Multi-Agent Reinforcement Learning with Serverless Computing

Rui Wei
Stevens Institute of Technology
Hoboken, USA
rwei7@stevens.edu

Hanfei Yu
Stevens Institute of Technology
Hoboken, USA
hyu42@stevens.edu

Xikang Song
University of Chicago
Chicago, USA
xikang@uchicago.edu

Jian Li
Stony Brook University
Stony Brook, USA
jian.li.3@stonybrook.edu

Devesh Tiwari
Northeastern University
Boston, USA
d.tiwari@northeastern.edu

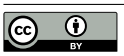
Ying Mao
Fordham University
New York, USA
ymao41@fordham.edu

Hao Wang
Stevens Institute of Technology
Hoboken, USA
hwang9@stevens.edu

Abstract

Multi-agent reinforcement learning (MARL) has emerged as a promising approach for tasks requiring multiple agents for cooperation or competition, such as scientific simulation, multi-robot collaboration, and traffic control. Serverless computing, with its dynamic and flexible resource allocation, has demonstrated potential for improving training efficiency and cost-efficiency in RL workloads. However, existing serverless RL training systems focus primarily on single-agent scenarios, overlooking the unique characteristics and inherent complexities of MARL—such as dynamic inter-agent relationships and heterogeneous policy requirements across agents—leaving inefficient and even infeasible support to diverse and complex MARL algorithms.

This paper introduces **MARLess**, the *first* serverless MARL framework designed to support general MARL algorithms. MARLess decomposes MARL algorithms into serverless functions. It further integrates a dynamic learner sharing mechanism that exploits agent similarities to reduce model update costs and employs actor scaling tailored to MARL tasks, minimizing unnecessary sampling costs based on the data requirements of agents' models. This design optimizes both training efficiency and costs without harming the training quality. Experiments on AWS EC2 testbeds show that MARLess outperforms SOTA MARL baselines with up to 1.27× faster training and 68% cost reduction. Large-scale evaluations on a 15-node cluster with a total of 1,920 vCPUs demonstrate MARLess's scalability and consistent performance under increasing workloads. For a real-world scientific application—turbulent flow simulation, MARLess achieves a 34% cost reduction and 1.1× speedup.



SoCC '25, Online, USA

ACM ISBN 979-8-4007-2276-9/25/11
https://doi.org/10.1145/3772052.3772227

CCS Concepts

• **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Multi-agent reinforcement learning**.

Keywords

Serverless Computing, Cloud Computing, Multi-Agent Reinforcement Learning

ACM Reference Format:
Rui Wei, Hanfei Yu, Xikang Song, Jian Li, Devesh Tiwari, Ying Mao, and Hao Wang. 2025. Multi-Agent Reinforcement Learning with Serverless Computing. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3772052.3772227

1 Introduction

Multi-agent reinforcement learning (MARL) has been widely applied to applications across varying areas, such as turbulent flow simulations [7, 51], biomedical data processing [3, 63, 69], smart city and transportation [35, 70, 88, 89], and robotic control [2, 20, 66]. The high-performance computing (HPC) community has developed reinforcement learning (RL) frameworks tailored for supercomputing infrastructures [34, 91], such as MIT SuperCloud. In parallel, the AI community has rapidly advanced RL toolkits, including MARLlib [28] and RLlib [37]. However, existing frameworks either target single-agent reinforcement learning (SARL) settings which do not generalize to MARL, or rely on serverful deployments that often lead to inefficient resource and time usage.

Multi-Agent RL (MARL) *v.s.* Single-Agent RL (SARL): Unlike SARL, which involves a single agent interacting with the environment,[1] MARL involves multiple agents interacting with the environment and with one another, leading to a significantly larger and more complex joint state-action space. This interaction introduces *non-stationarity* and *dynamic coordination* challenges, but also enables richer solution strategies for complex tasks.

[1] An RL agent is an autonomous decision-maker (*e.g.*, a game player) that learns to take actions to maximize rewards based on feedback from the environment (*e.g.*, a game). The agent develops a policy (or a strategy) by learning from the rewards and feedback.

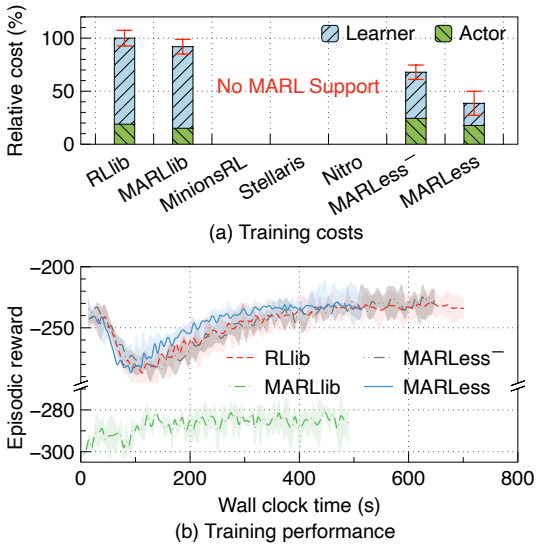


Figure 1: We benchmark the training of the simple-spread task [40] using IPPO [12] for 100 rounds on AWS EC2. Existing serverful MARL frameworks (*e.g.*, RLlib [37] and MARLlib [28]) are (a) costly and (b) inefficient. In contrast, serverless RL frameworks (*e.g.*, MinionsRL [85], Stellaris [86], and Nitro [84]) only support single-agent RL (SARL) and are incompatible with multi-agent settings. As the first serverless framework for MARL, MARLess achieves (a) cheaper and (b) faster training. MARLess$^-$ is MARLess without core capabilities: learner sharing (§4.3) and actor scaling (§4.4).

Fig. 1 shows that existing SARL frameworks [84–86] fall short in supporting efficient MARL training due to their inability to address MARL's unique characteristics: 1) *Agents' dynamic relationship*: In multi-agent environments, agents may exhibit similar behaviors or assume distinct roles [18, 21, 77], creating opportunities for similar agents to share policies [10, 18]; 2) *Varying data needs*: During RL training, the agent needs varying volumes of training data over time [13, 43]. MARL further amplifies this variety due to multiple agents' heterogeneous, varying demands for training data. 3) *Diverse MARL variants*: Owing to MARL's design complexity, many algorithmic variants have emerged—from independent learning methods [12, 76] to centralized training approaches [40, 59, 74, 82]. This diversity poses significant challenges for RL training systems in maintaining broad compatibility and support.

Why Serverless? Serverful deployments using virtual machines (VMs) or physical servers struggle to meet MARL's volatile and dynamic computational demands, due to their coarse-grained resource allocation and slow scaling processes. Recent studies [23, 84–86] have demonstrated the feasibility of training SARL in serverless environments. Building on these findings, we argue that serverless computing is well-suited to MARL, as it can better accommodate its dynamic training patterns and fluctuating resource needs. Fig. 1 shows that serverful solutions [28, 37] take longer time and higher monetary costs than serverless solutions to similar or even less rewards. Moreover, serverless MARL training facilitates portable deployment across cloud and supercomputing infrastructures. However, despite these advantages, serverless adoption for MARL remains non-trivial due to MARL algorithms' inherence complexity. Effectively leveraging the flexibility, auto-scaling, and high concurrency of serverless platforms requires decomposing complex MARL algorithms into independent, parallelizable computation units, executed by serverless functions. Unlike SARL, the presence of multiple interacting agents in MARL significantly complicates the computation logic, posing fundamental challenges for the co-design of MARL algorithms and serverless runtime systems.

This paper presents MARLess, the *first* serverless MARL training framework that integrates system and algorithm co-design to enable *faster and more cost-efficient* training across a broad range of MARL algorithm variants. Specifically, MARLess develops a MARL-algorithm compatible design that decouples and interprets MARL algorithm variants into fundamental primitives (*e.g.*, actor, learner, policy model, critic model *etc.*),[2] executed by carefully-orchestrated serverless functions (§4). MARLess dynamically shares learners with higher-performing policies among behaviorally similar agents, enabling them to reuse the same policy. This approach reduces the number of required learners and saves computational resources.

Additionally, MARLess evaluates agents' varying demands for training data (*i.e.*, feedback from the environment) and auto-scales to a just-right number of actors, further optimizing training efficiency and cost.

Our main contributions can be summarized as follows:

- We design MARLess, the first serverless MARL training framework that enables *faster and cheaper* training with a co-design of serverless architecture and MARL algorithms.
- MARLess introduces three core features, *only* enabled by the unique co-design of serverless computing systems and MARL algorithms: *a compatible design for diverse MARL algorithms*, *dynamic learner sharing*, and *cost-aware actor scaling*. Together, these features are key to leveraging the flexibility and expressiveness of serverless functions to address the complexity and dynamic nature of MARL training.
- We implement and evaluate MARLess on an AWS EC2 testbed and a 15-node HPC cluster with 1,920 vCPUs. Experimental results show that MARLess accelerates the training process by up to 1.27× and reduces training costs by up to 68% in representative MARL environments. For a large-scale scientific MARL workload—turbulent flow simulation, MARLess consistently reduces costs by 34% and achieves 1.1× speedup.

2 Background and Motivation

2.1 SARL *v.s.* MARL

Fig. 2 shows the fundamental difference between SARL and MARL. SARL only employs one agent to interact with the environment by learning a policy $\pi(a \mid s)$, which indicates taking action a given the state s. The objective is to maximize the expected cumulative reward, defined as $\mathbb{E}_\pi \left[\sum_{t=0}^{T} \gamma^t r^t\right]$ [48], where r_t is the reward

[2]Each actor interacts with an environment that involves all agents, allowing it to collect rewards and feedback from their interactions. Learners then train policy with or without critic models with the rewards and feedback (no critic models for some MARL variants).

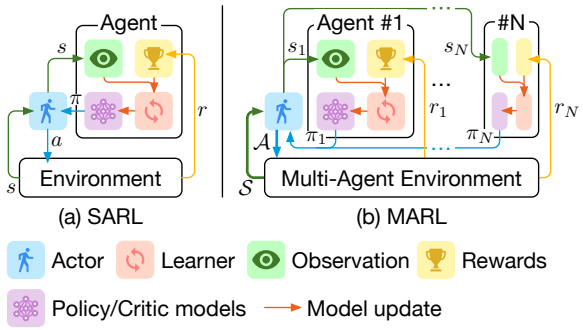


Figure 2: SARL *v.s.* MARL.

received at time step t, and $\gamma \in [0, 1)$ is the discount factor that determines the importance of future rewards.

Unlike SARL, MARL involves multiple agents interacting with the environment and each other, fostering a stochastic game characterized by joint state and action spaces, $\mathcal{S}$ and $\mathcal{A} = \prod_{i=1}^{N} \mathcal{A}_i$, as well as shared or individual rewards r_i^t. Agents learn their own policies $\pi_i(a_i|s_i)$ for $i \in \{1, \ldots, N\}$, with individual partial observation s_i [8]. Thus, due to MARL's inherence learning dynamics and complexities arising from the heterogeneity, interactions among multiple agents, and environmental volatiles, existing SARL frameworks [84–86] can hardly support efficient MARL training.

SARL and MARL algorithms can be implemented in a distributed fashion using the *actor-learner architecture* [15, 16, 44, 49], which decouples data collection (by actors) from policy updates (by learners), enabling scalable and parallelized training. In this architecture, multiple distributed *actors* interact independently with their environments—executing actions a and collecting observations s. Once enough samples are collected, they are sent to the *learner*, which computes gradients and updates the policy models π.

2.2 Motivating Dynamic Learner Sharing

Competition, cooperation, and combinations of both emerge among agents in MARL. Agents with similar objectives may exhibit similar behaviors [18, 21, 77], enabling learner sharing to simplify training and improve generalization by allowing these agents to share a common policy model [10, 18, 21, 29, 77]. From a system perspective, reducing the number of learners also lowers computational costs.

However, naïve learner sharing may result in reduced rewards compared to training agents with separate individual policy model. Thus, we demonstrate the necessity of a reward-aware dynamic sharing method by applying indexed learner sharing [18], which feeds the agent index as an extra input to the shared policy, in two MARL tasks from Multi-Agent Particle Environment (MPE) [40]: *simple-spread* and *simple-adversary*, both with three agents. The *simple-spread* task represents a fully cooperative scenario, and the *simple-adversary* involves both competition and cooperation.

Fig. 3(a) and (b) show the mean episodic rewards across the training process.[3] Due to the varying relationship between agents, static learner sharing strategies can lead to lower rewards compared to the no-sharing setting, as Fig. 3(a) shows. When running the *simple-adversary* task with learner sharing disabled, we measured

[3]One training round is defined as a single update of all agents' policies. One round may involve multiple episodes. Each episode indicates a game from the start to finish.

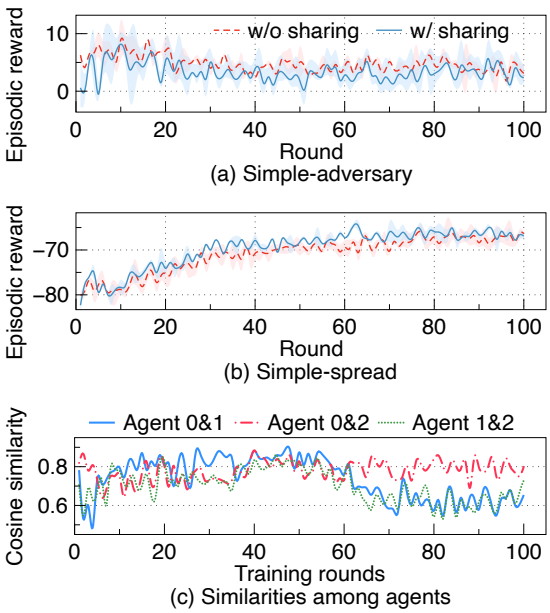


Figure 3: Average episodic rewards per round in two MPE scenarios with and without learner sharing, respectively. We run 100 training rounds for each setting: (a) *simple-spread* and (b) *simple-adversary*. (c) shows dynamic relationships among agents change during the training.

the similarity of action distributions dictated by agents' policies to show their relationship. Fig. 3(c) shows that the similarity among agents' policies changes over time. Existing static learner sharing strategies [10, 18] fail to adapt to the dynamic nature of agent behaviors. Thus, we argue that it is essential to enforce dynamic learner sharing based on agents' runtime behaviors.

2.3 Motivating Actor Scaling

As training progresses in deep reinforcement learning (DRL), policy models typically require varying amounts of data for each round to reach the optimal performance [13, 43]. This variability directly impacts the scale of the round and is influenced by the number of active actors. Dynamically adapting the number of actors ensures generating sufficient data while minimizing resource waste.

Existing actor-scalable distributed RL frameworks [85, 86] focus on SARL, where only one policy model is considered. They estimate its data needs and adjust the number of actors accordingly. While effective in single-agent settings, these methods are not directly applicable to MARL. MinionsRL [85], for example, uses a RL-driven scheduler that predicts actor demand based on rewards but requires hours of pre-training. In contrast, Nitro [84] adopts a data-driven approach by analyzing the Hessian matrix and outperforms MinionsRL in both cost and efficiency.

MARL introduces higher complexity with multiple agents and policy models, each requiring different amounts of data that vary over time. Since all actors generate data for all agents, mismatches in model demands can cause inefficiencies—for instance, one model may require much more data than others. This complexity makes it non-trivial to transfer SARL scaling methods to MARL. It calls for

MARL-specific strategies that balance heterogeneous data demands across policy models and scale actors efficiently to reduce costs without harming training quality.

2.4 Motivating MARL with Serverless

The emergence of serverless computing has created new opportunities to enable dynamic and scalable resource allocation, which releases idle resources in time and enhances overall efficiency. As shown in Fig. 2, training in RL tasks is typically synchronous, which often leads to idle time between actors and learners. Serverless computing can reduce training costs by deallocating resources during these idle periods, as illustrated by MARLess$^-$ in Fig. 1. However, simply applying serverless infrastructure by turning MARL components into stateless functions is not sufficient to fully optimize training efficiency in MARL workloads. To achieve this, additional system-algorithm co-design optimizations, such as learner sharing and actor scaling, are required.

At the same time, serverless computing is inherently well-suited for MARL due to its ability to handle dynamic and fine-grained resource demands. In distributed RL systems, each actor or learner typically requires a small resource footprint (*e.g.*, one CPU) and must be instantiated quickly (*e.g.*, within one second), making it difficult for traditional serverful platforms to adapt efficiently and reduce training costs under such dynamic conditions. In contrast, serverless environments support rapid provisioning and fine-grained resource control, enabling MARLess to reduce training costs by dynamically releasing similar policy models through learner sharing and avoiding the launch of redundant actors via actor scaling.

Recent serverless-based SARL frameworks [84–86] have demonstrated significant improvements in training- and cost-efficiency. However, these serverless SARL training frameworks are not compatible with MARL workloads and are difficult to adapt, as their core designs focus solely on training a single policy model. This leads to two key limitations: 1) they overlook the interactions and dependencies among multiple policy models, and 2) they cannot balance the varying data needs across models, which is essential for maintaining overall training quality.

3 Objectives and Challenges

Prior research has addressed several challenges of serverless computing. For example, cold-start latency [9, 68, 83] and state management [41, 61], have been extensively studied. However, executing MARL workloads in a serverless environment introduces new challenges that remain largely unexplored. Based on the latest related work and the observations discussed in the previous section, we propose the design of a serverless MARL system with the aim of achieving three primary goals listed below:

Interpretability: MARLess aims to be a generic serverless MARL framework that supports diverse MARL algorithms. It should allow users to interpret MARL algorithms to serverless functions with minimal refactoring, thereby fully leveraging the flexibility and auto-scaling capabilities of serverless computing.

Efficiency: Our primary goal is to leverage the benefits of serverless computing and the similarity among agents' policy models to improve training- and cost-efficiency. In MARLess, we design novel *multi-agent actor scaling* and *dynamic learner sharing* methods combined with serverless computing to achieve this.

Scalability: MARLess is designed to fully utilize available resources, scaling up or down as needed. Similar to existing serverless DRL frameworks [85, 86], MARLess includes scalable actors that can be dynamically adjusted during training rounds.

Running MARL training in a serverless environment can lower costs by dynamically releasing idle components. However, to optimize training efficiency and cost reduction, we aim to fully exploit serverless flexibility through actor scaling and learner sharing. To achieve the three objectives above, the following new challenges must be addressed:

How to decompose diverse, complex MARL algorithms into serverless functions? Most frameworks [28, 37, 64, 90] use centralized learners to simplify logic, but this design limits scalability and efficiency in serverless settings. MARLess decomposes learners into specialized functions, enabling fine-grained resource control while supporting a wide range of MARL algorithms.

How to design dynamic learner sharing without affecting training quality? Learner sharing, also known as parameter sharing, is widely used in MARL to streamline training [12, 21, 26, 33, 40, 82]. We extend this concept to dynamically launch and release learners to reduce training costs. However, most methods rely on static groupings [1, 10, 21, 59, 72], which can hurt final performance as agent behaviors evolve [29, 87]. To maintain training quality, we require a lightweight, adaptive method that responds to performance changes without adding significant overhead.

How to balance the varying data needs among agents? Existing scaling strategies [84, 85] are tailored to SARL and cannot handle MARL's heterogeneous data demands. MARLess introduces a new metric to dynamically scale actors and ensure most policy models receive sufficient training data, balancing quality and cost.

4 MARLess's Design

4.1 Overview

MARLess is a serverless, distributed MARL training framework that leverages serverless computing and dynamic learner sharing to improve cost-efficiency. We first analyze a wide range of MARL algorithms and identify decomposable and parallelizable computation motifs to interpret MARL algorithms into serverless functions. In addition, MARLess's core components include a performance decrease detector, a learner categorizer, and an actor scaler. After decomposing MARL algorithms into concurrent actor functions and learner functions (§4.2), the training process is divided into four main steps, as illustrated in Fig. 4.

Step (1): Sampling and training. Actors retrieve the latest model weights from the external cache and continuously sample trajectories using the current policies. Each actor is configured with its own copy of the policies and a separate MARL environment for interaction. The generated trajectories are then stored in the external cache. Each trajectory records the previous observation, action, current observation, and reward for a single agent. Once enough data is collected, learner functions calculate the loss values and update their corresponding policy models. This process repeats until training is complete.

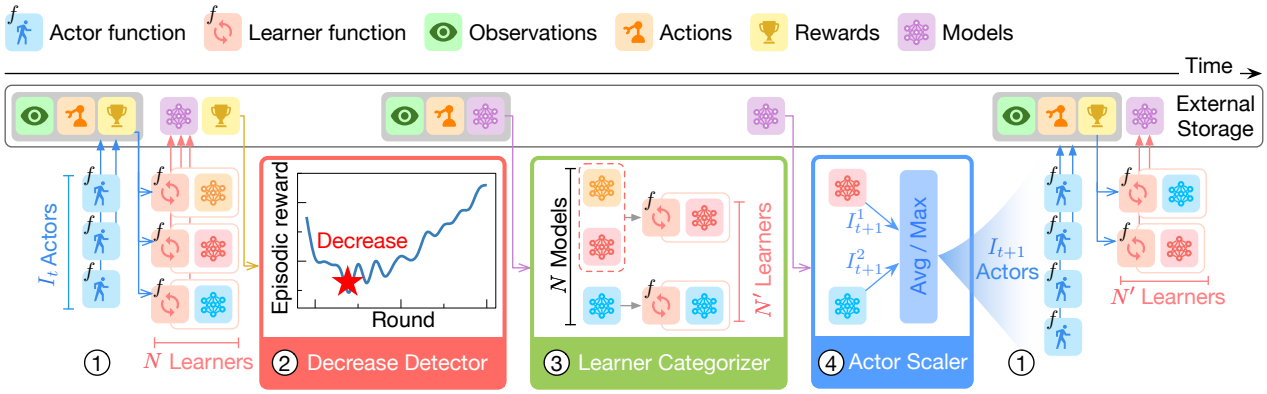


Figure 4: MARLess's workflow.

Step ②: Performance decrease detection. MARLess computes a global reward R'_t by aggregating the agents' rewards from round t. This global reward is normalized to facilitate performance drop detection. If a performance decline is detected, a new policy sharing strategy is triggered. The training process alternates between enabling and disabling learner sharing to help agents recover from the performance drop more effectively.

Step ③: Dynamic learner sharing. If learner sharing is enabled after a performance drop is detected, the learner categorizer retrieves the latest trajectories and analyzes their similarity across agents. The similar agents will be grouped together for learner sharing. Agents within the same group share a common learner and policy model, resulting in a new policy set $\pi_{N'}$ with N' learners.

Step ④: Actor scaling. The actor scaler estimates the required number of actors per policy using the Hessian matrix eigenvalues and applies a scaling rule to compute the total number I_{t+1} for the next round. Training then proceeds to round t with I_{t+1} actors and N' learners, repeating from Step ① until training ends.

4.2 Decomposing Diverse MARL Algorithms

In existing MARL frameworks [28, 37, 64, 90], learners are typically implemented with coarse granularity, managing all agents within a centralized learner. In contrast, MARLess decomposes the centralized learner into separate functions, enabling fine-grained control in a serverless environment. This design improves flexibility and allows the system to manage each function independently, adapting more efficiently to the dynamic learner sharing design. The function decomposition follows the core execution logic of mainstream MARL algorithms, enhancing serverless functions' interpretability and compatibility. In MARLess, serverless learner functions are categorized into three types based on their algorithmic roles: critic functions, policy functions, and post-processing functions.

The *policy function* trains the agent's policy network, which generates actions based on the agent's observations. In actor-critic algorithms like Proximal Policy Optimization (PPO) [67] and Deep Deterministic Policy Gradient (DDPG) [38], the policy network serves as the actor network.[4] For value-based methods like Independent Q-Learning (IQL) [48], the policy network is instead the value network. The *critic function* is specific to actor-critic methods. It updates the critic network, which estimates returns and assists in training the policy network [75]. The *post-processing function* is used in centralized training methods with information sharing like Value-Decomposition Networks (VDN) [74], Multi-Agent Proximal Policy Optimization (MAPPO) [82] and Multi-Agent Deep Deterministic Policy Gradient (MADDPG) [40], where models are trained using global information. This function processes the trajectory data produced by actors into the correct format for the critic and policy models. As shown in Fig. 5, by flexibly combining the *policy function*, *critic function*, and *post-processing function*, all common types of MARL algorithms can be constructed. The categorization of these algorithms follows existing work [28, 90].

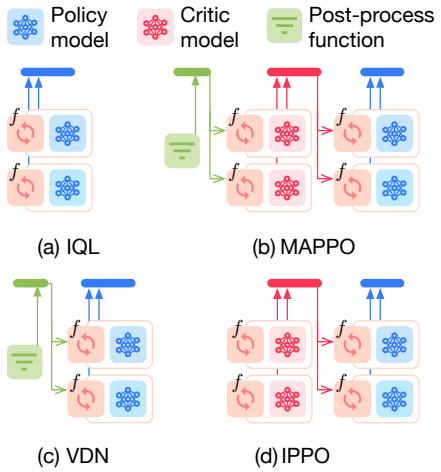


Figure 5: MARLess's learner function composition varies based on different MARL algorithms, including: (a) independent value-based methods [76], (b) centralized actor-critic methods [40, 82], (c) centralized value-based methods [59, 74], and (d) independent actor-critic methods [12, 40].

4.3 Dynamic Learner Sharing

Unlike SARL, where a single policy model is used, MARL typically assigns each agent a separate policy model that maps observations to actions. During training, some policy models may become

[4]The term "actor" here refers to the actor component in the actor-critic architecture, which is distinct from the actor function in the actor-learner training paradigm.

similar—producing identical actions given the same observations—creating opportunities for learner sharing to reduce training overhead [18, 21, 77]. However, static sharing can degrade training quality by limiting agents' adaptability to evolving and diverse roles [29, 87]. To address this, MARLess integrates a dynamic learner sharing mechanism that is aware of training quality and can enable or disable sharing based on performance trends. To enable this capability, four key challenges must be addressed: **1)** How to mitigate the effects of reward fluctuations in training when detecting the performance decrease? **2)** How to calculate the similarities among policies without introducing extra overhead? **3)** How to determine the appropriate number of shared learners? **4)** How to identify which policy models should be shared among similar agents? We address these challenges using four corresponding techniques, as described below.

4.3.1 Performance decrease detection. In MARLess, learner sharing is toggled on and off when the performance of the alternative strategy declines. Specifically, when sharing is enabled and the reward decreases, MARLess disables sharing to allow agents to diverge and explore independently. Conversely, when sharing is disabled and the reward decreases, MARLess enables sharing so that similar agents can reuse the same policy, thereby reducing the number of active learners and overall learner cost. To fairly assess each agent's contribution to the global rewards, the individual reward of agent i at round t is first normalized as $\hat{r_i}^t := \frac{r_i^t - \min(r_i)}{\max(r_i) - \min(r_i)}$ [39], before being aggregated. However, DRL training often exhibits significant reward fluctuations. To mitigate the impact of these fluctuations, MARLess uses a tolerant window of the most recent W rounds of global rewards to monitor performance trends, where W defines the window size. MARLess then fits a least squares regression line and calculates its slope k; it switches strategies only if k falls below a predefined threshold γ and the tolerant window is full. Besides, we observe that each time the strategy changes, there may be a temporary drop in performance, which typically recovers shortly afterward. To allow the system sufficient time to stabilize after a strategy switch, the window is cleared following each change.

4.3.2 Policy similarity calculation. Once a performance decrease is detected and learner sharing is enabled, MARLess groups similar agents by comparing their policy models, allowing shared models. A common approach measures similarity by vectorizing each agent's trajectories—observations and actions—and computing distances between these vectors. state-of-the-art (SOTA) methods like selective parameter sharing [10] and Multi-Agent Policy Distance (MAPD) [29] train encoder models to generate latent representations [10, 29]. However, these introduce extra training costs and lack generality across diverse MARL tasks. Instead, MARLess compares the probability distributions of each agent's policy model. During updates, learner functions construct a Kernel Density Estimation (KDE) model for each agent using the latest trajectories. These models represent observation-action distributions. For agent i's policy π_i, MARLess computes the Kullback–Leibler (KL) divergence between every agent pair. To reduce computation in high-dimensional spaces, MARLess compresses each trajectory using principal component analysis (PCA).

4.3.3 Agent categorization. After computing agent similarities, the categorization step becomes a clustering problem. The next challenge is to determine how many groups to form. As previously discussed, changing the sharing strategy often causes a temporary drop in reward. The size of this drop is closely related to the change in the number of policy models—the larger the change, the greater the drop. Although models typically recover, a large disruption late in training can negatively impact final performance. To mitigate this, we introduce a policy count scheduler in the dynamic learner sharing module. It limits how much the number of policy models can vary over time, helping maintain training stability and preserve final rewards. When learner sharing is enabled for the jth time, the number of policy models N' is given by $N' = \min(2^j, 2^{\lfloor \log_2 N \rfloor + 1})$, where N is the total number of agents. This scheduler prevents drastic changes in strategy, especially during the later stages of training. Finally, MARLess uses K-Means—a widely used clustering algorithm—to group similar agents for subsequent learner sharing.

4.3.4 Shared model selection. When a group of agents is clustered together, MARLess selects one agent's model as the base for sharing, following a strategy similar to existing work [29]. The selection is based on evaluating the performance of each agent's policy model. The Hessian matrix—a common tool for analyzing the curvature of neural network loss surfaces [30, 55, 73]—has been applied in scalable RL training [84]. Following prior work, we compute the convexity ratio C of a policy π_j using the largest and smallest eigenvalues of its Hessian matrix:

$$C^j := -\frac{e^j_{max}}{e^j_{min}}, \tag{1}$$

where e^j_{max} and e^j_{min} are the largest and smallest eigenvalues, respectively. A higher convexity ratio C^j suggests the model is closer to a local optimum on the reward surface, while a lower C^j indicates flatter or less promising reward regions [84]. The model with the highest C is selected as the base for sharing. Once the new policy set is finalized, the corresponding learner functions are configured and remain active until the next strategy switch. By adjusting the number of policies, MARLess dynamically scales the number of learner functions to optimize learner costs.

4.4 Actor Scaling

Previous research has shown that data requirements vary across training stages in deep neural networks [13, 43], including RL models [84, 85]. In such cases, the number of trajectories needed by agents' policy models changes over time. Unlike actor scaling in MinionsRL [85] and Nitro [84], MARLess scales actors based on each policy model's performance, taking into account the varying data needs of different models. We begin by defining how to compute the required number of actors for a single policy model, following the methodology proposed in prior work [84].

The number of actors I_t at round t, bounded by an upper limit I_{max} and a lower limit I_{min}, is determined based on the estimated training improvement, as shown in Eq. (1). As mentioned, a smaller convexity ratio C^j for policy π_j indicates poorer exploration due to less diverse trajectories. This suggests that generating more trajectories can lead to greater benefits for π_j, requiring additional

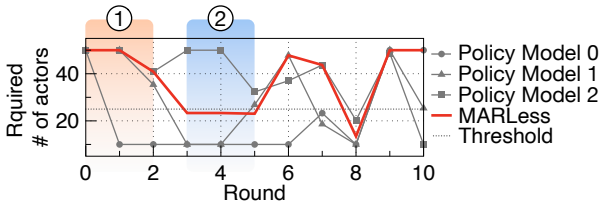


Figure 6: MARLess scales the number of actors to balance training quality with cost efficiency.

actors. Thus, more actors are launched when policies have lower convexity ratios. To account for training fluctuations, MARLess reuses the tolerant window to record convexity ratios of each policy over the last W rounds, represented as $C^j_W := \{C^j_{t-W+1}, \ldots, C^j_t\}$. Using this record, the scaling ratio S^j_t for π_j at round t is calculated with min-max normalization as

$$S^j_t := \frac{C^j_{max} - C^j_t}{C^j_{max} - C^j_{min}},$$

where C^j_{max} and C^j_{min} are the maximum and minimum convexity ratios of policy π_j within the window. The number of actors for a policy π_j can be calculated as

$$I^j_t := Clip(S^j_t \times I_{max}, I_{min}, I_{max}),$$

where the *Clip* function is used to ensure that I^j_t stays within the defined upper and lower bounds. This design addresses the skewed scaling scores with many near-zero values. A direct linear map is too sensitive, as small noise can trigger unnecessary scaling. Clipping adds a dead zone to ignore minor fluctuations and activates scaling only when real demand appears, following prior actor-scaling work [84]. After computing the required number of actors for each policy model, we analyze the distribution of data needs across models during training. To illustrate this, we run Independent Proximal Policy Optimization (IPPO) on MPE's three-agent *simple-spread* task and record the evolving actor requirements for each policy model, as shown in Fig. 6. Based on these observations, we design a cost-aware actor-scaling strategy to guide the allocation process. Specifically, the total number of actors to launch in the next round, denoted as I_t, is computed as:

$$I_{\text{threshold}} := \beta \times (I_{\max} - I_{\min}),$$

$$I_{\text{average}} := \frac{\sum_{j=1}^{N'} I^j_t}{|N'|},$$

$$I_t := \begin{cases} \max(I^j_t \mid j \in N'), & \text{if } I_{\text{average}} \geq I_{\text{threshold}}, \\ I_{\text{average}}, & \text{if } I_{\text{average}} < I_{\text{threshold}}, \end{cases} \quad (2)$$

where N' represents the set of current policies, and β is an upper threshold. As shown in Fig. 6, the threshold β helps balance ① **training quality** by allocating enough actors when most policy models require more data, and ② **cost reduction** by launching only a moderate number of actors when increased demand is limited to the minority of models. As long as $\beta < 1/K$, MARLess avoids the extreme cases where a single outlier causes unnecessary scaling, which can be theoretically verified.

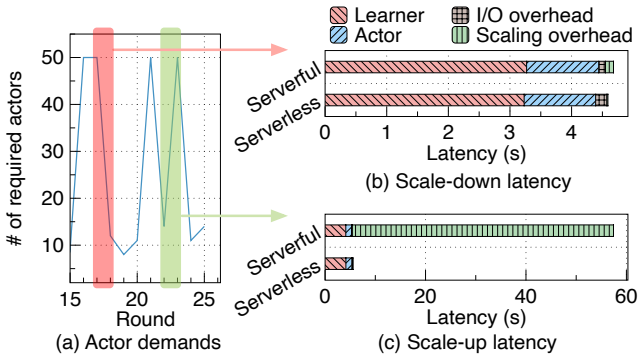


Figure 7: Serverful platforms cannot effectively implement actor scaling due to slow provisioning, which introduces significant scaling overhead. (a) Fluctuating actor demands I_t estimated according to Equation (2). (b) Both serverless and serverful platforms have negligible scale-down latency. (c) Serverful platforms (*i.e.*, AWS EC2) typically have a high scale-up latency due to lengthy provisioning and initialization.

Moreover, the actor scaling design can speed up training by sampling and learning only from the necessary number of trajectories. Although serverful platforms [4, 19, 47] can also improve learner update speed by adjusting the amount of training data, they have difficulty balancing performance and cost. Integrating actor scaling with serverful platforms is impractical due to the high overhead of resizing instances or VMs during training. Even when using a cluster of multiple instances and dynamically turning them on and off, the process remains inefficient. This inefficiency arises because adding instances causes frequent startup delays that increase total training time (*e.g.*, AWS EC2 [4] instances typically take seconds to minutes to initialize), as the system must frequently add or remove instances. We conduct experiments in both serverless and serverful environments using IPPO on AWS EC2 with the *simple-spread* scenario and a fixed random seed, as Fig. 7 shows. We extend the serverful baseline to support actor scaling by dynamically spawning new instances or shutting down redundant ones. Fig. 7(b) and (c) present the per-round latency breakdowns during scale-up and scale-down operations, respectively. The results show that serverful platforms incur minimal extra overhead when scaling down, since active actors do not need to wait for redundant instances to shut down. However, the serverful baseline introduces significant additional latency when scaling up due to the time required to spawn new instances in response to increased actor demand. Using large instances can reduce the frequency of such adjustments, but they are often underutilized, resulting in unnecessary costs.

5 Implementation

MARLess is a generic MARL framework designed to accelerate MARL training and reduce training costs through serverless computing while supporting all mainstream MARL algorithms. We build MARLess on top of Ray RLlib [37], without changing the outer RL interfaces exposed to the user, which ensures the usability and makes it easy for previous RLlib users to adapt within around two

hours. The implementation of MARLess consists of approximately 4K lines of Python code, which is open-sourced[5].

Serverless cluster. We implement our own serverless cluster using AWS EC2 instances to manage the serverless functions in MARLess for simplicity. We deploy the cluster using Docker containers [45], a widely adopted choice in open-source serverless frameworks such as Knative [11] and OpenWhisk [6]. For serverful baselines, learners and actors are implemented as Python processes. The resources allocated to each serverless function are preconfigured and remain fixed throughout a training run. Typically, each function uses 1–2 CPUs and at least 1 GB of memory.

Pre-warm serverless functions. Cold-start latency in serverless computing refers to the delay when a function is invoked for the first time or after being idle, requiring the platform to allocate resources and initialize the environment [9, 68, 71, 83]. To align the initial invocation overhead with real-world serverless environments, we pre-warm the containers during the first round of training. After this initial invocation, containers are kept alive for 10 minutes, following the standard keep-alive mechanism adopted by existing serverless frameworks [6] to reduce the startup overhead of serverless functions. MARLess's design is orthogonal to existing pre-warming techniques [6, 36, 42, 52, 68, 71, 83, 83] and can be easily integrated with them.

State management. To reduce data transfer overhead, we adopt a temporary caching technique [61]. Specifically, we use Redis [60], an in-memory key-value store, as the external cache. All intermediate data, including model weights and trajectories, are efficiently serialized using Pickle [57] and CloudPickle [56].

RL models. The policy and critic models within these functions are implemented using PyTorch [58], with model wrappers inspired by the design of Ray RLlib [37]. The interfaces of the environments and policy models follow Gymnasium's design [79] The original MARL environments are provided by existing libraries [64, 78].

Learner sharing and actor scaling in MARLess. To enable dynamic learner sharing without extra latency, MARLess computes policy similarity in parallel with learner updates, right after actors finish sampling. The latency of building the KDE model is thus hidden if it is shorter than the learner's update time. For actor scaling, we use PyHessian [81] to calculate Hessian eigenvalues for estimating the scaling ratio S_t^j. To further reduce computation time, we follow the method in [84], which approximates these eigenvalues using a subset of trajectories.

6 Evaluation

6.1 Experimental Setup

Testbeds. We deploy all the baselines and MARLess to an AWS `c6a.16xlarge` instance, with 64 AMD EPYC 7R13 vCPU cores and 128 GB CPU memory for MARL training tasks. We also evaluated the setup using a GPU-based testbed with a `c6a.16xlarge` instance and a `g5.16xlarge` instance (8 NVIDIA A10G GPUs). However, training on GPUs was similar or even slower than on CPUs across all baselines and MARLess, due to small model and batch size with frequent CPU-GPU communication in the tested MARL tasks. Thus, we run the experiments with only CPUs for a fair evaluation. Furthermore, we evaluate MARLess's scalability through large-scale experiments on a simulated HPC cluster consisting of fifteen AWS EC2 `c6a.32xlarge` instances, each equipped with 128 AMD EPYC 7R13 vCPUs and 256 GB of CPU memory.

Cost model. Referring to the cost models from existing research [62, 86], we calculate the training cost for serverless baselines as a dollar-unit budget based on the resource requirements and invocation time for each function. The unit price, expressed as dollar-per-resource-second, is based on AWS Lambda's pricing [5]. Pre-warm costs are included in the training costs, while keep-alive periods are excluded. This is consistent with the cost models of existing serverless platforms [5, 46]. For serverful baselines, the training cost is calculated using the unit price of the smallest AWS EC2 instance that can accommodate the workload. In our experiments, the price is based on the `c6a.16xlarge` instance.

Workloads. To comprehensively evaluate MARLess against SOTA baselines, we selected two environments representing a range of workloads, from moderate to heavy, including OpenAI's MPE [40] and StarCraft Multi-Agent Challenge (SMAC) [14, 64]. 1) For MPE, the *simple-spread* task and the *simple-adversary* task are used to evaluate system performance in cooperative and competitive scenarios, respectively. We increase the number of agents to six in both tasks to introduce higher complexity and more effectively evaluate MARLess's capability. 2) For SMAC, two scenarios will be evaluated: `8m` (homogeneous agents),[6] and `3s5z` (heterogeneous agents),[7] covering all major MARL task variations. These scenarios involve micromanagement across agents, closely resembling multi-agent decision-making in scientific computing tasks [3, 7, 66, 88], and provide a comprehensive evaluation of the system's ability to handle diverse multi-agent dynamics.

MARLess's settings. We set the tolerant window size W to 10 for all experiments, and the upper threshold β for scaling actors is set to 0.5 when evaluating MARLess. The learning rate remains fixed throughout training. The decrease detection threshold γ is configured based on the specific algorithm and environment. The maximum number of available actors, I_{max}, for all baselines is set to 50. In MARLess, the minimum number of actors, I_{min}, is set to $\frac{I_{max}}{N}$, where N is the number of agents in the environment. Additionally, we evaluate the sensitivity of these hyperparameters in §6.5.

6.2 Overall Performance

6.2.1 Performance Improvement in MARL Algorithms. We evaluate how MARLess improves training efficiency for SOTA MARL algorithms. To demonstrate its performance and compatibility, we implement four representative algorithms in MARLess, covering independent and centralized, actor-critic and value-based methods. 1) **IPPO** [12] is the multi-agent extension of PPO [67], a popular on-policy actor-critic algorithm. We implement it with Generalized Advantage Estimation (GAE) and surrogate objective clipping. 2) **IQL** [76] is the multi-agent version of Deep Q-Network (DQN) [48], representing a standard off-policy value-based approach. 3) **MAPPO** [82] enhances PPO with a centralized critic for improved stability and remains on-policy. 4) **VDN** [74] builds

[5] https://github.com/IntelliSys-Lab/MARLess-SoCC25

[6] `8m` denotes the task "8 Marines v.s. 8 Marines."

[7] `3s5z` denotes "3 Stalkers & 5 Zealots v.s. 3 Stalkers & 5 Zealots."

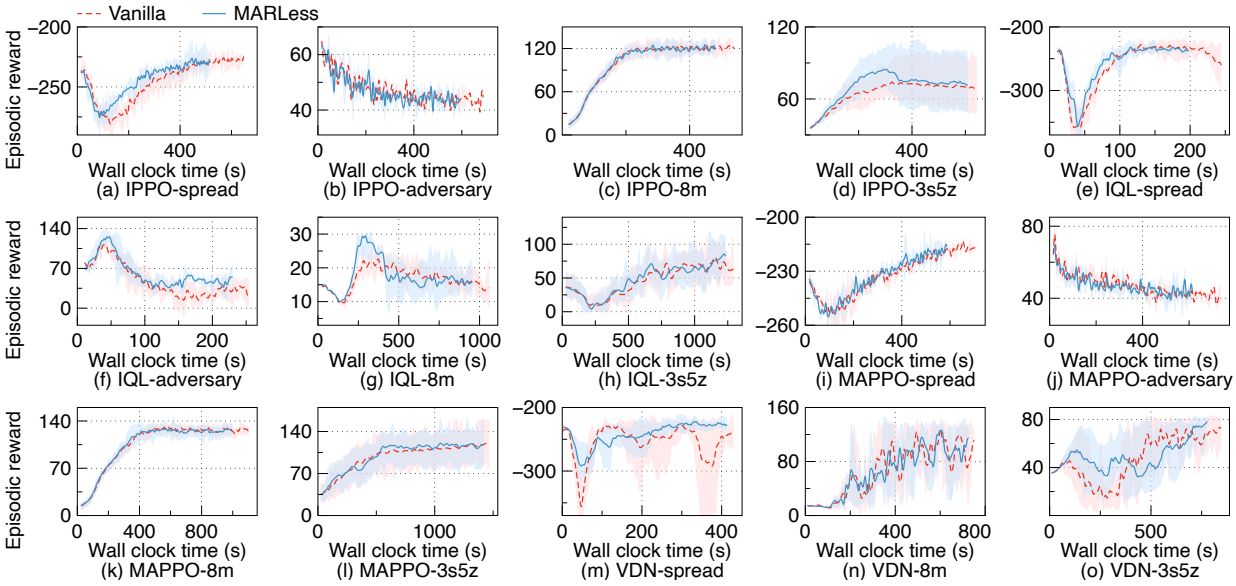


Figure 8: MARLess improves training efficiency for vanilla MARL algorithms.

Table 1: Network architecture settings.

Environment	Layer	Activation	Size
MPE	Fully-connect	Tanh	[64, 64]
SMAC	Fully-connect	Tanh	[128, 256]

on DQN by aggregating agents' value outputs into a shared loss for more stable training.

We use the four vanilla algorithms mentioned above as baselines and integrate them with MARLess for comparison. Hyperparameter selection follows tuned examples from RLlib [37] and MARLlib [28]. The detailed hyperparameter settings for algorithm baselines can be found in the GitHub repository[8]. The decrease detection threshold γ is set to 0.1 for MPE and -1 for SMAC. Table 1 summarizes the policy models' neural network configurations for different MARL environments. For algorithms involving critic networks, the critics are configured identically to the policy networks. The model scale is aligned with previous MARL research [28, 40, 59, 64, 78, 82].

All training processes use the Adam optimizer [31] by default. When running IQL and VDN in the SMAC environment, the optimizer for these algorithms is set to RMSProp [25]. Each algorithm is trained for 100 rounds across all available scenarios selected from two environments. Note that VDN is applicable only to fully cooperative scenarios and does not include *simple-adversary*. The results are derived by averaging values sampled from ten repeated experiments with different random seeds.

Training efficiency. We run each setting multiple times with different initial evaluation sets. The averaged rewards across multiple runs, together with the upper and lower bounds, are visualized in the Fig. 8. The sudden drops and fluctuations in reward curves observed during training are common in DRL and are often attributed to catastrophic interference [17, 24, 32, 40, 53], which is primarily

[8]https://github.com/IntelliSys-Lab/MARLess-SoCC25

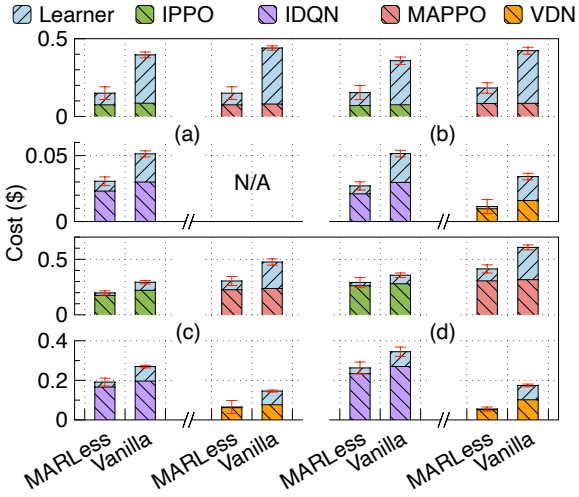


Figure 9: MARLess reduces training costs of vanilla IPPO, IQL, MAPPO, and VDN in (a) *simple-adversary*, (b) *simple-spread*, (c) `8m`, and (d) `3s5z`. Blue bars indicate the costs of learner functions, and the rest represent the costs of actors.

caused by non-stationarity in multi-agent environments. Actor-critic methods, such as IPPO and MAPPO, demonstrate greater stability compared to value-based methods like IQL and VDN. Also, centralized methods do not consistently outperform independent methods, a trend also observed in prior work [12]. MARLess reduces the end-to-end training time for all vanilla algorithms across all scenarios while maintaining or improving training quality, achieving the same or higher final rewards. Compared to the vanilla algorithms, MARLess accelerates the training process by up to 1.27× when running IPPO on MPE's *simple-spread* task.

Training Cost. Fig. 9 illustrates the training costs of the vanilla algorithm baselines and their variants integrated with MARLess.

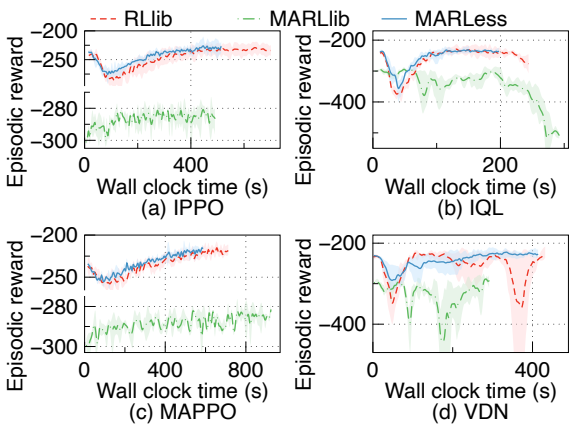


Figure 10: MARLess achieves a better balance between training speed and training quality.

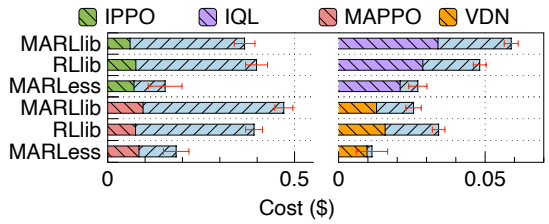


Figure 11: MARLess has lower training costs compared to other MARL frameworks.

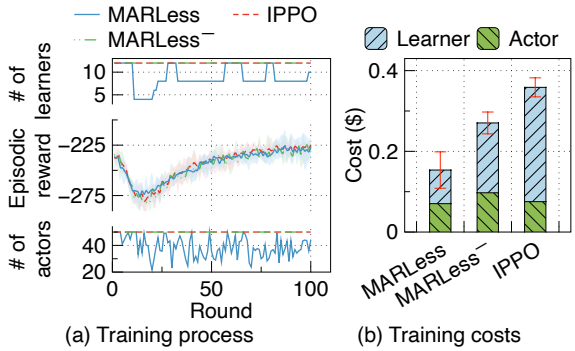


Figure 12: MARLess reduces costs by combining dynamic learner sharing and actor scaling with serverless computing.

In MPE tasks, such as *simple-adversary* and *simple-spread*, actor costs are lower compared to SMAC due to the simplicity and low simulation cost of the MPE environment. In contrast, SMAC involves intensive computation to simulate the StarCraft II game, resulting in higher actor costs. Compared to the original algorithm baselines, MARLess reduces training costs by up to 66% in the MPE environment and up to 68% in the SMAC environment.

6.2.2 Performance Improvement in MARL Frameworks. We also evaluate how MARLess accelerates the training process and reduces training costs compared to SOTA MARL frameworks. Two popular MARL frameworks were selected for comparison with MARLess: 1) **Ray RLlib** [37], an open-source, industrial-grade DRL library built on the Ray [50] cluster for task scheduling. 2) **MARLlib** [28], a SOTA MARL-optimized framework that includes extensive implementations of various MARL algorithms, incorporating multiple implementation tricks from existing work [27]. Due to page limitations, we tested IPPO, IQL, MAPPO, and VDN under the *simple-spread* task for comparison. All algorithms use the same settings as described in §6.2.1.

Training efficiency. Fig. 10 illustrates the episodic rewards during training using four algorithms with MARLess, RLlib, and MARLlib. The results show that MARLlib exhibits lower training quality while achieving the fastest training speed for IPPO and VDN. RLlib maintains similar training quality compared to MARLess, but has the slowest training speed for IPPO and VDN. In contrast, MARLess strikes a balance between training speed and quality, accelerating MARL training while maintaining training quality and achieving comparable final rewards.

Training cost. Fig. 11 illustrates the training costs for MARLess, RLlib, and MARLlib. MARLess achieves significant cost savings, reducing training costs by up to 67% compared to RLlib and 61% compared to MARLlib.

6.3 Effectiveness of MARLess

To validate MARLess's core designs: *dynamic learner sharing* and *actor scaling*, we run IPPO on MPE's *simple-spread* task for 100 rounds. During training, we record the number of learner and actor functions at each round, as shown in Fig. 12(a), to verify that MARLess reduces training costs by releasing unnecessary actors and learners. In addition to the serverful **IPPO** baseline, we also evaluate IPPO directly migrated to a serverless environment without dynamic learner sharing or actor scaling, denoted as **MARLess$^-$**. Since the unit price under this setting for serverless actors is higher than that for serverful actors, directly using serverless platforms without MARLess results in higher actor costs. With MARLess's dynamic learner sharing and actor scaling, training costs for serverful IPPO are reduced by 57%, and training costs for MARLess$^-$ are reduced by 43%, as shown in Fig. 12(b). Moreover, both the actor scaling [84] and learner sharing [29, 87] have been theoretically proved with a bounded convergence of the reward.

6.4 Ablation Study

To further validate the effectiveness of dynamic learner sharing and scalable actors, we introduce two additional variants of MARLess for comparison: 1) **MARLess w/o scaling**: This variant runs MARLess with a fixed number of actors set to I_{max}. 2) **MARLess w/o sharing**: This variant runs MARLess with fixed learner functions, disabling dynamic learner sharing. In this setup, the number of learner functions corresponds to the number of agents.

We run IPPO in MPE's *simple-spread* with the same setting as described in §6.2.1. Fig. 13(a) shows that MARLess accelerates the training process by up to 25% compared to its variants, without compromising training quality. Fig. 13(b) demonstrates that MARLess reduces actor costs by 33% through actor scaling compared to w/o scaling, and lowers learner costs by 49% through learner sharing among agents compared to w/o sharing. By combining both features, MARLess achieves optimal resource efficiency.

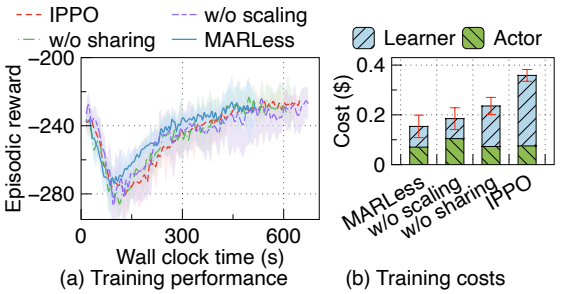


Figure 13: Ablation study of MARLess.

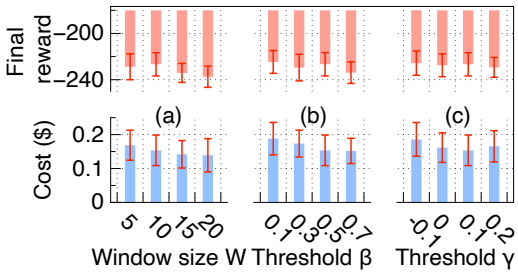


Figure 14: Sensitivity analysis of MARLess.

6.5 Sensitivity Analysis

We analyze the sensitivity of three parameters in MARLess: the tolerant window size W, the scaling upper threshold β, and the decrease detection threshold γ. The same experiment described in §6.2.1 is conducted, *i.e.*, training IPPO on the *simple-spread* task, with varying values for these parameters. Fig. 14 presents the sensitivity analysis results for MARLess.

Tolerant window size W. In previous evaluation experiments, the window size W was set to 10. For sensitivity analysis, we vary W from 5 to 20 in steps of 5. As shown in Fig. 14(a), increasing W reduces the training cost since MARLess switches strategies less frequently. However, when W exceeds 10, the final reward starts to decline because MARLess becomes slower at adapting the correct sharing strategy when rewards are low.

Scaling threshold β. In the evaluation section, the upper threshold β for scaling actors is set to 0.5. For sensitivity analysis, β is varied from 0.1 to 0.7 in steps of 0.2. As shown in Fig. 14(b), increasing β reduces training costs because MARLess scales the number of actors to the maximum required amount less frequently. However, when β reaches 0.7 or higher, the final reward decreases due to insufficient data for effective training.

Decrease detection threshold γ. The value of γ is influenced by the choice of algorithm and the MARL environment used in MARLess. A lower γ triggers more frequent sharing strategy switches, typically reducing learner costs but potentially increasing actor costs due to frequent refreshing of the tolerant window. Excessive switching can also harm training quality, as illustrated in Fig. 14(c). If users prefer not to tune this parameter for specific settings, setting γ to 0 provides a relatively balanced performance.

6.6 Scalability

Actor scalability on HPC cluster. We evaluate the scalability of MARLess by varying the maximum number of actors, increasing the limit from 50 to 1,850 in increments of 450. Using the settings described in §6.2.1, we run IPPO, IQL, MAPPO, and VDN on MPE's *simple-spread* task and measure the end-to-end training time. The experiments are conducted on a simulated HPC cluster consisting of fifteen AWS EC2 `c6a.32xlarge` instances. We limit each training run to 50 rounds. Fig. 15(a) presents the average total training time for the four algorithms across different actor amounts. The training time of IPPO and MAPPO decreases as the number of actors increases because both are on-policy methods that may trigger multiple sampling rounds per update. More actors provide larger batches per round, reducing the number of sampling triggers. When the actor count exceeds 160, only one sampling trigger is needed, so further scaling brings no speedup. In contrast, IQL and VDN slow down with more actors due to the overhead of the Prioritized Experience Replay (PER) [65] buffer, which must preprocess data and compute priority scores for each trajectory before storage.

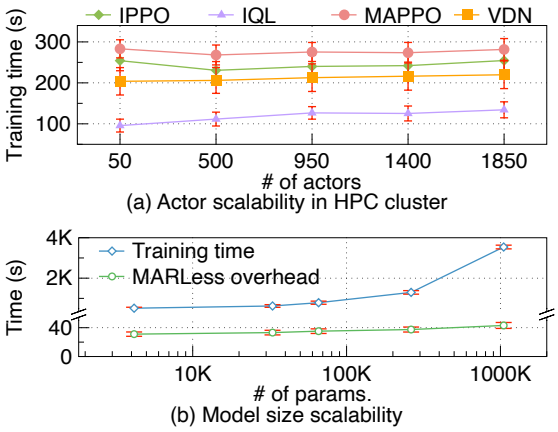


Figure 15: MARLess's scalability.

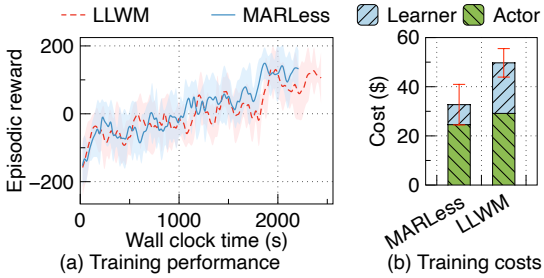


Figure 16: MARLess speeds up the training and reduces the costs for large-scale scientific MARL workload.

Model size scalability. To test MARLess with model size scalability, we run IPPO on MPE's *simple-spread* task using models ranging from *4K* to *10K* parameters, which are aligned with typical MARL settings [12, 21, 29, 40, 82]. We measure total training time and MARLess overhead, defined as the total time minus actor sampling and learner updates, using the setup described in §6.2.1. As shown in Fig. 15(b), training time increases sharply with model size, while MARLess overhead remains small and grows much slower.

Large-scale scientific workload. To further validate our system in real-world scenarios, we integrate and run a turbulent flow simulation MARL workload [7] using MARLess. This workload applies a MARL algorithm to train the log-law-based wall

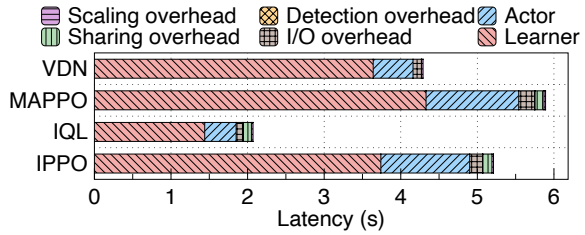


Figure 17: Latency breakdown when running four algorithms with MARLess in one single round.

model (LLWM) for turbulent flow simulation. We conduct the large-scale scientific MARL experiment on our HPC testbed. Fig. 16 shows that MARLess achieves an average of 34% cost reduction and 1.1× speedup compared to the serverful baseline.

6.7 Latency Breakdown and Overheads

We categorize the latency and overheads into the following components: 1) **I/O overhead** includes time spent saving and retrieving states from external storage. 2) **Actor sampling time** is the time actors take to sample trajectories. 3) **Learner updating time** is the time spent when the learner updates policies and critic networks. 4) **Decrease detection overhead** is the time spent detecting performance decreases. 5) **Learner sharing overhead** is the time incurred by the dynamic learner sharing mechanism. 6) **Actor scaling overhead** is the time required to calculate ideal actor numbers and scale actors dynamically based on needs. Fig. 17 presents the latency breakdown and overheads for a single round when running *simple-spread* with four different algorithms. The experimental setup follows the same configuration as described in §6.2.1.

7 Related Work

MARL algorithms. Existing MARL algorithms can be categorized based on two key aspects. First, by policy construction, they are divided into value-based methods (*e.g.*, IQL [76], VDN [74], QMIX [59]) derived from DQN [48], and actor-critic methods (*e.g.*, IPPO [12], MAPPO [82], MADDPG [40]) extended from Soft Actor-Critic (SAC) [22]. Second, by training scheme, they are classified into independent learning [12, 76], where each agent trains separately, and centralized learning [40, 59, 74, 82], which utilizes shared information for improved training.

MARL frameworks. On the other hand, several MARL frameworks have been proposed recently [27, 28, 37, 44, 44, 54, 64, 80, 90]. We choose RLlib [37] and MARLlib [28] as baselines because RLlib offers industry-level scalability for heavy MARL workloads, and MARLlib builds on RLlib to implement diverse algorithms. On the contrary, PyMARL [64] and PyMARL2 [27] support only value-based algorithms, EpyMARL [54] is limited to RNN-based architectures, and MALib [90] prioritizes model searching over single-training optimization. SRL [44] focuses on optimizing heterogeneous resource allocation for actors and learners, which is orthogonal to MARLess and can be easily integrated with it.

Learner sharing in MARL. Learner sharing was first explored to enable cooperation among agents in early research comparing independent and cooperative agents [77]. It is widely applied in cooperative MARL tasks with homogeneous agents [12, 21, 26, 33]. Indexed learner sharing [18] uses an index as input to policy models to generate agent-specific actions. Selective learner sharing [10] trains an encoder to create latent representations for grouping agents before training. MAPD [29] dynamically updates policy assignments using an encoder based on the agent's information. GoMARL [87] employs fine-grained subgroups for partially shared parameters in value-based algorithms. Unlike these methods, MARLess leverages a lightweight dynamic learner sharing approach based on KL divergence from previous trajectories.

Scalable DRL training with serverless computing. In recent years, serverless DRL training frameworks have been proposed [84–86]. MinionsRL [85] firstly introduced a serverless paradigm for DRL training, employing a DRL-driven scheduler to scale actors dynamically. Nitro [84] advanced this approach by leveraging the Hessian matrix to capture the convexity of the objective surface curvature. Stellaris [86], on the other hand, focuses on asynchronous training. However, all of the above research primarily focuses on SARL, and their serverless-specific optimizations are not compatible with MARL training. Existing serverless RL frameworks fail to explore the potential benefits of serverless computing for distributed MARL. MARLess is the first framework designed specifically for serverless MARL training.

8 Conclusion

This paper proposes MARLess, the first serverless MARL training paradigm, which is compatible with a wide range of MARL algorithms. By combining MARL training with the flexibility and scalability of serverless computing, MARLess leverages dynamic agent similarities and varying data requirements. It features a novel dynamic learner sharing technique and an actor scaling rule tailored for MARL, balancing the diverse data demands among policies. We evaluate MARLess with four representative MARL algorithms and compare it against two popular frameworks. Experiments on AWS EC2 testbeds show that MARLess outperforms state-of-the-art MARL baselines, achieving up to 1.27× faster training speeds and reducing training costs by up to 68%. Experimental results from a simulated HPC cluster with 1,920 vCPUs demonstrate MARLess's scalability. Moreover, MARLess achieves a 34% cost reduction and 1.1× speedup on large-scale scientific MARL workloads of wall-models for turbulent flow simulation, further validating its potential for real-world tasks with intensive computational demands.

Acknowledgments

The work of Rui Wei, Hanfei Yu, and Hao Wang was supported in part by NSF 2527416, 2534241, 2534286, and 2523997, and the AWS Cloud Credit for Research program. The work of Devesh Tiwari was supported in part by NSF 2124897. The work of Jian Li was supported in part by NSF 2315614 and 2337914. This research was supported by the National Artificial Intelligence Research Resource (NAIRR) Pilot allocation 240269. This work used Jetstream2 at Indiana University through allocation 240498 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants 2138259, 2138286, 2138307, 2137603, and 2138296.

References

[1] Akshat Agarwal, Sumit Kumar, and Katia Sycara. 2019. Learning Transferable Cooperative Behavior in Multi-Agent Teams. arXiv:1906.01202 [cs.LG] https://arxiv.org/abs/1906.01202

[2] Akash Agrawal, Sung Jun Won, Tushar Sharma, Mayuri Deshpande, and Christopher McComb. 2021. A multi-agent reinforcement learning framework for intelligent manufacturing with autonomous mobile robots. *Proceedings of the Design Society* 1 (2021), 161–170.

[3] Hanane Allioui, Mazin Abed Mohammed, Narjes Benameur, Belal Al-Khateeb, Karrar Hameed Abdulkareem, Begonya Garcia-Zapirain, Robertas Damaševičius, and Rytis Maskeliūnas. 2022. A multi-agent deep reinforcement learning approach for enhancement of COVID-19 CT image segmentation. *Journal of personalized medicine* 12, 2 (2022), 309.

[4] Amazon Web Services. 2025. AWS Elastic Compute Cloud. https://aws.amazon.com/ec2/.

[5] Amazon Web Services. 2025. AWS Lambda pricing rules. https://aws.amazon.com/lambda/pricing/.

[6] Apache. 2018. Apache OpenWhisk Official Website. https://openwhisk.apache.org.

[7] H Jane Bae and Petros Koumoutsakos. 2022. Scientific multi-agent reinforcement learning for wall-models of turbulent flows. *Nature Communications* 13, 1 (2022), 1443.

[8] Lucian Busoniu, Robert Babuska, and Bart De Schutter. 2008. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 38, 2 (2008), 156–172.

[9] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*.

[10] Filippos Christianos, Georgios Papoudakis, Arrasy Rahman, and Stefano V. Albrecht. 2021. Scaling Multi-Agent Reinforcement Learning with Selective Parameter Sharing. *ArXiv* abs/2102.07475 (2021). https://api.semanticscholar.org/CorpusID:231924963

[11] Cloud Native Computing Foundation. 2018. Knative. https://knative.dev/docs/.

[12] Christian Schroeder de Witt, Tarun Gupta, Denys Makoviichuk, Viktor Makoviychuk, Philip H. S. Torr, Mingfei Sun, and Shimon Whiteson. 2020. Is Independent Learning All You Need in the StarCraft Multi-Agent Challenge? arXiv:2011.09533 [cs.AI] https://arxiv.org/abs/2011.09533

[13] Aditya Devarakonda, Maxim Naumov, and Michael Garland. 2017. AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks. *CoRR* abs/1712.02029 (2017). arXiv:1712.02029 http://arxiv.org/abs/1712.02029

[14] Benjamin Ellis, Jonathan Cook, Skander Moalla, Mikayel Samvelyan, Mingfei Sun, Anuj Mahajan, Jakob Nicolaus Foerster, and Shimon Whiteson. 2023. SMACv2: An Improved Benchmark for Cooperative Multi-Agent Reinforcement Learning. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=5OjLGiJW3u

[15] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. 2020. SEED RL: Scalable and Efficient Deep-RL with Accelerated Central Inference. arXiv:1910.06591 [cs.LG] https://arxiv.org/abs/1910.06591

[16] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad 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 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 1407–1416. https://proceedings.mlr.press/v80/espeholt18a.html

[17] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip H. S. Torr, Pushmeet Kohli, and Shimon Whiteson. 2018. Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning. arXiv:1702.08887 [cs.AI] https://arxiv.org/abs/1702.08887

[18] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. 2016. Learning to communicate with Deep multi-agent reinforcement learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 2145–2153.

[19] Google Cloud Platform. 2025. GCP Compute Engine Instance. https://cloud.google.com/compute/docs/instances.

[20] Shangding Gu, Jakub Grudzien Kuba, Yuanpei Chen, Yali Du, Long Yang, Alois Knoll, and Yaodong Yang. 2023. Safe multi-agent reinforcement learning for multi-robot control. *Artificial Intelligence* 319 (2023), 103905.

[21] Jayesh K. Gupta, Maxim Egorov, and Mykel Kochenderfer. 2017. Cooperative Multi-agent Control Using Deep Reinforcement Learning. In *Autonomous Agents and Multiagent Systems*, Gita Sukthankar and Juan A. Rodriguez-Aguilar (Eds.). Springer International Publishing, Cham, 66–83.

[22] 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. *CoRR* abs/1812.05905 (2018). arXiv:1812.05905 http://arxiv.org/abs/1812.05905

[23] Jinbo Han, Xingda Wei, Rong Chen, and Haibo Chen. 2024. Seraph: A Performance-Cost Aware Tuner for Training Reinforcement Learning Model on Serverless Computing. In *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*. 95–101.

[24] Pablo Hernandez-Leal, Michael Kaisers, Tim Baarslag, and Enrique Munoz de Cote. 2019. A Survey of Learning in Multiagent Environments: Dealing with Non-Stationarity. arXiv:1707.09183 [cs.MA] https://arxiv.org/abs/1707.09183

[25] Geoffrey Hinton, Nish Srivastava, and Kevin Swersky. 2017. Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf Lecture slides, CSC321.

[26] Hengyuan Hu and Jakob N. Foerster. 2019. Simplified Action Decoder for Deep Multi-Agent Reinforcement Learning. *ArXiv* abs/1912.02288 (2019). https://api.semanticscholar.org/CorpusID:208637067

[27] Jian Hu, Siyang Jiang, Seth Austin Harding, Haibin Wu, and Shih wei Liao. 2023. Rethinking the Implementation Tricks and Monotonicity Constraint in Cooperative Multi-Agent Reinforcement Learning. arXiv:2102.03479 [cs.LG] https://arxiv.org/abs/2102.03479

[28] Siyi Hu, Yifan Zhong, Minquan Gao, Weixun Wang, Hao Dong, Xiaodan Liang, Zhihui Li, Xiaojun Chang, and Yaodong Yang. 2023. MARLlib: A Scalable and Efficient Multi-agent Reinforcement Learning Library. *Journal of Machine Learning Research* (2023).

[29] Tianyi Hu, Zhiqiang Pu, Xiaolin Ai, Tenghai Qiu, and Jianqiang Yi. 2024. Measuring Policy Distance for Multi-Agent Reinforcement Learning. In *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems*. 834–842.

[30] Sham Kakade and John Langford. 2020. A Closer Look at Deep Policy Gradients. In *International Conference on Learning Representations (ICLR)*.

[31] Diederik P Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980* (2014).

[32] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences* 114, 13 (March 2017), 3521–3526. https://doi.org/10.1073/pnas.1611835114

[33] Jakub Grudzien Kuba, Ruiqing Chen, Munning Wen, Ying Wen, Fanglei Sun, Jun Wang, and Yaodong Yang. 2021. Trust Region Policy Optimisation in Multi-Agent Reinforcement Learning. *ArXiv* abs/2109.11251 (2021). https://api.semanticscholar.org/CorpusID:237605219

[34] Marius Kurz, Philipp Offenhäuser, Dominic Viola, Oleksandr Shcherbakov, Michael Resch, and Andrea Beck. 2022. Deep reinforcement learning for computational fluid dynamics on HPC systems. *Journal of Computational Science* 65 (2022), 101884.

[35] Yujing Li, Su Su, Minghao Zhang, Qiujiang Liu, Xiaobo Nie, Mingchao Xia, and Dan D Micu. 2023. Multi-agent graph reinforcement learning method for electric vehicle on-route charging guidance in coupled transportation electrification. *IEEE Transactions on Sustainable Energy* 15, 2 (2023), 1180–1193.

[36] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *2022 USENIX annual technical conference (USENIX ATC)*.

[37] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. 2017. RLlib: Abstractions for Distributed Reinforcement Learning. In *International Conference on Machine Learning*. https://api.semanticscholar.org/CorpusID:49546141

[38] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2019. Continuous control with deep reinforcement learning. arXiv:1509.02971 [cs.LG] https://arxiv.org/abs/1509.02971

[39] Zongkai Liu, Chao Yu, Yaodong Yang, peng sun, Zifan Wu, and Yuan Li. 2022. A Unified Diversity Measure for Multiagent Reinforcement Learning. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 10339–10352. https://proceedings.neurips.cc/paper_files/paper/2022/file/435cce71b4007699041dfffa4f034079-Paper-Conference.pdf

[40] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. 2017. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. *Neural Information Processing Systems (NIPS)* (2017).

[41] Rohan Mahapatra, Soroush Ghodrati, Byung Hoon Ahn, Sean Kinzer, Shu-Ting Wang, Hanyang Xu, Lavanya Karthikeyan, Hardik Sharma, Amir Yazdanbakhsh, Mohammad Alian, et al. 2024. In-storage Domain-Specific Acceleration for Serverless Computing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[42] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing,

Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

[43] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. An Empirical Model of Large-Batch Training. *CoRR* abs/1812.06162 (2018). arXiv:1812.06162 http://arxiv.org/abs/1812.06162

[44] Zhiyu Mei, Wei Fu, Jiaxuan Gao, Guangju Wang, Huanchen Zhang, and Yi Wu. 2024. SRL: Scaling Distributed Reinforcement Learning to Over Ten Thousand Cores. arXiv:2306.16688 [cs.DC] https://arxiv.org/abs/2306.16688

[45] Dirk Merkel et al. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal* (2014).

[46] Microsoft Azure. 2025. Microsoft Azure Functions. https://azure.microsoft.com/pricing/details/functions/.

[47] Microsoft Azure. 2025. Microsoft Azure Virtual Machines. https://azure.microsoft.com/pricing/details/virtual-machines/.

[48] Volodymyr Mnih. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[49] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. arXiv:1602.01783 [cs.LG] https://arxiv.org/abs/1602.01783

[50] 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. arXiv:1712.05889 [cs.DC] https://arxiv.org/abs/1712.05889

[51] Guido Novati, Hugues Lascombes de Laroussilhe, and Petros Koumoutsakos. 2021. Automating turbulence modelling by multi-agent reinforcement learning. *Nature Machine Intelligence* 3, 1 (2021), 87–96.

[52] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX annual technical conference (USENIX ATC)*.

[53] Georgios Papoudakis, Filippos Christianos, Arrasy Rahman, and Stefano V Albrecht. 2019. Dealing with non-stationarity in multi-agent deep reinforcement learning. *arXiv preprint arXiv:1906.04737* (2019).

[54] Georgios Papoudakis, Filippos Christianos, Lukas Schäfer, and Stefano V. Albrecht. 2021. Benchmarking Multi-Agent Deep Reinforcement Learning Algorithms in Cooperative Tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks (NeurIPS)*. http://arxiv.org/abs/2006.07869

[55] Jack Parker-Holder, Luke Metz, Cinjon Resnick, Hengyuan Hu, Adam Lerer, Alistair Letcher, Alexander Peysakhovich, Aldo Pacchiano, and Jakob Foerster. 2020. Ridge rider: Finding Diverse Solutions by Following Eigenvectors of the Hessian. *Advances in Neural Information Processing Systems (NIPS)* (2020).

[56] Python. 2008. CloudPickle — Extension of Pickle. https://pypi.org/project/cloudpickle/.

[57] Python. 2008. Pickle — Python Object Serialization. https://docs.python.org/3/library/pickle.html.

[58] PyTorch. 2018. PyTorch: Tensors and Dynamic Neural Networks in Python with Strong GPU Acceleration. https://pytorch.org.

[59] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder de Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. 2018. QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning. arXiv:1803.11485 [cs.LG] https://arxiv.org/abs/1803.11485

[60] Redis. 2009. Redis Official Website. http://redis.io/.

[61] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.

[62] Francisco Romero, Mark Zhao, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3472883.3486972

[63] Pranab Sahoo, Ashutosh Tripathi, Sriparna Saha, and Samrat Mondal. 2024. Fedmrl: Data heterogeneity aware federated multi-agent deep reinforcement learning for medical imaging. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 640–649.

[64] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder de Witt, Gregory Farquhar, Nantas Nardelli, Tim G. J. Rudner, Chia-Man Hung, Philiph H. S. Torr, Jakob Foerster, and Shimon Whiteson. 2019. The StarCraft Multi-Agent Challenge. *CoRR* abs/1902.04043 (2019).

[65] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized Experience Replay. arXiv:1511.05952 [cs.LG] https://arxiv.org/abs/1511.05952

[66] Paul Maria Scheikl, Balázs Gyenes, Tornike Davitashvili, Rayan Younis, André Schulze, Beat P Müller-Stich, Gerhard Neumann, Martin Wagner, and Franziska Mathis-Ullrich. 2021. Cooperative assistance in robotic surgery through multi-agent reinforcement learning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 1859–1864.

[67] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG] https://arxiv.org/abs/1707.06347

[68] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX annual technical conference (USENIX ATC)*.

[69] Thanveer Shaik, Xiaohui Tao, Lin Li, Haoran Xie, Hong-Ning Dai, Feng Zhao, and Jianming Yong. 2023. Adaptive Multi-Agent Deep Reinforcement Learning for Timely Healthcare Interventions. *arXiv preprint arXiv:2309.10980* (2023).

[70] Shai Shalev-Shwartz, Shaked Shammah, and Amnon Shashua. 2016. Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint arXiv:1610.03295* (2016).

[71] Yifan Sui, Hanfei Yu, Yitao Hu, Jianxun Li, and Hao Wang. 2024. Pre-Warming is Not Enough: Accelerating Serverless Inference With Opportunistic Pre-Loading. In *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC)*.

[72] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. 2016. Learning Multiagent Communication with Backpropagation. arXiv:1605.07736 [cs.LG] https://arxiv.org/abs/1605.07736

[73] Ryan Sullivan, Justin K Terry, Benjamin Black, and John P Dickerson. 2022. Cliff Diving: Exploring Reward Surfaces in Reinforcement Learning Environments. In *Nineteenth International Conference on Machine Learning (ICML)*.

[74] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. 2017. Value-Decomposition Networks For Cooperative Multi-Agent Learning. arXiv:1706.05296 [cs.AI] https://arxiv.org/abs/1706.05296

[75] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems* (Denver, CO) *(NIPS'99)*. MIT Press, Cambridge, MA, USA, 1057–1063.

[76] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. 2015. Multiagent cooperation and competition with deep reinforcement learning. *PLoS ONE* 12 (2015). https://api.semanticscholar.org/CorpusID:12046082

[77] Ming Tan. 1997. Multi-Agent Reinforcement Learning: Independent versus Cooperative Agents. In *International Conference on Machine Learning*. https://api.semanticscholar.org/CorpusID:274281842

[78] J Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, et al. 2021. Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems* 34 (2021), 15032–15043.

[79] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. 2024. Gymnasium: A Standard Interface for Reinforcement Learning Environments. arXiv:2407.17032 [cs.LG] https://arxiv.org/abs/2407.17032

[80] Nicholas Ustaran-Anderegg, Michael Pratt, and Jaime Sabal-Bermudez. [n. d.]. *AgileRL*. https://github.com/AgileRL/AgileRL

[81] Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael Mahoney. 2020. PyHessian: Neural Networks Through the Lens of the Hessian. arXiv:1912.07145 [cs.LG] https://arxiv.org/abs/1912.07145

[82] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. 2022. The Surprising Effectiveness of PPO in Cooperative Multi-Agent Games. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

[83] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[84] Hanfei Yu, Jacob Carter, Hao Wang, Devesh Tiwari, Jian Li, and Seung-Jong Park. 2025. Nitro: Boosting Distributed Reinforcement Learning with Serverless Computing. In *51st International Conference on Very Large Data Bases (VLDB)*.

[85] Hanfei Yu, Jian Li, Yang Hua, Xu Yuan, and Hao Wang. 2024. Cheaper and Faster: Distributed Deep Reinforcement Learning with Serverless Computing. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

[86] Hanfei Yu, Hao Wang, Devesh Tiwari, Jian Li, and Seung-Jong Park. 2024. Stellaris: Staleness-Aware Distributed Reinforcement Learning with Serverless Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*.

[87] Yifan Zang, Jinmin He, Kai Li, Haobo Fu, Qiang Fu, Junliang Xing, and Jian Cheng. 2023. Automatic Grouping for Efficient Cooperative Multi-Agent Reinforcement

Learning. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 46105–46121. https://proceedings.neurips.cc/paper_files/paper/2023/file/906c860f1b7515a8ffec02dcdac74048-Paper-Conference.pdf

[88] Weijia Zhang, Hao Liu, Fan Wang, Tong Xu, Haoran Xin, Dejing Dou, and Hui Xiong. 2021. Intelligent electric vehicle charging recommendation based on multi-agent reinforcement learning. In *Proceedings of the Web Conference 2021*. 1856–1867.

[89] Weijia Zhang, Hao Liu, Hui Xiong, Tong Xu, Fan Wang, Haoran Xin, and Hua Wu. 2022. RLCharge: Imitative multi-agent spatiotemporal reinforcement learning for electric vehicle charging station recommendation. *IEEE Transactions on Knowledge and Data Engineering* 35, 6 (2022), 6290–6304.

[90] Ming Zhou, Ziyu Wan, Hanjing Wang, Muning Wen, Runzhe Wu, Ying Wen, Yaodong Yang, Yong Yu, Jun Wang, and Weinan Zhang. 2023. MALib: A Parallel Framework for Population-based Multi-agent Reinforcement Learning. *Journal of Machine Learning Research* 24, 150 (2023), 1–12. http://jmlr.org/papers/v24/22-0169.html

[91] Yutai Zhou, Shawn Manuel, Peter Morales, Sheng Li, Jaime Pena, and Ross Allen. 2020. Towards a distributed framework for multi-agent reinforcement learning research. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–9.