Reinforcement Learning for Resource Management in Multi-tenant Serverless Platforms

Haoran Qiu
University of Illinois, Urbana-Champaign
haoranq4@illinois.edu

Weichao Mao
University of Illinois, Urbana-Champaign
weichao2@illinois.edu

Archit Patke
University of Illinois, Urbana-Champaign
apatke@illinois.edu

Chen Wang
IBM Research
chen.wang1@ibm.com

Hubertus Franke
IBM Research
frankeh@us.ibm.com

Zbigniew T. Kalbarczyk
University of Illinois, Urbana-Champaign
kalbarcz@illinois.edu

Tamer Başar
University of Illinois, Urbana-Champaign
basar1@illinois.edu

Ravishankar K. Iyer
University of Illinois, Urbana-Champaign
rkiyer@illinois.edu

Abstract

Serverless Function-as-a-Service (FaaS) is an emerging cloud computing paradigm that frees application developers from infrastructure management tasks such as resource provisioning and scaling. To reduce the tail latency of functions and improve resource utilization, recent research has been focused on applying online learning algorithms such as reinforcement learning (RL) to manage resources. Compared to existing heuristics-based resource management approaches, RL-based approaches eliminate humans in the loop and avoid the painstaking generation of heuristics. In this paper, we show that the state-of-the-art single-agent RL algorithm (S-RL) suffers up to 4.6× higher function tail latency degradation on multi-tenant serverless FaaS platforms and is unable to converge during training. We then propose and implement a customized multi-agent RL algorithm based on Proximal Policy Optimization, i.e., multi-agent PPO (MA-PPO). We show that in multi-tenant environments, MA-PPO enables each agent to be trained until convergence and provides online performance comparable to S-RL in single-tenant cases with less than 10% degradation. Besides, MA-PPO provides a 4.4× improvement in S-RL performance (in terms of function tail latency) in multi-tenant cases.

CCS Concepts: • **Software and its engineering → Cloud computing**; • **Computing methodologies → Multi-agent planning**; **Multi-agent systems**.

EuroMLSys'22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9254-9/22/04…$15.00
https://doi.org/10.1145/3517207.3526971

Keywords: Function-as-a-Service, serverless computing, resource allocation, reinforcement learning, multi-agent

ACM Reference Format:
Haoran Qiu, Weichao Mao, Archit Patke, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2022. Reinforcement Learning for Resource Management in Multi-tenant Serverless Platforms. In *2nd European Workshop on Machine Learning and Systems (EuroMLSys'22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3517207.3526971

1 Introduction

Serverless Function-as-a-Service (FaaS) is a category of cloud computing services that frees customers from managing resource allocation for their functions (e.g, scaling the number of containers and their resource limits, and scheduling functions to servers) while leaving the performance-utilization trade-off problem with the cloud provider [3, 17, 22, 27]. The problem of resource management to achieve performance and utilization objectives is at its core an intractable NP-hard problem [4, 13]. While the majority of the problems are approached using meticulously designed heuristics with extensive application- and system-specific domain-expert-driven tuning, a substantial line of work has recently been focused on learning-based approaches such as reinforcement learning (RL) [4, 8, 11, 12, 16, 18, 19, 30, 32, 34].

As a viable alternative to human-generated heuristics, RL enables an agent to learn the optimal policy directly from interacting with the environment. RL is especially well-suited for resource management problems [12] due to the sequential nature of the decision-making process, and it has been shown that deep neural networks can express the complex dynamics and decision-making policies in such system-application environment. Since resource management decisions made for each serverless function are highly repetitive, an abundance of data can be easily generated for training such RL algorithms. By continuing to learn, the RL agent can optimize for a specific workload and adapt to varying conditions.

Motivation. Despite the recent successes, existing RL-based solutions [4, 8, 11, 16, 18, 19, 29, 34] are all single-agent RL (S-RL) where each RL agent controls one function. In contrast, a serverless FaaS platform is multi-tenant where heterogeneous functions from all customers compete for shared resources in a cluster. Multi-tenancy makes the environment *non-stationary* from each agent's own perspective, as it is also affected by the actions of other agents, which breaks the standard assumption that underpins S-RL algorithms. Since the transitions and rewards depend on the joint actions of all agents, whose decision policies keep changing in the learning process, each agent can enter an endless cycle of adapting to other agents in the shared environment. We show in §5.1 that in multi-tenant scenarios, the state-of-the-art S-RL algorithm suffers up to 4.6× higher function tail latency degradation and is unable to converge during training.

Design principles. To address the convergence failure during training and the online performance degradation problem for multiple RL agents in a shared environment, we target the following challenging objectives: (a) *Performance isolation*: agent performance in multi-tenant environments should be comparable to single-RL agent performance in single-tenant scenarios. (b) *Scalability*: a serverless FaaS platform is multi-tenant and new functions from different customers can be increasingly registered. (c) *Fast training*: functions from any customer can be registered, removed, or updated at any time, which changes the joint state space.

Solution. Based on the defined design principles, we propose and implement MA-PPO (§6.2), a customized multi-agent RL algorithm motivated by its single-agent counterpart PPO [20]. In MA-PPO, we customize PPO, where each agent treats the other agents as part of the environment. In addition, we use aggregated and mean state-action values across all the other agents to make the MA-PPO model agnostic to agent order or the size of the agent group, and thus more scalable and faster to train.

Results. An evaluation of MA-PPO (§6.3) shows that it enables each agent's behavior to converge during training and that it provides online performance in multi-tenant cases comparable to S-RL in single-tenant cases with less than 10% degradation. Besides, functions controlled by MA-PPO achieve up to 4.4× improvement in tail latencies of function invocations compared to S-RL in multi-tenant cases.

2 Background and Related Work

2.1 Serverless Function-as-a-Service

Serverless Function-as-a-Service (FaaS) is a cloud programming model and architecture where customers execute small code snippets without any control over the resources on which the code runs [3]. A serverless FaaS platform runs functions in response to invocations (i.e., requests) from end-users or clients. It consists of a central *controller* and a group of *invokers*. In our study, we chose OpenWhisk [10], a production-grade serverless FaaS platform based on Docker containers. Fig. 1 shows the architecture of a distributed OpenWhisk platform. The controller (labeled as ❺) creates function containers, allocates CPU and RAM for each function container, and assigns the containers to invokers (labeled as ❻). When requests arrive via the API gateway, the controller distributes the requests to invokers. An invoker executes the function after it receives a request and the execution results are written to a data store ❷.

2.2 Learning-based Resource Management

Lately, ML-based resource management approaches have gained significant momentum towards achieving application service-level objectives (SLOs) [15, 16, 29, 30, 35]. In [16], the authors proposed an ML-based resource management framework, FIRM, to tackle the issue of microservices under-utilization and SLO violations. The two-tier ML model is responsible for first identifying the microservices that cause SLO violations and then mitigating those violations via dynamic resource reprovisioning. Sinan [30], an SLO-aware resource management framework for microservices, is another model that employs ML algorithms to improve resource utilization while meeting end-to-end performance SLOs.

Since Mao et al. proposed a reinforcement learning (RL)-based solution for scheduling cluster resources [12], there has been a line of work using RL for resource management [4, 8, 11, 16, 18, 19, 32, 34]. Schuler et al. [19] proposed a Q-Learning-based autoscaler that decides the horizontal concurrency for a serverless function with the only objective being minimizing the function latency. Zafeiropoulos et al. [34] also applied Q-Learning to threshold-based autoscaling for determining the CPU and memory usage threshold. Both SLO violation and resource utilization are considered in the reward function of each RL agent for a function. FaaS-Rank [32] is an RL-based serverless function scheduler that uses a policy gradient method (i.e., PPO) to minimize function completion time. However, no existing work has been focused on addressing the single-agent RL failure in multi-tenant environments (as we will describe in §5.1).

3 Reinforcement Learning Formulation

In this section, we present the problem formulation and our design for online serverless resource management with RL. In the following, we first provide a brief introduction to RL.

RL Primer. An RL agent solves a *sequential decision-making problem* (modeled as a Markov decision process, or MDP) by interacting with an unknown environment. At each discrete time step t, the agent observes the current *state* of the environment $s_t \in S$, and performs an *action* $a_t \in A$ based on its *policy* $\pi_\theta(s)$ (parameterized by θ), which maps the state space S to the action space A. The agent then observes an *immediate reward* $r_t \in \mathbb{R}$ given by a reward function $r(s_t, a_t)$; the immediate reward represents the loss/gain in transitioning from s_t to s_{t+1} because of action a_t. The agent's goal is

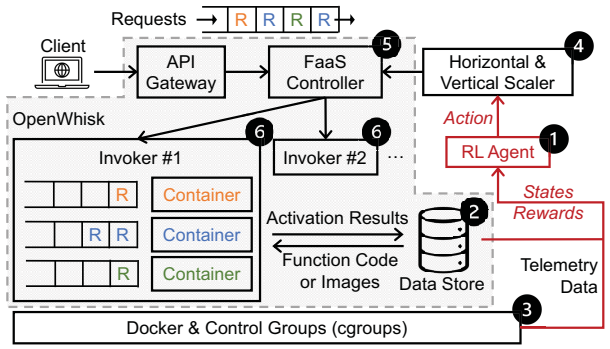


Figure 1. Resource management in OpenWhisk [10] as a sequential decision-making process. At each step, the RL agent perceives system and application conditions from the environment. The measurements are then translated to state and reward signals that mapped by the agent to an action.

to optimize the policy π_θ so as to maximize the expected *cumulative discounted reward* $\mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t]$ starting from a certain initial state s_0, where the expectation is taken over the "randomness" of state transitions and the agent's possibly randomized policy. The discount factor $\gamma \in (0, 1)$ penalizes the rewards far in the future. Two main categories of approaches are proposed for RL training: value-based methods and policy-based methods [2]. In value-based methods, the agent learns an estimate of the optimal value function and approaches the optimal policy by maximizing it. In policy-based methods, the agent tries to directly approximate the optimal policy. We refer the readers to [2, 25, 36, 37] for detailed surveys and rigorous derivations of value-based and policy-based RL algorithms.

Problem Formulation. We model the resource management in a serverless FaaS platform as a sequential decision-making problem that can be solved by the RL framework (illustrated in Fig. 1). Since all serverless FaaS platforms have similar controller-worker architectures and function request serving workflows, for the modeling purpose, we chose to use an open-source serverless FaaS platform, OpenWhisk [10]. At each step in the sequence, the RL agent (labeled as ❶) monitors system and application conditions from both the OpenWhisk data store (labeled as ❷) and the Linux cgroups (labeled as ❸). Measurements include function-level performance statistics (i.e., tail latencies on execution time, waiting time, and cold-start time for serving function requests) and system-level resource utilization statistics (e.g., CPU utilization of function containers). These measured telemetry data are pre-processed and used to define a state, which is then mapped to a resource management decision by the RL agent. In this model, we consider both vertical and horizontal resource scaling actions. A vertical-scaling action in OpenWhisk corresponds to scale either up or down the `cpu.shares` or the memory limit of a function container. A horizontal-scaling action in OpenWhisk corresponds to scale either out or in the function containers, i.e., changing the number of created containers for a function. The decision made by the RL agent is then passed by the horizontal and vertical scaler (labeled as ❹) to the FaaS controller (labeled as ❺) and finally changes the system state and function performance.

Table 1. State-action space of the RL formulation.

State Space S_t (for both single- and multi-agent)
SLO Preservation Ratio ($SP(t)$), Resource Utilization ($RU_{cpu}(t)$, $RU_{mem}(t)$), Arrival Rate Changes ($AC(t)$), Resource Limits ($RLT_{cpu}(t)$, $RLT_{mem}(t)$), Horizontal Concurrency ($NC(t)$)
Action Space A_t (for both single- and multi-agent)
Vertical Scaling: Resource Limits ($RLT_{cpu}(t), RLT_{mem}(t)$) Horizontal Scaling: Number of Containers ($NC(t)$)
Auxiliary Global State Space G_t (for multi-agent)
Aggregated Resource Limits ($ARLT_{cpu}(t), ARLT_{mem}(t)$), Aggregated Vertical Actions ($AV(t)$), Aggregated Horizontal Actions ($AH(t)$), Mean SLO Preservation Ratio ($MSP(t)$), Mean Resource Utilization ($MRU(t)$)

We carefully design the state-action space, state transition and reward function to satisfy the Markov property in the RL framework. Each function instance is deployed in a separate container with resource limits RLT_{cpu} and RLT_{mem} since OpenWhisk's default resource model includes `cpu.shares` and memory limits, both are configurable parameters in all commercial serverless platforms. The initial limit for each type of resource is overprovisioned before containers are created for a function and later controlled by the RL agent. The number of created containers is denoted as NC. We define the action space (as shown in Table 1) to be the available vertical scaling actions which changes RLT_{cpu}, RLT_{mem} and horizontal scaling actions which changes NC.

We define the state space based on the five features listed in Table 1. At each time step t, the utilization $RU(t)$ for each type of resource is retrieved from cgroups as telemetry data in ❸. The current resource allocation $RLT_{cpu}(t)$, $RLT_{mem}(t)$, and $NC(t)$ are kept as part of the state. In addition, OpenWhisk's data store (labeled as ❷) also collects function latency composition and request arrival rate. Based on these measurements, the RL agent calculates the remaining two states listed in Table 1 and described below:

- *SLO preservation ratio* ($SP(t)$) is defined as `latency_SLO / latency_measured` if there is an SLO violation. The ratio is smaller for more critical SLO violations. Otherwise, $SP(t)$ is set to 1, meaning that there is no SLO violation or no function request coming.
- *Arrival rate change* ($AC(t)$) is defined as $(AR(t) - AR(t-1))/max\{AR(t), AR(t-1)\}$, where $AR(t)$ and $AR(t-1)$ denote the function request arrival rates at the current and previous time steps, respectively. A positive value indicates an increasing arrival rate and vice versa.

All variables in the state vector are of range $[-1, 1]$ except $RLT(t)$ and $NC(t)$. To facilitate RL training, we normalized the two variables by setting a predefined resource upper limit $\hat{R}_i$ and a lower limit $\check{R}_i$. For instance, the `cpu.shares` for a container cannot be smaller than 128 or larger than 2048; and the number of containers cannot be smaller than

Table 2. Serverless benchmarks adopted from [21, 33].

Benchmark	Description
Base64	Encode and decode a string with the Base64 algorithm.
Primes	Find the list of prime numbers less than 10^7.
Markdown2HTML	Render a Base64 uploaded text string as HTML.
Sentiment-Analysis	Generate a sentiment analysis score for the input text.
Image-Resize	Resize the Base64-coded image with new sizes.

0 or larger than 1000 (the default maximum concurrency setting in AWS Lambda [1]). If the amount of resources to be vertically-scaled reaches the total available amount, then a horizontal scaling operation is needed.

The goal of the RL agent is, given a time duration T, to learn an optimal policy π_θ that results in as few SLO violations as possible (i.e., $\max_{\pi_\theta} \sum_{t=0}^{T} SP(t)$) while keeping the resource utilization as high as possible (i.e., $\max_{\pi_\theta} \sum_{t=0}^{T} RU(t)$). Based on both objectives, the reward function is then defined as $r_t = \alpha \cdot SP(t) \cdot |\mathcal{R}| + (1-\alpha) \cdot \sum_i^{|\mathcal{R}|} RU_i(t) + penalty$, where $\mathcal{R}$ is the set of resources and $penalty$ is set to -1 in the following cases: (a) Illegal actions such as scaling-in/up/down when the number of function containers is zero or scaling beyond the resource limits. Since illegal actions are not executable in the serverless FaaS platform, an illegal action leads to a self-loop transition from a state to itself with a negative reward. (b) Undesired actions such as frequent dangling decisions, which are detected by comparing the actions of the current and last time step.

Implementation with PPO. We use a policy gradient method, Proximal Policy Optimization (PPO) [20], to learn the optimal resource management policy under the RL problem formulation described above. PPO is the default RL algorithm at OpenAI [14] which performs comparably or better than state-of-the-art approaches while being much simpler to tune. It is also hypothesized that the smooth policy updates (due to clipping) in PPO can help mitigate the non-stationarity issue in multi-agent RL [6]. The algorithm and implementation details are left to Appendix A.

4 Experimental Methodology

OpenWhisk Cluster Setup. We deploy OpenWhisk [10] on five physical nodes in our local cluster with one master node (which runs the FaaS Controller) and four worker nodes (each of which runs an Invoker), as shown in Fig. 1. Each node has a dual-socket Intel Xeon E5-2683 v3 processor with 14 cores per socket and 500 GB memory. All nodes run Ubuntu 18.04.3 LTS with Linux kernel version 4.15. Memory swapping is disabled for the Docker service. We run the workload generator [21] and the RL controller from two separate nodes in the same cluster and use FaaSProfiler [21] to trace requests to measure the end-to-end latency.

Serverless Benchmarks. The benchmarks used in this study (listed in Table 2) are from widely used open-source FaaS benchmark suites [21, 33]. They include both micro-benchmarks and macrobenchmarks which have different runtime behaviors and resource demands (e.g., CPU utilization and memory bandwidth utilization). The functions are written in either Python or Java.

Workloads. We evaluated using both real-world and synthetic function invocation patterns. For real-world workloads, we sampled and replayed the function invocations from Azure function traces [22] from 2019 collected over two weeks. For synthetic workloads, we used common patterns [7] indicating flat and fluctuating loads, with a Poisson inter-arrival rate ranging from zero to the maximum observed in the sampled Azure function traces. The change of arrival rates is intended to evaluate whether RL agents could adapt to such workload change.

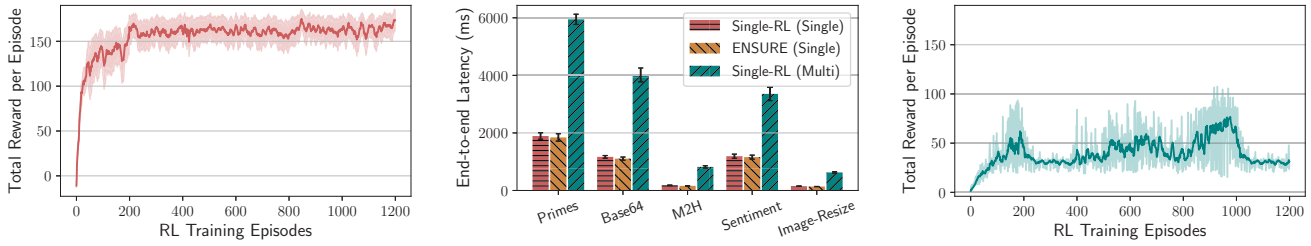
5 The Multi-tenancy Challenge

5.1 Single-RL in Single-tenant Environments

We implemented the RL prototype described in §3, conducted training convergence analysis, and evaluated the function SLO preservation and resource utilization with a state-of-the-art heuristics-based approach ENSURE [24] as the baseline. Since each RL agent in this approach manages resources for one specific function, we call it *single-agent RL* or *S-RL*.

Convergence Analysis. RL training proceeds in episodes (iterations). To understand the convergence behavior of the S-RL agent in single-tenant environments, we trained the agent by using the workload described in §4 without any other functions running on the platform. We then analyzed the per-episode reward evolution and Fig. 2(a) shows the results. We found that the agent training progress is similar across different function benchmarks so we chose to show the RL agent for function `primes`. We fixed the number of time steps in each training episode to be 200, except that for the initial stage of the training process, we terminated the RL exploration early so that the agent could reset and try again from the initial state. We did so because the initial policies of the agent were unable to mitigate SLO violations. As the training progressed, the agent improved its resource allocation policy and could mitigate SLO violations in less time. At that point (around 70 episodes), we linearly increased the number of time steps to let the agent interact with the environment for a longer time before terminating the exploration and entering the next iteration. The agent's behavior was able to converge after around 300 episodes (ranging from 280 to 350 across functions).

Performance Assessment. After convergence, we leveraged the trained S-RL agent (by using the saved checkpoints at episode 1000) as the function resource controller and compared it with the baseline approach ENSURE [24], which is a state-of-the-art threshold-based autoscaler implemented based on OpenWhisk as well. In the baseline approach, we set the parameters and thresholds the same as the paper specified. Fig. 2(b) shows the online performance comparison. S-RL agent was able to keep the CPU utilization at a higher



(a) Training curve of the single-agent RL (for function `primes`) in single-tenant environments.

(b) Single-agent RL 99th-% end-to-end latency comparison for all function benchmarks.

(c) Training curve of the single-agent RL (for function `primes`) in multi-tenant environments.

Figure 2. Training convergence analysis and online performance evaluation of S-RL in single- and multi-tenant environments.

level (around 20% higher than ENSURE) and achieved similar end-to-end latency compared to the baseline approach. We found that it is because ENSURE over-provisioned containers and resources in some cases. After increasing the performance SLO threshold to a higher value (namely from 15% to 25%), we observed that the S-RL agent significantly improved over the baseline approach regarding tail latencies at similar CPU utilizations (not shown in the figure).

By interacting with dynamic FaaS environments under complicated loads and resource allocation scenarios, the S-RL agent dynamically learns the policy that maximizes the cumulative rewards and hence outperforms heuristics-based approaches.

5.2 Single-RL in Multi-tenant Environments

Serverless FaaS platforms are in essence multi-tenant where different function owners deploy and run heterogeneous functions with various function characteristics, SLOs, and workload patterns [3, 9, 17, 27]. Each function controlled by an RL agent competes with other functions on the same platform for limited resources. Function container co-location for higher utilization has made resource contention worse on a cloud serverless platform [17, 21, 27]. The transition from single-tenant to multi-tenant settings introduces new challenges that require a fundamentally different RL algorithm design. Before presenting our multi-agent solution, we first explored the environment non-stationarity issue, and conducted training convergence analysis and performance assessment of S-RL agents in multi-tenant environments.

Environment Non-stationarity. The shared environment in a multi-agent setting can be affected by the actions of all agents; thus, from a single agent's perspective, the environment becomes non-stationary, which breaks the critical stationarity assumption made by most RL algorithms [25]. Under non-stationarity, an agent needs to explore the unknown environment efficiently while keeping in mind that the information it gathers now will soon become outdated, because the other agents are also updating their policies. A naive approach to tackle the non-stationarity issue is to use centralized learning, e.g., joint action learners (JAL) [5]. In such a centralized approach, the agents are jointly modeled and a centralized policy for all the agents is trained. The input to this algorithm is the concatenation of the observations of all the agents, and the output is the actions specified to the agents. This approach eliminates the problem of non-stationarity entirely; however, it is computationally inefficient (with exponential complexity) because the centralized learner needs to search in the joint action space of the size $\prod_{i=1}^{N} |A_i|$ in order to enumerate all possible action combinations, where N is the number of agents and A_i is the individual action space of agent i for $1 \leq i \leq N$. The exponential dependence on N makes the centralized learning approach difficult to scale up beyond a few agents [5]. Therefore, given that there could be tens or even hundreds of functions on a server [26], we do not proceed further with this centralized approach.

Performance Degradation. We conducted a performance assessment of the S-RL agent in multi-tenant cases where each function is in control of an independent S-RL agent trained in isolation. Fig. 2(b) shows the performance degradation after introducing multiple tenants on the same serverless FaaS platform. In this experiment, we created one function for each benchmark from Table 2 and trained one S-RL agent for each function in isolation until convergence. Then, we ran all five functions simultaneously with each function controlled by its trained S-RL agent. Evaluation results show that the degradation can be up to 78.2% (for `markdown2html`) and as low as 64.5% (for `sentiment-analysis`). That is because when the RL agent made a decision, it was based on the state information measured at the current time step; but at the same time, all other agents were also making their resource allocation decisions which can affect the shared environment. Therefore, the estimated value function for an action by the S-RL model is no longer accurate in a non-stationary multi-tenant environment.

Convergence Failure. Multi-tenancy not only affects the online performance of S-RL agents trained in isolation but also leads to problems during training. We trained five S-RL agents together in the same serverless FaaS platform, each of which controlled one function from the benchmarks listed in Table 2. Each S-RL agent was trained independently and did not consider the other agents in the same environment. Everything else was kept the same with the S-RL agent trained

in isolation. Fig. 2(c) shows the per-episode reward evolution for the agent controlling function `primes`. It would be expected for the S-RL agent to adapt to the stochasticity introduced by other agents. However, compared to the learning curve of the S-RL agent trained in isolation (as shown in Fig. 2(a)), the S-RL agent trained in a multi-tenant environment achieved lower performance (a 55.9% drop in terms of per-episode reward) with higher variance and did not converge in a stable manner. We observed similar behaviors from the training curves of the other agents across different function benchmarks (not shown due to page limit).

System support for many-agent RL controllers that provides both training convergence and performance isolation is needed.

6 Multi-agent Reinforcement Learning

To tackle the non-stationarity issue for S-RL agents in multi-tenant serverless FaaS platforms, we first present the re-modeled resource management problem as a multi-agent extension of the Markov decision process (MDP) in §6.1 and then designed a customized multi-agent RL algorithm *multi-agent PPO* (or *MA-PPO* [1]) for this problem (§6.2). Our design choices were made to favor the scalability and adaptivity (to agent churn for added/removed functions) of the multi-agent model. Evaluation shows that MA-PPO enables the convergence behavior of the agents and provides online performance comparable to S-RL in single-tenant cases (§6.3).

6.1 Multi-agent RL (MARL) Formulation

We extended the MDP formulation for single-agent RL (§3) to a Markov game (also known as stochastic game [23]) for N agents, each of which controls the resource management for one particular function. In our formulated Markov game, the state space is defined as the Cartesian product of the state spaces of all S-RL agents (as defined in §3). After observing the environment state s_t at time t, each agent i takes an action a_t^i based on its policy π_{θ^i} (parameterized by θ^i), and receives a reward r_t^i. The environment state then transitions to a new state depending on the joint action of all the agents. When we view the MARL setup from the point of view of a particular agent i, all the other agents are part of the environment. Each agent extracts its local information l_t^i and auxiliary global information g_t^i from the environment state s_t. The local information l_t^i of agent i is from the same state space in S-RL (as listed at the first row in Table 1). The action and reward of each agent i are also the same as defined in S-RL. We use the auxiliary global information g_t^i to describe the status of all the other agents (except for agent i) and it consists of the following variables (as listed at the third row in Table 1):

- *Aggregated resource limits*: $ARLT^i_{cpu}(t) = \sum_{j\neq i}^{N} RLT^j_{cpu}(t)$, $ARLT^i_{mem}(t) = \sum_{j\neq i}^{N} RLT^j_{mem}(t)$
- *Aggregated vertical actions*: $AV^i(t) = \sum_{j\neq i}^{N} \Delta RLT(t)$
- *Aggregated horizontal actions*: $AH^i(t) = \sum_{j\neq i}^{N} \Delta NC^j(t)$
- *Mean SLO preservation ratio*: $MSP^i(t) = \frac{\sum_{j\neq i}^{N} SP^j(t)}{(N-1)}$
- *Mean resource utilization*: $MRU^i(t) = \frac{\sum_{j\neq i}^{N} RU^j(t)}{N-1}$

Using aggregated and mean values across all the other agents allows the MARL model to be agnostic to agent order or the amount of agents. This is especially important in a serverless environment where tenants can join and leave from time to time, creating and removing their functions at any time. However, in a centralized MARL approach that explicitly models each function or agent (e.g., the JAL [5] mentioned in §5.1), the whole MARL algorithm needs to be retrained because the input to the algorithm has been changed. In the typical case where the policy or value functions are parameterized by neural networks, the network structure would also need to be reconstructed. In contrast, the agent group order- and size-agnostic MARL formulation is more scalable and enables fast-retraining.

The goal in the MARL setting is (see [36, 37]), given a time duration T, to determine an optimal collection of policies $\pi = \{\pi_{\theta^1}, \pi_{\theta^2}, ..., \pi_{\theta^N}\}$ that results in fewer SLO violations across all functions (i.e., $\max_{\theta^1,\theta^2,...,\theta^N} \sum_{i=1}^{N}\sum_{t=0}^{T} SP^i(t)$) while keeping the resource utilization as high as possible (i.e., $\max_{\theta^1,\theta^2,...,\theta^N} \sum_{i=1}^{N}\sum_{t=0}^{T} RU^i(t)$). The team-averaged reward function for the MARL setting is then defined as $r_t = \sum_{i=1}^{N} r_t^i/N = \sum_{i=1}^{N}(\alpha \cdot SP^i(t) \cdot |\mathcal{R}| + (1-\alpha) \cdot \sum_j^{|\mathcal{R}|} RU_j^i(t) + penalty^i)/N$, where $\mathcal{R}$ is the set of resources and $penalty$ is the same as defined in §3. Our objective is to maximize the expected total return $\mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t] = \mathbb{E}[\sum_{t=0}^{T} \gamma^t \cdot \sum_{i=1}^{N} r_t^i/N]$, where $\gamma \in (0, 1)$ is the discount factor.

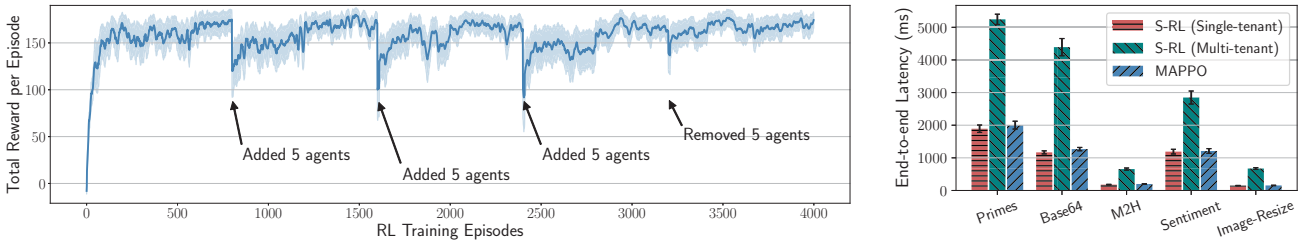
6.2 Multi-agent PPO

Based on the PPO algorithm for S-RL described in Appendix A, we introduce a multi-agent version of PPO (MA-PPO) in this section. MA-PPO follows the same algorithmic structure of the PPO algorithm by learning a policy π_{θ^i} and a value network V_{ϕ^i} (parameterized by ϕ^i) for each agent i. We concatenate the auxiliary global information g^i (described in §6.1) to each agent's local information l^i to feed it as input to the value network V_ϕ^i. The value network then outputs a single number representing the estimated state value. The policy network π_{θ^i} is the same as the policy network in PPO, which takes the states as inputs and outputs an action from the same action space. The other extension from PPO is that the reward for each agent at time t is changed to the total reward across all agents: $r_t = \sum_{i=1}^{N} r_t^i$. The hyperparameters are kept the same. We provide the pseudocode of MA-PPO algorithm in Appendix B.2.

6.3 Preliminary Results

In this sub-section, we present evaluation results for MA-PPO training and its online performance. During the training of MA-PPO agents, we intentionally added and removed a few

[1] Note that MA-PPO is a novel multi-agent RL algorithm and is not to be confused with MAPPO [31], which is a different algorithm that happens to share a similar acronym. In particular, MAPPO [31] focuses on the fully cooperative setting while serverless functions are not.



(a) Training curve of the multi-agent RL in multi-tenant environments (starting from 5 agents).

(b) Multi-agent RL 99th-% end-to-end latency comparison for all function benchmarks.

Figure 3. Training convergence analysis and online performance evaluation of multi-agent RL (MA-PPO).

agents from the environment to evaluate the adaptability of the MA-PPO model to agent updates. Fig. 3(a) shows the training curves of MA-PPO in multi-tenant environments. To start with, we created five functions (one from each benchmark in Table 2), and each function is then controlled by an initialized MA-PPO agent. Since all agents use the team reward which is the average reward across all agents, Fig. 3(a) shows the evolution of the average total reward per episode. The MA-PPO agents were able to reach a stable converged policy after around 500 episodes. Then at episode 800, we updated the multi-tenant environment by adding five functions (one from each benchmark), each of which is controlled by a different MA-PPO agent. As noticed in the figure, the total reward per episode dropped to around 80 and that was mainly because the added five new MA-PPO agents were learning the optimal policy which led to low reward values. After around 300 more episodes (around 2.5 hours), the learning curve of the MA-PPO agents was able to converge again. We updated the environment three more times after every 800 episodes by either adding five new functions or removing five existing functions. We observed a similar reward drop and later convergence to about the same level after several hundreds of training episodes. When we removed five existing functions from the environment, the reward drop (i.e., around 125) was not as much as the previous cases. We attribute the smaller reward drop to the fact that there was no added agent whose reward starts to be randomly lower than a trained agent. The team reward still dropped due to the fluctuation of the environment as there were five newly added functions.

We saved the checkpoints at the 4000th episode for all MA-PPO agents and used those checkpoints to evaluate the online performance for each of the five functions together on a multi-tenant serverless FaaS platform. We observed that the function performance was similar at different episodes when the agents' behavior converged so we picked the 4000th episode. At the 4000th episode, there were 15 functions in total (i.e., three functions from each of the five benchmarks) Fig. 3(b) shows the performance comparison between MA-PPO-controlled functions and the single-RL trained in a multi-tenant environment. We averaged over all three functions of the same type of benchmark. As shown in the figure, MA-PPO was able to provide online performance comparable to single-RL in single-tenant cases, with the performance degradation ranging from 1.8% (for `sentiment-analysis`, 1190.2 ms to 1211.5 ms) to 9.9% (for `markdown2html`, 178.4 ms to 198.1 ms). Compared to the single-RL trained in multi-tenant environments (as shown in §5.1 and Fig. 2(c)), the MA-PPO achieves 2.5× (for `sentiment-analysis`, 1211.5 ms to 3047.8 ms) to 4.4× (for `image-resize`, 154.2 ms to 672.4 ms) improvement in terms of the 99th-percentile latency.

7 Conclusion and Future Steps

We presented our early work on a multi-agent framework to support RL-based resource management controllers in serverless FaaS platforms. We highlighted the multi-tenancy challenges that single-agent RL faces and proposed a scalable and validated MARL algorithm which resolves the training convergence problem while providing online performance comparable to single-RL in single-tenant scenarios. Nevertheless, a few challenges and potential improvement are left for future work, including fast retraining for newly joined or updated functions with network parameter sharing or transfer learning and fault tolerance to agent disconnection or RL transition corruption.

Acknowledgments

We thank the anonymous reviewers for their valuable comments that improved the paper. This work is partially supported by the National Science Foundation (NSF) under grant No. CCF 20-29049; by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR), a research collaboration that is part of the IBM AI Horizon Network; and by the IBM-ILLINOIS Discovery Accelerator Institute (IIDAI). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or IBM.

A Proximal Policy Optimization (PPO)

We use a policy-based method, PPO [20], to learn the optimal resource management policy under the MDP formulation described in §3. Compared to the vanilla policy gradient, PPO guarantees an

Table 3. RL training hyperparameters.

Parameter	Value
Learning Rate	Actor (3×10^{-4}), Critic (3×10^{-4})
Discount Factor (γ)	0.99
Number of Hidden Layers	Actor (2), Critic (2)
Number of Hidden Units	Actor (64), Critic (64)
Mini-batch Size	5
Number of SGD Epochs	5
Clip Value (ϵ)	0.2
Entropy Coefficient (β)	0.01
Critic Loss Discount (δ)	0.05
Number of Time Steps	200 (per Episode)
Reward Coefficient (α)	0.3

improved policy by specialized clipping in the objective function to prevent the new policy from getting far from the old policy. It does this by defining a probability ratio $p_t(\theta)$:

$$p_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{1}$$

This ratio denotes the change the policy has gone through within an episode of training. $\pi_{\theta_{old}}$ is the policy at the beginning of the update. The clipped surrogate objective function $J^{CLIP}(\theta)$ is then maximized to update the policy:

$$J^{CLIP}(\theta) = \mathbb{E}[min(clip(p_t(\theta), 1-\epsilon, 1+\epsilon)A_t, p_t(\theta)A_t)] \tag{2}$$

where ϵ is a hyperparameter specific to PPO and A_t is the estimator of the advantage function at t [20]. Taking the minimum of the clipped and unclipped terms results in the objective only being clipped if its value is improved by the new policy.

PPO has a critic component that estimates the value function V_ϕ (parameterized by ϕ) under a certain policy. The critic network receives the state as the input and outputs a single number representing the estimated value of that state. To update the critic network, we seek to minimize the difference between the estimated value and the actual return. Using the squared loss, we get another objective function:

$$J^{VF}(\phi) = -(V_\phi - g)^2, \tag{3}$$

where $g = \sum_{t=0}^{T} \gamma^t r_t$ is the actual cumulative discounted reward of running the policy being evaluated.

Finally, we added an entropy regularization term (i.e., $H(p) = -\sum_{i=1}^{n} p_i \log p_i$ for a probability distribution p in an n-dimensional simplex) of $\pi_\theta(s_t)$ multiplied by a factor β to the objective to discourage premature convergence [28]. While this is not mandatory, we found that it increased performance significantly. The final objective function is:

$$J(\theta, \phi) = J^{CLIP}(\theta) + \delta J^{VF}(\phi) + \beta H(\pi_\theta) \tag{4}$$

The ϵ, δ and β values in our experiments along with all the other hyperparameters are listed in Table 3. We provide the pseudocode of PPO algorithm in Appendix B.1.

B Pseudocode

We provide the pseudocode of PPO and MA-PPO used for our single-agent RL and multi-agent RL algorithms in this section.

Algorithm 1 PPO with Clipped Objective

Randomly initialize network parameters θ_0, ϕ_0.
for iteration $k = 0, 1, 2, \ldots, M$ **do**
  Run policy π_{θ_k} for T time steps.
  Estimate advantages A_t at all time steps t.
  Optimize the objective J as in Eq. 4 with respect to θ and ϕ using K steps of minibatch SGD (via Adam), and obtain the new parameters θ_{k+1} and ϕ_{k+1}.
end for

Algorithm 2 MA-PPO with Augmented State Space

Randomly initialize network parameters θ_0^i, ϕ_0^i for all i.
for iteration $k = 0, 1, 2, \ldots, M$ **do**
  for each agent i **do** (in parallel)
    Run policy $\pi_{\theta_k}^i$ for T time steps.
    Calculate augmented V_t^i
    Estimate advantages ${A_t}^i$ at all time steps t.
  end for
  for each agent i **do** (in parallel)
    Optimize the objective J as in Eq. 4 with respect to θ^i and ϕ^i using K steps of minibatch SGD (via Adam), and obtain the new parameters θ_{k+1}^i and ϕ_{k+1}^i.
  end for
end for

B.1 PPO

PPO [20] trains a stochastic policy in an on-policy way by sampling actions according to the latest version of its stochastic policy. The algorithm that uses fixed-length trajectory segments is shown below. In each iteration, the agent collects T time steps of data and then the surrogate loss on these time steps of data is calculated and optimized with minibatch SGD for K epochs. The pseudocode is described in Alg. 1.

B.2 Multi-agent PPO

MA-PPO follows the algorithmic structure of the PPO algorithm by learning a policy network and a value function for each agent. We extended the input vector to the value function with the auxiliary global states from the environment and used the team reward as the reward to each agent. Alg. 2 described the pseudocode of the MA-PPO algorithm.

References

[1] Amazon. 2022. AWS Lambda concurrency limit. https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html. Accessed: 2022-01-10.

[2] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866* (2017).

[3] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 1–20.

[4] Subho Banerjee, Saurabh Jha, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2020. Inductive-bias-driven Reinforcement Learning For Efficient Schedules in Heterogeneous Clusters. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 629–641. https://proceedings.mlr.press/v119/banerjee20a.html

[5] Caroline Claus and Craig Boutilier. 1998. The dynamics of reinforcement learning in cooperative multiagent systems. *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI)* 1998, 746-752 (1998), 2.

[6] Christian Schroeder de Witt, Tarun Gupta, Denys Makoviichuk, Viktor Makoviychuk, Philip HS Torr, Mingfei Sun, and Shimon Whiteson. 2020. Is Independent Learning All You Need in the StarCraft Multi-Agent Challenge? *arXiv preprint arXiv:2011.09533* (2020).

[7] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.

[8] Yisel Garí, David A Monge, Elina Pacini, Cristian Mateos, and Carlos García Garino. 2021. Reinforcement learning-based application autoscaling in the cloud: A survey. *Engineering Applications of Artificial Intelligence* 102 (2021), 104288.

[9] Samuel Ginzburg and Michael J Freedman. 2020. Serverless Isn't Server-Less: Measuring and Exploiting Resource Variability on Cloud FaaS Platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. 43–48.

[10] Github. 2022. Apache OpenWhisk. https://github.com/apache/openwhisk. Accessed: 2022-01-10.

[11] Sara Kardani-Moghaddam, Rajkumar Buyya, and Kotagiri Ramamohanarao. 2020. ADRL: A Hybrid Anomaly-Aware Deep Reinforcement Learning-Based Resource Scaling in Clouds. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 514–526.

[12] 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 (HotNet 2016)*. 50–56.

[13] Monaldo Mastrolilli and Ola Svensson. 2008. (Acyclic) job shops are hard to approximate. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 583–592.

[14] OpenAI. 2022. OpenAI Baselines: Proximal Policy Optimization. https://openai.com/blog/openai-baselines-ppo/. Accessed: 2022-01-10.

[15] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. GRAF: A graph neural network based proactive resource allocation framework for SLO-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking Experiments and Technologies (CoNext 2021)*. 154–167.

[16] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020)*. 805–825.

[17] Haoran Qiu, Saurabh Jha, Subho S Banerjee, Archit Patke, Chen Wang, Franke Hubertus, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2021. Is Function-as-a-Service a Good Fit for Latency-Critical Services?. In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*. 1–8.

[18] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: Workload Autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys 2020)*. Association for Computing Machinery, New York, NY, USA, Article 16, 16 pages. https://doi.org/10.1145/3342195.3387524

[19] Lucia Schuler, Somaya Jamil, and Niklas Kühl. 2021. AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid 2021)*. IEEE, 804–811.

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

[21] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of Function-as-a-Service computing. In *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO 2019)*. 1063–1075.

[22] Mohammad Shahrad, Rodrigo Fonseca, Íñigo 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*. 205–218.

[23] Lloyd S Shapley. 1953. Stochastic games. *Proceedings of the National Academy of Sciences* 39, 10 (1953), 1095–1100.

[24] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. 2020. ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments. In *International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS 2020)*. 1–10.

[25] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT Press.

[26] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. 559–572.

[27] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference*. 133–146.

[28] Ronald J Williams and Jing Peng. 1991. Function optimization using connectionist reinforcement learning algorithms. *Connection Science* 3, 3 (1991), 241–268.

[29] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. 2019. MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *2019 IEEE 39th international conference on distributed computing systems (ICDCS)*. IEEE, 122–132.

[30] Zhang Yanqi, Hua Weizhe, Zhou Zhuangzhuang, Suh G. Edward, and Delimitrou Christina. 2021. Sinan: ML-Based & QoS-Aware Resource Management for Cloud Microservices. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*.

[31] Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre Bayen, and Yi Wu. 2021. The Surprising Effectiveness of PPO in Cooperative, Multi-Agent Games. *arXiv preprint arXiv:2103.01955* (2021).

[32] Hanfei Yu, Athirai A Irissappane, Hao Wang, and Wes J Lloyd. 2021. FaaSRank: Learning to Schedule Functions in Serverless Platforms. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS 2021)*. IEEE, 31–40.

[33] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. ServerlessBench (SoCC 2020). https://github.com/SJTU-IPADS/ServerlessBench.

[34] Anastasios Zafeiropoulos, Eleni Fotopoulou, Nikos Filinis, and Symeon Papavassiliou. 2022. Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms. *Simulation Modelling Practice and Theory* (2022), 102461.

[35] 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 2019)*. 1049–1062.

[36] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. 2021. Decentralized multi-agent reinforcement learning with networked agents: Recent advances. *Frontiers of Information Technology & Electronic Engineering* 22, 6 (2021), 802–814.

[37] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. 2021. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *Handbook of Reinforcement Learning and Control* (2021), 321–384.