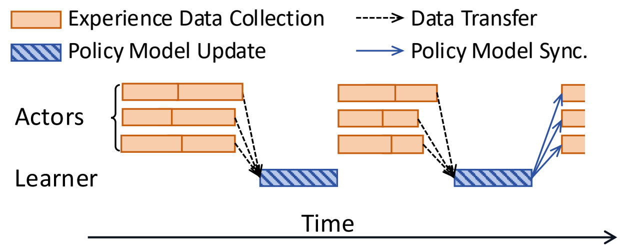
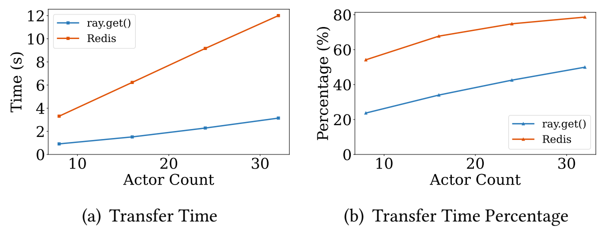


Figure 5: To support a large range of off-policy algorithms, widely used distributed RL frameworks and libraries such as Ray [37] and CleanRL [20] adopt the implementation following the synchronization pattern similar to on-policy algorithms, and introduce additional synchronization point, leading to increased overhead and resource idleness.



(a) Transfer Time (b) Transfer Time Percentage

Figure 6: We run PPO in Atari PongNoFrameSkip-v4 environment and use different data transfer methods to monitor performance. Utilizing Redis for data transfer introduces a large amount of additional overhead compared with `ray.get()`. This method has up to 4× overhead compared with `ray.get()` and becomes bottleneck which takes for up to 78.65% total iteration time with the increase of actor count.

management patterns, provide elasticity to accelerate training while efficiently utilizing resources, and minimize the transfer overhead of experience data. However, none of the existing distributed RL frameworks achieve all these requirements, causing problems of resource under-utilization or increased end-to-end latency.

We start by examining a widely used open-source distributed RL framework Ray [37] which manages resources at a granularity of VMs, and an easy-to-use RL library CleanRL [20]. These works provide a coarse-grained resource management paradigm and long-running processes that would always hold their resources, thus lacking elasticity. This causes resource under-utilization as illustrated in § 2.2. Besides, they do not consider the ability to abstract diverse buffer management strategies, data buffer is treated as a mere infrastructure rather than a key technology to optimize. As a result, their expressiveness is limited and affects performance. For example, none of them implements a pure off-policy algorithm as shown in Figure 1(down), instead, they follow the on-policy workflow in Figure 1(up) and provide implementations with additional synchronization overhead as illustrated in Figure 5. Although this implementation is classified as off-policy, it sacrifices algorithmic specificity due to the lack of generality and introduces synchronization points due to the sequential execution of the workflow. This causes additional synchronization overhead and resource under-utilization akin to that of on-policy algorithms.

Table 2: Comparison of distributed RL frameworks.

Framework	Expressiveness	Data Transfer	Elasticity
Ray [37]	Limited	Fast	×
MinionsRL [59]	Poor	Slow	✓
Nitro [58]	Limited	Slow	✓
ARTOS	High	Fast	✓

There are also research works that managed to accelerate distributed RL training or save resources, which consider elasticity. MinionsRL [59] and Nitro [58] provide elasticity, they are implemented atop AWS Lambda [5] to provide elasticity naturally. However, such a platform requests third-party storage for transferring experience data and model parameters, e.g., a Redis server [3] or AWS S3 [1]. Figure 6 illustrates the significant data transfer overhead with this approach, our experiment data corresponds with previous work [60]. Compared with Ray, our experience shows that using a Redis server for data transfer would introduce up to 382.09% additional transfer overhead, and cause up to 78.65% of the total iteration time. This low performance of data transfer slows down training. Additionally, these frameworks also lack expressiveness because they failed to consider the heterogeneous buffer management requirements, MinionsRL only supports on-policy algorithms and Nitro relies on a single scaling rule for actors.

We compare all these distributed RL frameworks and summarize in Table 2, and there is no general framework that settles all the challenges listed in § 2.2 and provides high expressiveness, fast data transfer, and elasticity. As a result, they either suffer from resource under-utilization or high end-to-end latency. We will further illustrate our design principles and how ARTOS utilizes unique characteristics of distributed RL training to these challenges in §3.

3 DESIGN PRINCIPLE

In this section, we demonstrate the design principles of ARTOS. We give a novel abstraction of distributed RL training workflows and illustrate why our design would settle all the challenges of distributed RL training.

3.1 Novel Abstraction of Distributed RL

Based on our observation in §2.2, we introduce a novel abstraction of distributed RL workflows. We abstract distributed RL training into three stages: data collection, buffer data management, and data consumption. Data collection refers to the process in which the actors interact with the environment for multiple steps and store the experience data in a centralized buffer. Buffer data management is how an algorithm fetches experience data from the buffer, and how to synchronize the updated policy model parameters through the buffer. After fetching experience data, the learner uses them to update the policy model, which is the experience data consumption.

With this abstraction, we further divide distributed RL training workflow into two distinct cycles: data collection and data consumption, as illustrated in Figure 7. The data collection cycle involves experience data sampling, which generates the experience data. Conversely, the data consumption cycle utilizes the fetched experience data to update the policy model, resulting in the production of the policy model parameters and its associated intermediate data.

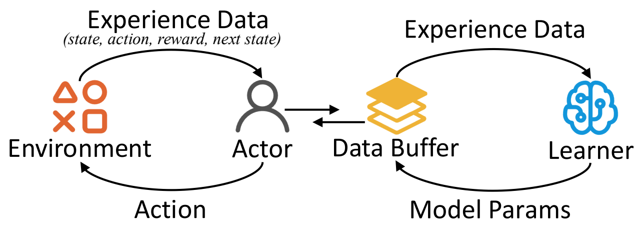


Figure 7: Abstraction of distributed RL training workflows: There are two main cycles, the CPU-bounded data collection cycle (left) and the GPU-bounded data consumption cycle (right). The buffer data management is a communication channel between the two cycles, and for different RL algorithms, the two cycles are coupled with different synchronization and experience data fetching strategies.

Buffer data management serves as the communication channel between the two cycles. Various RL algorithms employ heterogeneous buffer data management strategies (see Table 1), introducing unique synchronization requirements and constraints between the two cycles. In summary, buffer data management is significant in distributed RL training, as each RL algorithm uniquely couples the two cycles through its specific buffer data management strategy.

3.2 Buffer-Centric Design

As discussed in § 2.2, heterogeneous buffer data management strategies of distributed RL training require an RL framework to focus on buffer management, thus improving expressiveness and effectively supporting different algorithms. Our abstraction introduced in § 3.1 also demonstrates the significance of the data buffer in distributed RL training: The difference between diverse distributed RL algorithms lies in how they manage and utilize the experience data in the data buffer and how they synchronize the policy model parameters between the learner and the actors. In other words, heterogeneous buffer management is crucial as it uniquely couples the data collection and consumption cycles, directly influencing how experience data is utilized and how policy model parameters are synchronized during the training workflow, which results in distinct control flows for each distributed RL algorithm.

In § 2.3, we show that current distributed RL frameworks do not apply a specific design for the data buffer. Although data buffers are either logically or physically implemented in their workflows, such platforms treat buffers as a mere component in distributed RL training and fail to design for heterogeneous access [27, 39]. This limits the expressiveness of such frameworks; they either support a specific type of algorithm [37, 59] or support a specific type of optimization [58]. In other words, these practices fail to decouple the data collection cycle and the data consumption cycle and adopt a specific coupling pattern, e.g., as illustrated in Figure 5, Ray adopts a sequential invocation for off-policy workloads and introduces synchronization overhead due to the fixed coupling pattern.

Following this insight, we propose a novel *buffer-centric approach* to orchestrate distributed RL training workload, since buffer data management strategies vary from algorithms. ARTOS makes the consumption of buffer data explicit and enables it to trigger various workflow patterns in different algorithms. This design settles the limitations of current frameworks. ARTOS supports heterogeneous management strategies of experience data and model parameter synchronization. Developers can specify when and how to access buffer data, enabling flexible control over data consumption, and expressing a rich set of workflow patterns of different RL algorithms. This buffer-centric approach decouples the data collection cycle and the data consumption cycle in distributed RL training, with a rich set of trigger primitives. ARTOS eliminates additional synchronization overhead resulting from the limited expressiveness of current frameworks, thus improving performance.

3.3 Lightweight Elasticity

Distributed RL training has several characteristics that motivate ARTOS to adopt an elastic approach, and to provide fine-grained resource management to improve resource efficiency while optimizing end-to-end performance. First, the CPU-bounded data collection cycle and the GPU-bounded data consumption cycle have a mismatch of resource demand, which would result in fragmented idle resources without the ability to scale based on demands. Second, the feature of adaptive batch size training introduces a feature of dynamic resource demands in distributed RL training.

To provide elasticity and fine-grained resource management, ARTOS proposes an elastic cloud process-based architecture. On the CPU side, actor instances are implemented as lightweight elastic cloud processes, by eliminating the state recovery overhead and enabling a fast checkpoint and restore mechanism, ARTOS is able to allocate CPU resources rapidly and reclaim resources after use. On the GPU side, ARTOS implements an on-demand GPU service, by maintaining a GPU pool for learner processes, reducing idle time, and improving GPU utilization with a lightweight software-level approach. It is notable that distributed RL training workloads typically have a small model size and rely on large-size experience data in each iteration, the input data in each iteration is much larger than the size of a policy model. Applying for GPU in each iteration would only introduce negligible overhead for model movement, which is minor compared with the improvement of GPU utilization.

ARTOS tends to further accelerate the start-up latency for actor processes. Recognizing the periodic nature of distributed RL training workloads, ARTOS integrates an analytical workload monitor that continuously evaluates the performance of ongoing tasks. This monitor facilitates a dynamic workload pre-warming strategy, which anticipates the incoming invocation for actor instances and initiates them before they are required. This proactive approach significantly reduces start-up latency, allowing for seamless transitions between data collection and model training phases.

3.4 Minimizing Data Transfer Latency

§ 2.2 demonstrates that data transfer overhead is significant in distributed RL training, to minimize this overhead and improve end-to-end latency, ARTOS implements a high-performance data transfer architecture and provides optimization opportunities for data transfer overhead and end-to-end performance.

To minimize data transfer overhead, ARTOS divides the sampled data into mini-batches and provides fast data access and data pipelines to accelerate data transfer. Firstly, ARTOS adopts a mini-batch data pipeline to optimize data transfer overhead, this pipeline design enables the learner process to start training the policy model

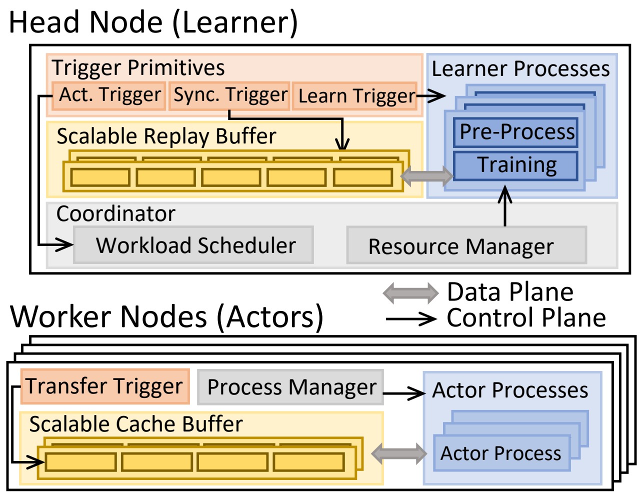


Figure 8: The system architecture overview of Artos.

with only a mini-batch of sampled data and reduce the latency to wait until all data is received. Because in distributed RL training workloads, each update utilizes randomly collected sample data from actor processes, and irregularity exists among actors, this mini-batch data pipeline does not violate randomness and would not affect model accuracy. We further evaluate model accuracy in §6. To fully utilize the characteristics of on-policy algorithms, Artos minimizes data transfer overhead through a granular dual-stage pipeline that effectively overlaps learner-actor data transfers with data pre-processing on the learner node. RL algorithms typically need to perform a data pre-processing at the beginning of each update, which only requires a small portion of data transferred from the actor. This allows for an initial transfer of only the data needed for pre-processing, while the remainder of the data can be transferred concurrently.

4 SYSTEM DESIGN

In this section, we present the design of Artos, an elastic distributed RL framework with a novel buffer-centric design.

4.1 Overview

Figure 8 illustrates Artos's architecture. Artos runs on a cluster of machines, with one GPU server as head node for learner processes and several CPU servers as worker nodes for actors. On the head node, training workloads are registered as learner processes. A workload scheduler is implemented to schedule and dispatch actors for training workloads to available worker nodes, and to monitor the performance metric of each workload. The resource manager manages remote CPU and local GPU resources and coordinates with the workload scheduler to dispatch actors, reclaim CPU resources after use, and maintain the on-demand GPU service for learners. The process manager on worker nodes communicates with the head node. Upon receiving an invocation request, it allocates the request with local CPU resources, starts an actor process, either initializing from scratch or restoring from existing image files, and monitors the life cycles of actor processes. It is also responsible for shutting down or checkpointing an actor process when one finishes sampling, and reclaiming CPU resources. Artos enables developers to register a set of trigger primitives to flexibly manage the control pattern of workloads, e.g., learner triggers, actor triggers, etc. When data arrive in the buffer, Artos checks if any trigger conditions are met, and triggers the next step of a workflow if satisfied. Artos uses ZeroMQ[2] for high-performance data transfer. The experience data is stored in the scalable cache buffer on worker nodes, Artos transfers the cached experience data to the centralized scalable replay buffer following the invocation of the transfer trigger.

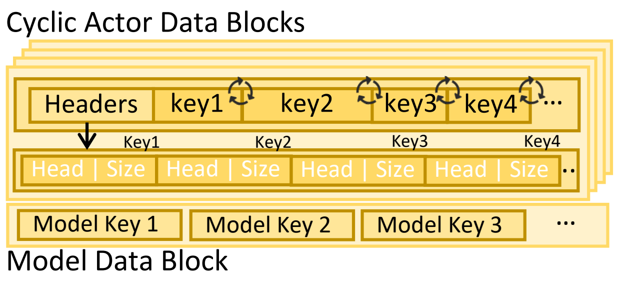


Figure 9: The buffer structure of Artos. The buffer consists of scalable actor data blocks for each actor's experience data and a model data block for model-related data. Artos implements a cyclic buffer structure and maintains a header block of each data key for fast and heterogeneous access.

Table 3: Selected control logic of different distributed RL algorithms and how to trigger via Artos's trigger primitives.

Control Logic	Trigger Primitive	Parameters
Temporal Training	`TimeTrigger()`	`t=time_period`
Full Batch Update	`DataKeyTrigger()`	`n=num_envs,size=step_size`
FIFO Training	`DataKeyTrigger()`	`n=1,size=step_size`
Pipeline Training	`DataKeyTrigger()`	`keys=(keys_1,keys_2,...)`
Actor Invocation	`ObjectKeyTrigger()`	`key=model_param`
On-policy Sync.	`ObjectKeyTrigger()`	`key=model_param`
Off-policy Sync.	`TimeTrigger()`	`t=timestep_interval`
Data Transfer	`TransferTrigger()`	`keys=pipeline keys`

4.2 Data Buffer and Trigger Primitives

Buffer Structure. Figure 9 illustrates Artos's buffer structure. Buffers are implemented with shared memory blocks for fast access across multiple processes. Experience data is stored in actor data blocks, and model data blocks are used to store policy model-related data, e.g., model parameters, gradient noise scale, etc. Artos adopts a scalable structure for actor data blocks, which corresponds with the potential dynamic actor scaling requirements of distributed RL training. Data specifications can be configured by developers and each data key (e.g., *observations*, *actions*, *rewards*, etc.) in the required experience data are allocated to a consecutive shared memory block in the buffer. Artos implements a cyclic buffer structure for each data key and maintains a header block for each data key. By this design, Artos enables fast access to current data and heterogeneous access to different data keys in different RL algorithms.

```
class TriggerWrapper:

    def __init__(self, trigger_id: int,
                 trigger: Trigger, passer: MessagePasser):
        self.trigger = trigger # User configured trigger primitives
        self.trigger_id = trigger_id
        self.passer = passer # Indicate the destination of trigger

    def check(self, *args, **kwargs):
        satisfied, message = self.trigger.check(*args, **kwargs)
        if satisfied:
            '''
            If trigger condition is satisfied, pass the trigger
            invocation message to the corresponding destination
            '''
            message["trigger_id"] = self.trigger_id
            return self.passer.passMessage(message)
        else:
            return None
```

Figure 10: Trigger Wrapper Interface in Artos.

Trigger Primitives. Artos provides a rich set of trigger primitives and monitors the buffer data during training. Developers can configure the buffer with different triggers to specify different control logic and workflow patterns for different distributed RL algorithms. The trigger primitives we implement are listed as follows, and Table 3 illustrates how the trigger primitives support multiple control logic in distributed RL algorithm workflows.

- **`DataKeyTrigger()`**: It triggers the control logic when the associated buffer collects enough size of particular data keys from a specific number of actors (n). It can be used to enable various learner logic, with different parameters.
- **`ObjectKeyTrigger()`**: It triggers the control logic with specific model-related data is ready. This trigger can be used to control model parameter synchronization and actor invocation in on-policy algorithms.
- **`TimeTrigger()`**: It monitors the time period since the last invocation, and can be used to implement temporally sampling experience data from the replay buffer without introducing synchronization overhead, synchronizing model parameters in off-policy algorithms, etc.
- **`TransferTrigger()`**: This trigger is attached to scalable cache buffers on remote workers, supports different transfer conditions, and can easily enable pipelining transfer by indicating multiple data keys to be transferred sequentially.

Figure 10 presents a general interface of `TriggerWrapper` in Artos, which is used to wrap up the trigger primitives and pass the control message flexibly. The `check()` method is invoked every time the data in the buffer changes and passes the parameters to the trigger primitive to check the specified trigger condition. After the trigger condition is satisfied, `MessagePasser` specifies how the trigger should pass its control message to the associated target. Artos supports passing the message directly, through the local queue, and through remote ZeroMQ, to the destinations flexibly.

4.3 Elastic Distributed RL Training

Elastic Cloud Processes. In Artos, actor instances are implemented as elastic cloud processes on worker nodes. Artos adopts CRIU (Checkpoint/Restore In Userspace) [4] to enable rapid checkpoint and restore and to provide elasticity for actor processes. By enabling fast checkpoint and restore, CRIU significantly reduces overhead, making it particularly advantageous for distributed RL actors with frequent and iterative invocations. CRIU captures the complete state of a process, including its memory, CPU registers, and file descriptors, ensuring that upon restoration, the process resumes precisely from its previous context. Artos isolates resources for actors with `numactl`, a command-line utility in Linux that allows users to manage Non-Uniform Memory Access (NUMA) policies for processes and shared memory. It provides Artos the ability to isolate resources for different actor processes, enabling the process manager to bind actors to designated CPUs and memory nodes.

The process manager is in charge of managing the life cycle of actor processes. When receiving a request to start an actor instance from the head node, it checks the availability of worker resources, starts an actor for the first time from scratch, or applies CRIU to rapidly restore an existing actor image and allocate the configured resources to that actor with `numactl`. After an actor finishes sampling, the process manager checkpoints the current context of the actor into image files and stores them on disk, actor processes are shut down and all the related images are recycled when the workload finishes training. To enable scalability for accelerating RL training with adaptive batch size, Artos is able to dynamically scale the number of actor processes by simply registering all the required actors in advance, process manager can be configured to load all the actors on disk without consuming any CPU resources, and scale up by waking them in different iterations when necessary. This approach achieves high scalability with minimal overhead.

On-demand GPU Service. Artos implements a software-level on-demand GPU service model to control the access to a GPU pool. The GPU service model serves requests from learner processes and dynamically allocates GPU resources to a learner process. A training workload only requests for GPU consumption upon the learner process is triggered when the related condition of the learner trigger is satisfied, and releases the allocated GPU after updating the policy model. Artos enables an on-demand GPU service by implementing a time multiplexing strategy of GPU usage.

Analytical Workload Monitor Artos implements an analytical workload monitor within the workload scheduler to dynamically collect the performance metrics of distributed RL training workloads and design pre-warming strategies for actor processes to further mitigate start-up latency. The workload monitor continuously collects the average start-up overhead ($avg\_actor\_start\_time$) and the latency until the learner finishes updating the policy model ($learner\_complete\_time$) for each training workload. The workload monitor adopts a sliding window design to efficiently process and analyze the stream of real-time data, indicated as $actor\_starts$ and $learner\_fins$. The default window size is set to 5. For a training workload, the workload monitor analyzes the stability in real-time when the following equation holds:

$$\Delta_{\text{learner\_fins}} < \lambda \times \mu_{\text{learner\_fins}},$$

Where Δ and μ express the range and average value of the data, while λ is a threshold that indicates the stability condition. Empirically, based on our offline profiling, λ is set to 0.2 by default. This equation refers to how consistent the values in the window are around their average, thus monitoring the stability of the training performance. When the performance of a workload is recognized as stable, the workload monitor would dynamically configure a pre-warm threshold and register a pre-warm trigger for future

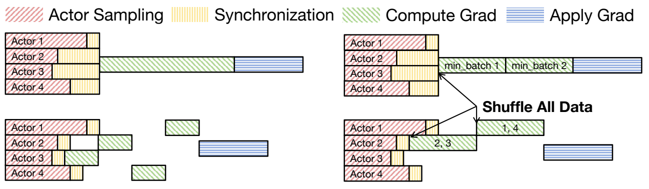


Figure 11: Accumulate gradient and pipelined RL training overview with full-batch training (left) and mini-batch training (right) for on-policy RL workload.

invocations, based on the start-up time and the performance of each iteration in the current window. The pre-warm threshold is indicated as follows:

$$\min(learner\_fins) - \max(actor\_starts) - \Delta_{\text{interval}},$$

where $\Delta_{interval}$ represent the range of time intervals between each value in *actor_starts* and *learner_fins*. We evaluate the accuracy of the workload monitor in §6, and Artos is able to accurately pre-warm actor processes in advance to minimize start-up latency.

4.4 High-performance Data Transfer

With flexible trigger primitives, Artos is able to adopt multiple data pipelining optimizations to minimize data transfer overhead. In this section, we give an optimization strategy for on-policy algorithms, since data transfer overhead for off-policy algorithms is overlapped with policy model update when Artos decouples the data collection cycle and the data consumption cycle.

Mini-batch Data Pipeline Artos manages to update the policy model with partial sampled experience data by adopting the design of accumulated gradient. Conventionally, on-policy algorithms require to update the policy model with full-batch training, as shown in Figure 11 (left). In this case, it is straightforward to calculate the gradient of data from each actor separately and apply the gradient to update the model afterward. However, Figure 11 (right) shows that the state-of-the-practice approach enables mini-batch training to accelerate the training process. Experience data is divided into several mini-batches and the policy model calculates gradient with each mini-batch. This approach requires shuffling the full-batch data and randomly dividing mini-batches for each epoch.

Artos manages data transfer in mini-batches and allows for updating the policy model after the arrival of each mini-batch. Developers can define the transfer trigger on remote scalable cache buffers, to indicate the pipelining transfer of experience data collected by different actors. The learner trigger will monitor the arrived data in the scalable replay buffer, upon receiving enough actors' data (parameter `n` when registering the `DataKeyTrigger()` to form a mini-batch, a learner process is triggered to start calculating the gradient with a partial of data. This approach would reduce synchronization overhead, as the learner process does not need to wait for the sampled data from all the actors.

Granular Dual-stage Pipeline The granular dual-stage pipeline utilizes the pre-processing stage of on-policy algorithms to minimize data transfer overhead. Specifically, for each iteration of distributed RL training with on-policy algorithms, each actor transfers a tuple $(s_{1:t}, a_{1:t}, s_{t+1}, r_{1:t})$ to the learner. $s_{1:t}$, $a_{1:t}$, and $r_{1:t}$ denote the observations, actions, and rewards for the previous t steps, respectively, while s_{t+1} represents the observation following the last action a_t. In practice, $s_{1:t}$ and $a_{1:t}$ constitute a substantial portion of the transferred data; however, the data pre-processing on the learner node for advantage calculation does *not* involve these components. To optimize this process, Artos can flexibly establish a dual-stage transfer pipeline. Initially, the remote cache buffer would only trigger the transfer with specific data keys: s_{t+1} and $r_{1:t}$ to the learner, which then a pre-processing trigger would trigger the data pre-processing for the learner process with a smaller portion of data keys. During this pre-processing, the transfer of remainder data keys (i.e., $s_{1:t}$, $a_{1:t}$) can be triggered simultaneously. Once the data pre-processing is completed, the learner trigger invokes the learner process after waiting for the additional data to arrive.

5 IMPLEMENTATION

We implement a prototype of Artos on an EC2 cluster. Key components of Artos shown in Figure 8 are implemented from scratch with 7.1K lines of Python code. Artos managements the communication and data transfer between the head node and worker nodes with ZeroMQ. On the head node, the coordinator integrates workload scheduler and resource manager instances, listens to specific ports, and waits for process invocation requests from distributed RL training workloads. It schedules actor invocations and manages CPU and GPU resources on demand. The scalable data buffer on the head node and worker nodes are implemented with shared memory for fast data exchange, triggers are bounded with buffers to specify control logics of different distributed RL algorithms. On worker nodes, the process manager waits for invocations and uses CRIU and `numactl` to manage the life cycle of actor processes.

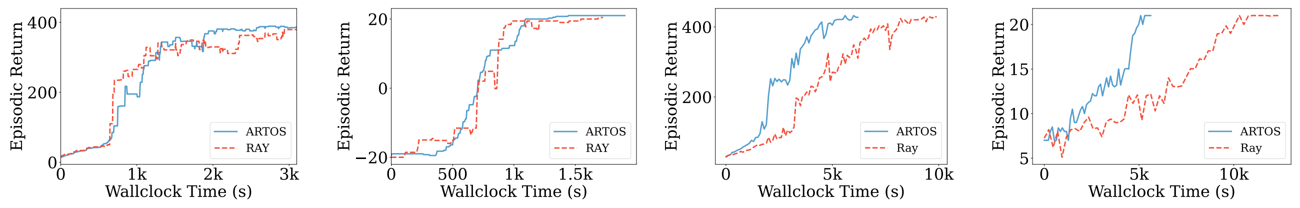
6 EVALUATION

We prototype Artos and evaluate its performance using multiple RL training workloads with different environments and extensive experiments. Evaluation highlights include:

(1) For on-policy algorithms, Artos reduces CPU and GPU time by up to 49.89% and 57.41%, respectively, while keeping end-to-end performance comparable, outperforming state-of-the-art open-source RL framework Ray.
(2) For off-policy algorithms, Artos reduces end-to-end training latency by up to 57.47% compared with Ray.
(3) Artos's data pipeline strategy achieves 11.76% performance increase. The transfer latency is close to fixed-scale implementation with a minor difference of 7.63%.
(4) Artos can monitor the workload performance and enable actor pre-warming to improve performance for 7.71%.
(5) Artos achieves zero-cost scalability. With adaptive batch size training, Artos can further reduce CPU time and GPU time by up to 14.88% and 7.86%, respectively, and reduce end-to-end latency by up to 15.22%.

6.1 Experimental Setup

Cluster Steup. We set up 2 types of AWS EC2 instances to prototype and evaluate Artos. The cluster contains one g5.12xlarge head node and multiple c5.2xlarge worker nodes. The head node hosts workload processes and is responsible for updating the policy model, the head node is equipped with 4 NVIDIA A10G Tensor Core GPU, 48 vCPUs, and 192GB memory. Each worker node is equipped



(a) PPO-Breakout (b) PPO-Pong (c) SAC-Breakout (d) SAC-Pong

Figure 12: Artos has a comparable end-to-end performance with Ray for the on-policy algorithm(a-b) and saves for up to 49.43% training time for the off-policy algorithm(c-d) compared with Ray.

with 8vCPUs and 4GB memory, worker nodes are responsible for hosting actor processes in each update.

Workloads. We evaluate Artos and baseline approach with popular RL environments: BreakoutNoFrameskip-v4, PongNoFrameskip-v4, from Atari gaming environments in OpenAI Gym. The policy model consists of three convolutional layers of 8×8, 4×4, and 3×3 kernel sizes, followed by a flattening layer and a fully connected layer with 512 hidden units. Convolutional layers and fully connected layers use ReLU activation. The observation data collected consists of stacked 4 channels of pixel images with size (84×84).

Baseline and Algorithms. We compare Artos using the following baseline and algorithms:

1) **Ray** [37] is an open-source unified framework for scaling AI and Python applications like machine learning. Ray is widely used in machine learning applications for distributed training, hyperparameter tuning, and model serving, leveraging libraries like Ray Tune for scalable optimization [28], Ray Train for distributed model training, and RLlib for reinforcement learning workloads [27, 53]. Its core primitives enable seamless scaling of Python code from single-node to cluster-level execution, while Ray Serve supports production-grade deployment of ML models.

2) **PPO** [43] is an on-policy algorithm uses Proximal Policy Optimization to iteratively refine stochastic policies through clipped surrogate objectives. It restricts updates via a trust region approach, employs a clip function to limit divergence between old and new policies, and ensures stable training. It discards outdated data, limiting sample efficiency but enabling robustness in complex tasks.

3) **SAC** [16] is an off-policy algorithm based on Soft Actor-Critic, combining policy gradients with Q-learning and entropy regularization. It maximizes expected returns while explicitly balancing exploration via adaptive entropy scaling. SAC employs experience replay to reuse historical data, achieving high sample efficiency.

Procedures. We first train all the selected RL workloads on Artos and baseline framework. We select the same hyper-parameters for each framework and set the random seed exactly the same, to conduct a fair comparison. For PPO, we set 16 remote actors to collect 512 steps of experience data before synchronization. For SAC, we allocate 4 concurrent remote actors to continuously collect experience data, and transfer to the head node every 512 steps, and we set the batch size of each update to 1024. Other hyper-parameters are set as indicated in CleanRL [20]. Each workload is trained for enough updates until the policy model converges. Next, to give an intuitive view of Artos's ability to utilize fragmented resources in each update, we increase the number of workloads by 2 times without provisioning additional CPU resources to the cluster. We also implement a GPU pool with a limited size to serve model update requirements from multiple learner processes.

Table 4: Time-to-accuracy for different tasks.

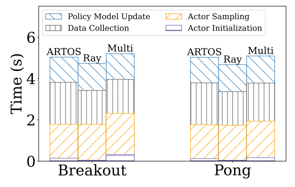
Environment	Algorithm	Baseline	Time (s)
Breakout	PPO	Ray	2438.25
		Artos	**2228.23 (-8.6%)**
	SAC	Ray	8700
		Artos	**5000 (-42.53%)**
Pong	PPO	Ray	1051.67
		Artos	**1140.64 (+8.5%)**
	SAC	Ray	10062
		Artos	**5088 (-49.43%)**

6.2 End-to-End Performance

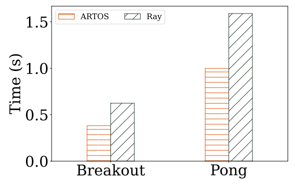
Metrics. We focus on time-to-accuracy and system throughput for training with different workloads. The time-to-accuracy of each training workload is measured by the total wall-clock time for each policy model to converge. We define system throughput as the number of workloads that a framework is able to run given an amount of CPU/GPU resources and with a specific time cost, and it is measured by two metrics. The first is the average update time for each workload. For PPO, it contains a full round of alternation between actor sampling and policy model update, from the invocation of actors in each update to the learner finishes updating the policy model. For SAC, it is a full round for the learner process to fetch and utilize experience data in the replay buffer. The second is the resource time, which is the total time of the CPU and GPU resources held by the training workload until the model converges.

Results. Figure 12 compares the average rewards from the environments with the training wall-clock time of each workload. Figure 12(a), 12(b) show the wall-clock time of on-policy algorithm(PPO). Figure 12(c), 12(d) show the wall-clock time of off-policy algorithm(SAC). In Table 4 we further compare the time-to-accuracy of each workload. Artos has a comparable end-to-end performance with Ray for the on-policy algorithm, the gap between Artos and Ray is negligible, which is only -8.6% ~ +8.5%. Artos reduces end-to-end latency by up to 49.43% for off-policy algorithm.

Figure 13 compares the average update time of different workloads, it also shows the performance overhead breakdown of each update. Figure 13(a) showcases the average update time breakdown for PPO. For a single workload, the average update time corresponds

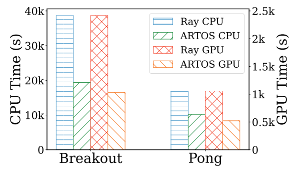


(a) PPO

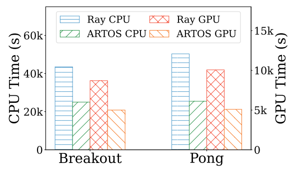


(b) SAC

Figure 13: The average update time of different tasks. Each bar in (a) represents the average update time breakdown for the PPO algorithm with different frameworks and environments, and multi means doubling the workload without increasing resources, which only works on Artos credited to elasticity. Each bar in (b) represents the average update time of the SAC algorithm. Corresponds to Figure 12, Artos significantly reduces update time in SAC compared with Ray.



(a) PPO

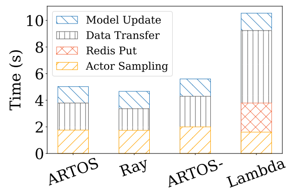


(b) SAC

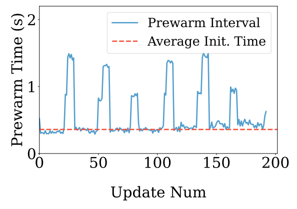
Figure 14: Resource time for training PPO and SAC workloads on Artos and Ray. The left y-axis indicates CPU time and the right y-axis indicates GPU time.

with the time-to-accuracy comparison results in Table 4. Compared with Ray, Artos has a slightly lower data transfer performance, which results in a small decrease in end-to-end latency. We further analyze this data transfer performance in § 6.3. For multiple workload training on Artos, we observe an additional overhead of 3.38% on average for time synchronization between 2 workloads sharing the same piece of resources. However, the average update time of Artos is still close to Ray with negligible overhead of less than 10%. Figure 13(b) showcases the average update time breakdown for PPO. Compared with Ray, Artos achieves an average decrease of 37.99% of average update time, indicating the benefit of expressiveness: Artos can seamlessly support off-policy algorithms without the additional synchronization overhead as in Ray.

Figure 14 presents the resource time for different workloads. We measure resource time for CPU and GPU. Ray holds both resources during the training process. For the on-policy algorithm, Artos consumes only 55.12% of CPU time and 45.97% of GPU time on average compared with Ray. This is due to the elasticity of Artos, CPU and GPU resources can be reclaimed right after use, reducing idle CPU and GPU time, and increasing system throughput. In practice, Artos is able to train a workload count of 2× without increasing CPU resources and only increasing the need for GPU resources to 1.5 times, shows that compared with Ray, Artos reduces the need for CPU and GPU resources to approximately 50% and 75%, respectively, significantly reduce the cost to train on-policy RL workloads. For the off-policy algorithm, Artos consumes 54.02% CPU and GPU time compared with Ray, this is due to the reduced end-to-end latency of Artos, with minimal synchronization overhead.



(a) Pipeline Performance



(b) Workload Monitor

Figure 15: The data pipeline performance in Artos (a), Artos- indicates Artos w/o data pipelining, Artos shows a 11.76% performance improvement on data transfer. The performance of workload monitor in Artos (b), the time interval between pre-warming actors and the actor invoke requests when adaptively changing actor count. Dashed line indicates the average initialization overhead of actor processes.

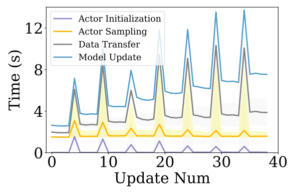
Table 5: Average update time (s) of Artos w/ and w/o pre-warming on different environments.

Environment	w/ pre-warmg	w/o pre-warming
Breakout	**5.14 (-7.55%)**	5.56
Pong	**5.03 (-7.71%)**	5.45

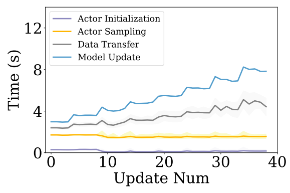
6.3 Microbenchmark

Data Pipeline. Artos adopts a mini-batch pipeline and a granular dual-stage pipeline to achieve efficient data transfer and reduce communication overhead between actors and the learner, we evaluate the data pipeline framework by comparing the data transfer performance of Atari PongNoFrameskip-v4 training PPO with actor count N = 16 and step size T = 512. We introduce Artos- which disables data pipelining on Artos, and Lambda, implementing actors on elastic platform AWS Lambda [5] and transfer data with Redis [3], the same implementation as current elastic RL frameworks [58, 59]. Figure 15(a) compares the average time for actor sampling and data transfer of different approaches. Data transfer time represents the latency to transfer all the sampled data to the learner, and Redis time for Lambda represents the time cost for actors to put the sampled data into Redis storage. Compared with Artos-, which is far from satisfactory with a 20.95% performance gap compared with Ray, the pipelining design of Artos shows an 11.76% performance improvement However, both Artos and Artos- outperform Lambda, which suffers from high latency of interacting with remote Redis storage, with 373.61%, and 297.99% transfer overhead compared with Artos, Artos-, respectively. Artos achieves an elastic design and a data transfer performance close to the upper bound of fixed-scale long-running processes in Ray, without introducing a bottleneck of data transfer as for the commercial elastic platform Lambda.

Workload Monitor and Actor Pre-warming. To highlight the effectiveness of the workload monitor and actor pre-warming strategy in Artos, we train the PPO workloads with Artos and replace the pre-warming mechanism. Table 5 compares the average update time of different environments w/ and w/o actor pre-warming. Although CRIU provides a lightweight process checkpoint and restores for actors, we observe an average initialization overhead of 7.63%. By implementing actor pre-warming strategy, Artos is



(a) Ray



(b) Artos

Figure 16: Time breakdown of each update with adaptive batch size training. Each line refers to a breakdown value, and the shadow of the same color represent the interval.

Table 6: Performance of adaptive batch size training, with the number of actors increasing from 4 to 32. Workloads are trained for 200 and 40 updates, and the frequency of actor count increase is every 25 and 5 updates.

	Update	Baseline	Time (s)
CPU Time	200	Ray	22687.47
		Artos	**12012.05(-47.05%)**
	40	Ray	5148.55
		Artos	**2329.51(-54.75%)**
GPU Time	200	Ray	1094.36
		Artos	**512.49(-53.17%)**
	40	Ray	245.63
		Artos	**101.91(-58.51%)**
Latency	200	Ray	1094.36
		Artos	**1053.15(-3.77%)**
	40	Ray	245.63
		Artos	**208.25 (-15.22%)**

able to mitigate the initialization overhead and reduce end-to-end latency by 7.71%. Figure 15(b) further measures the accuracy of the analytical workload monitor in Artos. The lightweight monitor dynamically adjusts the threshold to pre-warm actor processes even with the adaptively changing actor count. The pre-warm time would undergo a short period of burst when actor count is increased, due to the utilization of historical monitor data within the current window. However, the monitor would rapidly adjust the pre-warm threshold based on the incoming data, and dynamically change the pre-warm time to correspond with the average initialization overhead. In our experiments, Artos is able to predict the invoke request accurately and 92.75% invocations suffer from less than 0.05s waiting for the actor process to be restored. Artos can efficiently monitor the workload performance and provide accurate pre-warming for actor instances to reduce latency.

Elasticity. We evaluate the elasticity of Artos by training PPO algorithm in PongNoFrameskip-v4 environment. The workload is configured with an adaptive batch size by increasing the actor count from 4 to 32, by 4 actors. We run for 200 and 40 updates separately to illustrate different frequencies of batch size adjustment. With a total of 200 updates, the batch size is increased every 25 updates, and with a total of 40 updates, the batch size is adjusted more frequently for every 5 updates. Figure 16 compares the time breakdown of each update in Artos and Ray with 40 updates. With a fixed-scale resource management, Ray has to repeatedly initialize new actors when batch size scales up to avoid the excessive resource over-provision cost. This results in latency spikes in Figure 16(a). However, as shown in Figure 16(b), Artos achieves a smoother latency increase since we enable Artos to register a number of actors and store them to disk in advance, thus avoiding the increased resource time and end-to-end latency due to the costly initialization latency for newly added actors. Table 6 shows the resource time and end-to-end latency of each workload. Artos reduces end-to-end latency and resource time with its scalability. Artos saves 47.05% CPU time and 53.17% GPU time, and has an end-to-end latency of 96.23% compared with Ray with 200 updates. With a more frequent increase of actors, Artos reduces 54.75% CPU time and 58.51% GPU time and has an 84.78% end-to-end latency, indicating the increased performance with more frequent batch size adjustment.

7 DISCUSSION AND RELATED WORK

Security in Artos. Artos isolates resources with `numactl`, in this case, actors or learners on the same node share in-memory data to enable fast access. Commercial platforms do not commonly enable this feature for security issues. With this concern, we claim that, similar to Ray, in Artos, security should be enhanced outside the cluster, and inside Artos, the function code can be trusted. Artos expects to run in a safe environment, e.g., a private cluster or with access control of trusted code. When running distributed RL training workloads or providing applications with Artos, users are responsible for security checks and access control.

Elasticity in Machine Learning. Elasticity has been considered and implemented in machine learning, including model training and model inference. Elastic inference systems aim to implement a high-performance and cost-effective model serving. Research works focus on various domains such as balancing cost-effectiveness and scalability [62], requests batching [6, 7, 55], memory footprint optimization [25, 52, 61] and start-up latency reduction [18, 55]. Elastic training systems are mostly specifically designed for particular training workloads for efficiency [21, 22], such as distributed RL training [58, 59] and large-scale GNN training [46]. These works combine unique characteristics of different training workloads and achieve cost savings or optimizations on resource utilization.

8 CONCLUSION

In this paper, in order to improve resource utilization and reduce end-to-end latency of distributed RL training, we propose a novel *buffer-centric approach* to: 1) Support heterogeneous buffer management requirements of distributed RL algorithms without incurring additional overhead, 2) Decouple different stages with heterogeneous resource demands and provide elasticity, and 3) Provide a high-performance data transfer architecture to mitigate substantial transfer overhead. With this buffer-centric approach, we present Artos, an elastic and resource-efficient framework tailored for distributed RL training, which implements all these desired properties and provides seamless support of different distributed RL algorithms, light-weight elasticity with zero-cost scalability, and high-performance data transfer architecture with minimal additional overhead. Artos outperforms current state-of-the-art distributed RL framework Ray: It either reduces CPU and GPU time for 49.89% and 57.41% or reduces end-to-end latency by up to 57.47% for distributed RL training. Artos is able to utilize fragmented data to provide resource sharing with small private clusters.

REFERENCES

[1] 2006. AWS S3. https://aws.amazon.com/s3/.

[2] 2007. ZeroMQ. https://zeromq.org/.

[3] 2009. Redis. https://redis.io/.

[4] 2012. CRIU: Checkpoint/Restore In Userspace. https://criu.org/.

[5] 2014. AWS Lambda. https://aws.amazon.com/lambda/.

[6] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[7] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2022. Optimizing Inference Serving on Serverless Platforms. *Proc. VLDB Endow.* (2022).

[8] Gabriel Barth-Maron, Matthew W. Hoffman, David Budden, Will Dabney, Dan Horgan, Dhruva TB, Alistair Muldal, Nicolas Heess, and Timothy P. Lillicrap. 2018. Distributed Distributional Deterministic Policy Gradients. In *6th International Conference on Learning Representations (ICLR)*.

[9] Maximilian Böther, Ties Robroek, Viktor Gsteiger, Xianzhe Ma, Pınar Tözün, and Ana Klimovic. 2025. Modyn: Data-Centric Machine Learning Pipeline Orchestration. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[10] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. 2017. OpenAI Baselines.

[11] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. 2016. Benchmarking Deep Reinforcement Learning for Continuous Control. In *Proceedings of the 33nd International Conference on Machine Learning (ICML)*.

[12] Lasse Espeholt, Hubert Soyer, Rémi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. 2018. IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*.

[13] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv:1706.02677* (2017).

[14] Shixiang Gu, Tim Lillicrap, Richard E. Turner, Zoubin Ghahramani, Bernhard Schölkopf, and Sergey Levine. 2017. Interpolated Policy Gradient: Merging On-Policy and Off-Policy Gradient Estimation for Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[15] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. 2022. Hydrozoa: Dynamic Hybrid-Parallel DNN Training on Serverless Containers. In *Proceedings of the Fifth Conference on Machine Learning and Systems (MLSys)*.

[16] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, and Sergey Levine. 2018. Soft Actor-Critic Algorithms and Applications. *arXiv:1812.05905* (2018).

[17] Danijar Hafner, James Davidson, and Vincent Vanhoucke. 2017. TensorFlow Agents: Efficient Batched Reinforcement Learning in TensorFlow. *arXiv:1709.02878* (2017).

[18] Zicong Hong, Jian Lin, Song Guo, Sifu Luo, Wuhui Chen, Roger Wattenhofer, and Yue Yu. 2024. Optimus: Warming Serverless ML Inference via Inter-Function Model Transformation. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*.

[19] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. 2018. Distributed Prioritized Experience Replay. In *6th International Conference on Learning Representations (ICLR)*.

[20] Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and João G.M. Araújo. 2022. CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms. *Journal of Machine Learning Research* (2022).

[21] Jiawei Jiang, Shaoduo Gan, Bo Du, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, Sheng Wang, and Ce Zhang. 2024. A systematic evaluation of machine learning on serverless infrastructure. *VLDB J.* (2024).

[22] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[23] Ilya Kostrikov. 2018. PyTorch Implementations of Reinforcement Learning Algorithms.

[24] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* (2019).

[25] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference (ATC)*.

[26] Suyi Li, Luping Wang, Wei Wang, Yinghao Yu, and Bo Li. 2021. George: Learning to Place Long-Lived Containers in Large Clusters with Operation Constraints. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[27] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *International Conference on Machine Learning (ICML)*.

[28] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. *arXiv:1807.05118* (2018).

[29] Long-Ji Lin. 1993. *Reinforcement Learning for Robots Using Neural Networks.* Ph.D. Dissertation.

[30] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. 2020. KungFu: Making Training in Distributed Machine Learning Adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[31] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets-XV)*.

[32] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*.

[33] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An Empirical Model of Large-Batch Training. *arXiv:1812.06162* (2018).

[34] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of The 33rd International Conference on Machine Learning (ICML)*.

[35] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602* (2013).

[36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* (2015).

[37] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[38] Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, Shane Legg, Volodymyr Mnih, Koray Kavukcuoglu, and David Silver. 2015. Massively Parallel Methods for Deep Reinforcement Learning. *arXiv:1507.04296* (2015).

[39] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. 2021. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research* (2021).

[40] Michael Schaarschmidt, Alexander Kuhnle, Ben Ellis, Kai Fricke, Felix Gessert, and Eiko Yoneki. 2018. LIFT: Reinforcement Learning in Computer Systems by Learning From Demonstrations. *arXiv:1808.07903* (2018).

[41] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized Experience Replay. In *4th International Conference on Learning Representations (ICLR)*.

[42] John Schulman, Sergey Levine, Pieter Abbeel, Michael I. Jordan, and Philipp Moritz. 2015. Trust Region Policy Optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*.

[43] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *arXiv:1707.06347* (2017).

[44] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* (2016).

[45] Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. 2018. Don't Decay the Learning Rate, Increase the Batch Size. In *6th International Conference on Learning Representations (ICLR)*.

[46] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[47] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C. Lawrence Zitnick. 2017. ELF: An Extensive, Lightweight and Flexible Research Platform for Real-time Strategy Games. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[48] Chaoyang Wang, Zhiqiang Guo, Jianjun Li, Guohui Li, and Peng Pan. 2021. A Text-based Deep Reinforcement Learning Framework Using Self-supervised Graph Representation for Interactive Recommendation. *Trans. Data Sci.* (2021).

[49] Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. 2020. Metis: Learning to Schedule Long-Running Applications in Shared Container Clusters at Scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[50] Pengqin Wang, Meixin Zhu, and Shaojie Shen. 2023. Environment Transformer and Policy Optimization for Model-Based Offline Reinforcement Learning. *arXiv:2303.03811* (2023).

[51] Bin Wu. 2019. Hierarchical Macro Strategy Model for MOBA Game AI. In *Thirty-Third Conference on Artificial Intelligence (AAAI)*.

[52] Hao Wu, Yue Yu, Junxiao Deng, Shadi Ibrahim, Song Wu, Hao Fan, Ziyue Cheng, and Hai Jin. 2024. StreamBox: A Lightweight GPU SandBox for Serverless Inference Workflow. In *2024 USENIX Annual Technical Conference (ATC)*.

[53] Zhanghao Wu, Eric Liang, Michael Luo, Sven Mika, Joseph E. Gonzalez, and Ion Stoica. 2021. RLlib Flow: Distributed Reinforcement Learning is a Dataflow Problem. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[54] Zhengtong Yan, Valter Uotila, and Jiaheng Lu. 2023. Join Order Selection with Deep Reinforcement Learning: Fundamentals, Techniques, and Challenges. *Proc. VLDB Endow.* (2023).

[55] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: a native serverless system for low-latency, high-throughput inference. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[56] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2020. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. In *8th International Conference on Learning Representations (ICLR)*.

[57] Yang You, Jing Li, Sashank J. Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. 2020. Large Batch Optimization for Deep Learning: Training BERT in 76 minutes. In *8th International Conference on Learning Representations (ICLR)*.

[58] Hanfei Yu, Jacob Carter, Hao Wang, Devesh Tiwari, Jian Li, and Seung-Jong Park. 2024. Nitro: Boosting Distributed Reinforcement Learning with Serverless Computing. *Proc. VLDB Endow.* (2024).

[59] Hanfei Yu, Jian Li, Yang Hua, Xu Yuan, and Hao Wang. 2024. Cheaper and Faster: Distributed Deep Reinforcement Learning with Serverless Computing. In *Thirty-Eighth Conference on Artificial Intelligence (AAAI)*.

[60] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[61] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. 2021. Gillis: Serving Large Neural Networks in Serverless Functions with Automatic Model Partitioning. In *41st IEEE International Conference on Distributed Computing Systems (ICDCS)*.

[62] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (ATC)*.

[63] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[64] Qihua Zhang, Junning Liu, Yuzhuo Dai, Yiyan Qi, Yifan Yuan, Kunlun Zheng, Fan Huang, and Xianfeng Tan. 2022. Multi-Task Fusion via Reinforcement Learning for Long-Term User Satisfaction in Recommender Systems. In *The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*.

[65] Chao Zhao and Jungwon Seo. 2022. Learn from Interaction: Learning to Pick via Reinforcement Learning in Challenging Clutter. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

[66] Guanjie Zheng, Fuzheng Zhang, Zihan Zheng, Yang Xiang, Nicholas Jing Yuan, Xing Xie, and Zhenhui Li. 2018. DRN: A Deep Reinforcement Learning Framework for News Recommendation. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web (WWW)*.