SIMPPO: A Scalable and Incremental Online Learning Framework for Serverless Resource Management

Haoran Qiu
UIUC

Weichao Mao
UIUC

Archit Patke
UIUC

Chen Wang
IBM Research

Hubertus Franke
IBM Research

Zbigniew T. Kalbarczyk
UIUC

Tamer Başar
UIUC

Ravishankar K. Iyer
UIUC

Abstract

Serverless Function-as-a-Service (FaaS) offers improved programmability for customers, yet it is not server-"less" and comes at the cost of more complex infrastructure management (e.g., resource provisioning and scheduling) for cloud providers. To maintain service-level objectives (SLOs) and improve resource utilization efficiency, recent research has been focused on applying online learning algorithms such as reinforcement learning (RL) to manage resources. Despite the initial success of applying RL, we first show in this paper that the state-of-the-art single-agent RL algorithm (S-RL) suffers up to 4.8× higher p99 function latency degradation on multi-tenant serverless FaaS platforms compared to isolated environments and is unable to converge during training. We then design and implement a scalable and incremental multi-agent RL framework based on Proximal Policy Optimization (SIMPPO). Our experiments demonstrate that in multi-tenant environments, SIMPPO enables each RL agent to efficiently converge during training and provides online function latency performance comparable to that of S-RL trained in isolation with minor degradation (<9.2%). In addition, SIMPPO reduces the p99 function latency by 4.5× compared to S-RL in multi-tenant cases.

CCS Concepts

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

Keywords

Serverless computing, 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. SIMPPO: A Scalable and Incremental Online Learning Framework for Serverless Resource Management. In *SoCC '22: ACM Symposium on Cloud Computing (SoCC '22), November 7–11, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3542929.3563475

1 Introduction

In serverless Function-as-a-Service (FaaS) [4, 53, 66], cloud providers handle resource management for each function (e.g., scaling the number of function containers or their resource limits, and scheduling containers to servers) but no commercial cloud provider provides any performance service-level objectives (SLOs) [44]. Providing performance SLOs has been studied in various aspects and is critical to run latency-critical services on serverless platforms [24, 44, 55, 56, 68, 71]. The problem of managing resources to achieve performance SLOs while maintaining high resource utilization is at its core an intractable NP-hard problem [5, 37]. While the majority of the associated 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) [5, 20, 28, 34, 35, 41, 43, 45, 49, 64, 67, 72, 75, 77].

As a viable alternative to human-generated heuristics, RL enables an artificial agent to learn the optimal *policy* directly from interaction with the *environment* by observing its *state* and selecting an *action* from the policy. As a result, the environment transitions to the next state, and the agent receives feedback in the form of *rewards.* The goal is to take a sequence of actions that maximizes the expected cumulative rewards in the future. As learning continues, the agent can optimize for a specific workload and adapt to varying conditions (i.e., the *policy-training* stage). After convergence, the learned

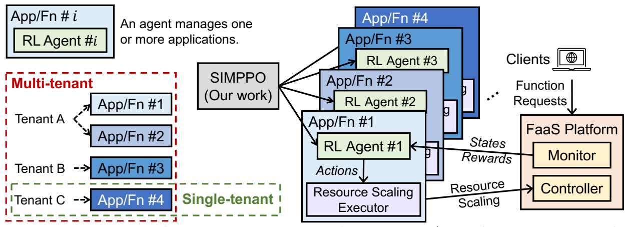

Figure 1: Single-agent RL solutions (each agent independently trained and unaware of each other) in single- or multi-tenant serverless environments. SIMPPO is a multi-agent framework managing all RL agents.

policy will continue being used by the agent to interact with the environment (i.e., the *policy-serving* stage) [40].

Motivation. Despite recent successes, existing RL-based solutions are all single-agent RL (S-RL) in which one agent manages one or more applications or functions[1] (as shown in Fig. 1), paying no attention to any other agents. The standard assumption for an S-RL algorithm is *environment stationarity* [42, 57, 69], which means that the environment is affected only by the agent interacting with it. Therefore, existing S-RL solutions assume that the agent is in an isolated single-tenant environment that contains only the application that the agent manages. In contrast, a serverless FaaS platform in any cloud data center is multi-tenant, and functions from all customers compete for shared resources in a cluster. Multi-tenancy makes the environment *non-stationary* from each agent's own perspective when all agents are jointly being trained. At the training stage, since the state transitions and rewards each agent gets depend on the joint actions of all agents whose policies keep changing in the learning process, each agent enters an endless cycle of adapting to other agents. At the policy-serving stage, an action might be suboptimal when applied, because the underlying environment is no longer the same as the one previously perceived for generating the action. Two examples in §2.4 demonstrate such undesirable behaviors caused by environment non-stationarity.

Challenges. We propose a multi-agent RL (MARL) solution that allows multiple agents to coexist in a shared environment. However, solving the training non-convergence problem and achieving policy-serving performance comparable to the performance obtained in single-tenant scenarios further present two main challenges.

a) *Scalability*: The solution should scale to the large number of functions in a multi-tenant serverless platform. Existing MARL approaches, such as centralized MARL [9, 10] or decentralized MARL with networked agents [78, 79], all suffer scalability issues as the computation complexity of searching in the joint state-action space grows exponentially with the number of agents.

[1]S-RL-based approaches are application-specific or function-specific.

b) *Adaptive and incremental training*: The proposed solution should adapt to dynamic changes in the environment due to function churns (i.e., add, remove, or update functions) or variations in the number of agents [8, 80]. Existing MARL approaches model agents jointly by optimizing over the Cartesian product of all the agents' action spaces. Consequently, they are computationally inefficient (with exponential complexity), and the whole MARL algorithm must be repeatedly retrained when the number of agents in the environment changes.

Our Work. We design and implement SIMPPO, a **S**calable and **I**ncremental **M**ulti-agent RL framework based on an RL algorithm, **P**roximal **P**olicy **O**ptimization (PPO) [50]. As illustrated in Fig. 1, each agent in SIMPPO still has its own policy trained using PPO and manages an individual function. We use PPO because it provides more stable policy learning, shorter training times, and higher rewards after convergence compared to other RL algorithms (e.g., DDPG [32], DQN [62]). SIMPPO manages all agents to dynamically and continuously maintain the SLO of each function while keeping high utilization efficiency. Each SLO is associated with a function instance specifying a latency threshold for users sending requests to the function instance. In contrast to S-RL, in which each agent is unaware of the other agents and trained in isolation, all agents in SIMPPO are jointly trained to reach convergence. Each agent is allowed to peek into the behavior of the other agents, and changes in other agents' behavior (policy) are handled during training.

To address the incremental training challenge, we designed the MARL model and the neural network architecture of each agent such that all other agents are treated as part of the environment; thus, the model is agnostic to agent sequence order or the number of agents. From each agent's perspective, we created a *virtual* agent that consists of the environment and all the other agents. The many-agent problem is converted to a two-agent problem, so there is no need to reconstruct the neural network, whose structure remains unchanged. In contrast, retraining from scratch for each agent would have been prohibitively costly in both time and resources. Finally, to further shorten the incremental training time, we leveraged neural network parameter sharing [58, 81] between new agents and existing agents that manage the same type of functions.

To address the scalability challenge, we modeled and approximated the collective behavior of the other agents (i.e., the virtual agent) via an *auxiliary global state* distribution. We constructed auxiliary global states by selecting each agent's relevant state and action variables based on domain knowledge and feature engineering to help learn an accurate estimation of the virtual agent. We took the aggregated actions and resource limits from all the other agents to represent the collective resource allocation since we viewed them as part of the environment. We also took the average function performance and resource utilization to indicate how the virtual agent behaves, as the goal is to achieve function SLO performance while maintaining high resource utilization. We removed redundant features that negatively affected RL training because it is hard for the neural network to learn from its unnecessarily large set of inputs. Auxiliary global states are provided to each agent to help it adapt to varying agents in the environment by learning the collective and average behavior of the virtual agent instead of all the other individual agents. Reducing the interaction between one agent and all others to the interaction between one agent and the virtual agent greatly alleviates the scalability issue.

We design SIMPPO as a general framework to support a variety of RL agents that employ online learning algorithms. Although we use serverless resource management as an example for demonstration, SIMPPO can be potentially applied in other RL for systems areas [35] such as load balancing and congestion control. Our approach for using virtual agents draws inspiration from the mean-field theory, which has been successful in economics and physics [2, 70]. Existing theory [36, 46, 47] shows that approximating the collective behavior of all the agents using a single population distribution term such as the average does not lose much optimality in finite multi-agent scenarios, and the approximation error decreases when the number of agents increases.

Contributions. In summary, our main contributions are:

a) Single-agent RL formulation and design using the state-of-the-art RL algorithm PPO for serverless resource management (§3).
b) The first quantitative characterization study, to the best of our knowledge, demonstrating that single-agent RL is unable to converge and the policy-serving performance is severely degraded in multi-tenant serverless environments (§5.2).
c) The design of a scalable and incremental MARL framework named SIMPPO that (i) enables multiple RL agents to be trained jointly to convergence and coexist in a multi-tenant serverless environment, and (ii) allows the agents to adapt to dynamic changes in the system, e.g., adding or updating of functions (§6).
d) The implementation and comprehensive evaluation of SIMPPO with real-world workloads that demonstrate scalable and incremental policy training and substantial policy-serving improvements relative to single-agent RL solutions (§7).

Results. We conducted experiments in which we deployed an open-source serverless FaaS platform, OpenWhisk [18], on our dedicated local cluster and a larger cluster on IBM Cloud. We ran widely used serverless benchmarks [11, 52, 76] driven by arrival rate patterns sampled from both synthetic datasets and public production traces from Azure Functions [53]. We show in our characterization study (§5) that S-RL trained in isolation (which we use as the *baseline* to assess MARL solutions) (i) improves p99 function latency by at least 1.6× without over-provisioning over a heuristics-based solution in single-tenant cases, and (ii) suffers from 2.2–4.8× higher p99 function latency degradation in multi-tenant cases. An evaluation of SIMPPO shows that it enables all agents' policies to converge when they are jointly trained and that it supports incremental training (§7.1). The online policy-serving performance of SIMPPO (in terms of p99 function latency) in multi-tenant cases is comparable to the baseline, with <9.2% degradation (§7.2); SIMPPO achieves 4.5× improvement compared to S-RL in multi-tenant cases with reasonable resource overhead (§7.3). SIMPPO is also scalable to larger numbers of functions and cluster size (§7.4).

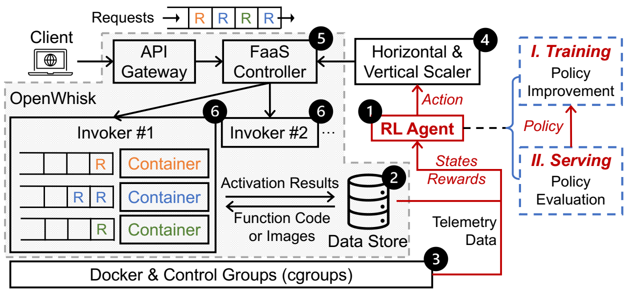

Figure 2: Resource management in OpenWhisk [18] with reinforcement learning (RL). At each step, the RL agent perceives system and application conditions from the environment. The measurements are then translated to *state* and *reward* signals that are mapped by the agent to an *action*.

2 Background and Motivation

2.1 Serverless Function-as-a-Service

Serverless FaaS is a cloud programming model and architecture wherein customers execute function code snippets without any control over the resources on which the code runs [4]. 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 [18], a production-grade open-source serverless platform based on Docker containers. Fig. 2 shows the architecture of a distributed OpenWhisk platform. The controller (i.e., 5) creates function containers, allocates CPU (`cpu.shares` is used in OpenWhisk [56, 60]) and RAM for each function container, and assigns the containers to invokers (i.e., 6). 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 (i.e., 2), which completes an *activation*.

Serverless FaaS workloads, like most cloud data-center services, have service-level objectives (SLOs) defined by each tenant that codify the expected performance [7, 12, 17, 19, 22, 27, 43, 73]. The most common type is a latency SLO, which specifies the acceptable latencies for function requests in the serverless computing context. For example, a latency SLO might specify that 99% of requests have latencies smaller than 100 ms. If a service fails to meet its SLOs, the service provider may risk severe penalties or financial loss. We focus on resource management to meet per-function SLOs while keeping the utilization at a high level, since low utilization efficiency is undesirable for the cloud provider [6, 16, 17, 23, 33]. The mechanism to convey SLO preferences is also necessary and Henge [26] allows SLOs to be specified by incorporating latency or throughput goals and workload priorities into a user-specifiable utility function.

2.2 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 (via `step() function` [40]). The whole sequence of transitions $\{(s_t, a_t, r_t, s_{t+1})\}_{t\geq 0}$ is called an *episode* (or *iteration*). During policy training, the agent's goal is to optimize its policy π_θ so as to maximize the expected *cumulative discounted reward* $\mathbb{E}[\sum_{t=0}^{T} \gamma^t r_t]$ (i.e., the value function) during one episode, starting from a certain initial state s_0, where the expectation is taken over the randomness of state transitions and the agent's policy. The discount factor $\gamma \in (0, 1)$ penalizes the rewards far in the future. After convergence, the learned policy will continue to be used (but not updated) by the agent to interact with the environment during the policy serving stage.

Two main categories of approaches have been proposed for RL training: value-based methods and policy-based methods [3]. Both the policy and the value function in RL algorithms are usually implemented using neural networks. We refer readers to [3, 57, 78, 79] for detailed surveys and rigorous derivations of the RL algorithms.

2.3 Related Work

Heuristics-based Resource Management. Various approaches have recently been proposed to address some of the existing challenges in serverless platforms, such as function request scheduling and resource allocation, using carefully designed heuristics [25, 38, 55, 56, 59, 68, 71]. Sequoia [59] is a drop-in front-end for serverless platforms that allows policies to dictate how or where functions should be prioritized, scheduled, and queued. The ideal performance for a specific workload is achieved by carefully designing and evaluating several scheduling algorithms, such as resource-aware scheduling and explicit priority-based scheduling. Atoll [55] is a delay-sensitive serverless framework that exploits a shortest-remaining-slack-first algorithm for scheduling serverless functions. Atoll uses a threshold-based resource scaling method based on queuing delays. ENSURE [56] is another rule-based function resource manager. It allocates $R + c\sqrt{R}$ containers to a function with load R, scales the resources within an invoker based on a latency degradation threshold, and scales the number of invokers based on a memory capacity threshold, tuned per function workload.

However, these approaches require recurring human efforts to tune the parameters or choose the appropriate thresholds for each function to achieve optimal performance. For example, threshold-based autoscaling that relies on CPU utilization or latency degradation would be simplistic and inefficient. The parameters need to be reconstructed, tuned, and tested for varying application workloads and infrastructures. Therefore, we focus on RL-based solutions to automatically learn the optimal resource management policies and adapt to frequent changes in serverless function workloads and dynamic cloud environments.

RL-based Resource Management. Lately, RL-based approaches have gained significant momentum toward achieving application SLOs [5, 20, 28, 34, 35, 41, 43, 45, 49, 64, 72, 75, 77]. RL is well-suited for learning resource management policies, as it provides a tight feedback loop for exploring the action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules). Since resource management decisions made for each function are highly repetitive, an abundance of data is generated for training such RL algorithms. RL allows direct learning from actual workload and operating conditions to understand how allocation of resources affects application performance. It has been shown that RL with neural networks can express complex system-application environment dynamics and decision-making policies. For instance, FIRM [43] is an RL-based resource management framework for microservices, designed to tackle the under-utilization issue and SLO violations. FIRM uses a two-tier RL model to first identify the microservices that cause SLO violations and then mitigate those violations via dynamic resource reprovisioning. Schuler et al. [49] propose a Q-Learning-based autoscaler that decides on the horizontal concurrency for a serverless

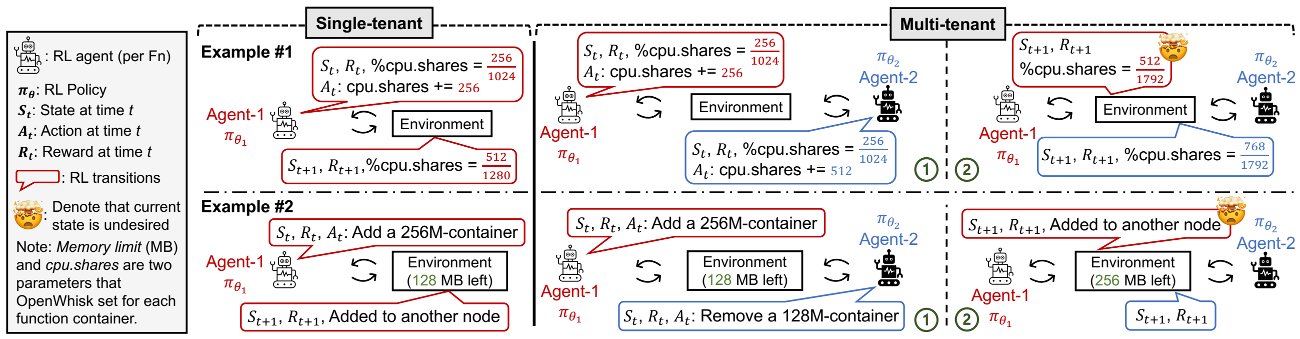
: RL agent (per Fn)
π_θ: RL Policy
S_t: State at time t
A_t: Action at time t
R_t: Reward at time t
: RL transitions
Denote that current state is undesired
Note: Memory limit (MB) and cpu.shares are two parameters that OpenWhisk set for each function container.
Single-tenant
Multi-tenant
Example #1
Example #2
Agent-1
π_{θ_1}
Agent-2
π_{θ_2}
Environment
S_t, R_t, %cpu.shares = $\frac{256}{1024}$
A_t: cpu.shares += 256
S_{t+1}, R_{t+1},%cpu.shares = $\frac{512}{1280}$
S_t, R_t, %cpu.shares = $\frac{256}{1024}$
A_t: cpu.shares += 512
S_{t+1}, R_{t+1} %cpu.shares = $\frac{512}{1792}$
S_{t+1}, R_{t+1}, %cpu.shares = $\frac{768}{1792}$
S_t, R_t, A_t: Add a 256M-container
Environment (128 MB left)
S_{t+1}, R_{t+1}, Added to another node
S_t, R_t, A_t: Remove a 128M-container
Environment (256 MB left)
S_{t+1}, R_{t+1}
1
2

Figure 3: Examples of single-agent RL failure in multi-tenant environments due to violation of the stationarity assumption.

function, with the only objective being to minimize the function latency. Zafeiropoulos et al. [77] also apply Q-Learning to threshold-based autoscaling to determine 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. FaaSRank [75] is an RL-based serverless function request scheduler that uses PPO to minimize function completion time. However, these works are all S-RL and fail to address non-stationarity in multi-tenant environments (as shown in §5.2).

Multi-armed bandits such as CloudBandits [30], as a variant of RL (i.e., one-state RL), do not apply in sequential decision problems with state transitions because they require static context with a single decision and reward. Multi-objective Bayesian optimization (BO) [31], on the other hand, is hard to scale to large state space (the known dimension-explosion problem). BO also has higher latency to find optimal solutions in the search space compared to RL inference.

2.4 Motivating Examples

We draw two examples (as shown in Fig. 3) from RL episodes in our experiments where environment non-stationarity causes suboptimal agent behaviors. In both examples, each agent controls the resource management for a serverless function and is independently trained to convergence. In the first example, suppose that agent-1 makes the decision to scale up `cpu.shares` in the single-tenant case (upper left). When both agent-1 and agent-2 are present (upper right), the action from agent-2 (i.e., scaling up `cpu.shares` by 512) affects the final CPU share ratio for both agents. Our evaluation shows that the suboptimal policy results in up to 14× performance degradation compared to the single-tenant case in terms of the end-to-end p99 latency. In the second example, agent-1 wants to scale out by adding a 256-MB container, but it ends by placing the container on a new server, since the available memory capacity is only 128 MB (bottom left). Consequently, function performance will be significantly affected by the launch of a new VM and cold starts [66]. When both agents are present (bottom right), the scale-in action from agent-2 avoids involving another server. Since agents are not aware of each other, the optimal solution is missed, leading to SLO violations and inefficient utilization of resources. We further show our quantitative characterization results in §5.

3 Single-agent RL Formulation

In this section, we present the design for serverless resource management with RL. We call this approach *single-agent RL* or *S-RL* since each RL agent manages one or more specific functions and is unaware of other agents in a shared environment. We first describe the problem formulation as an RL task (§3.1). We then outline a solution (§3.2) using a policy-based RL algorithm, Proximal Policy Optimization (PPO) [50].

3.1 S-RL Problem Formulation

We model the resource management for each serverless function (by reacting to autoscale after observing a bulk of function executions) as a sequential decision-making problem that can be formulated by the RL framework (illustrated in Fig. 2). Since all serverless FaaS platforms have similar controller-worker architectures [53, 55] and function request-serving workflows, for modeling purposes, we chose to use an open-source serverless platform called OpenWhisk [18]. At each step in the sequence, the RL agent (i.e., ❶) monitors system and application conditions from both the OpenWhisk data store (i.e., ❷) and the Linux control groups or cgroups (i.e., 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 and memory utilization of function containers). These measured telemetry data are used to define an RL state, which is then mapped to a resource management decision by the RL agent.

Action Space. We consider both vertical and horizontal resource-scaling actions. A vertical-scaling action corresponds to scaling either up or down the `cpu.shares` [60] or the memory limit of a function container, since OpenWhisk's default resource model includes `cpu.shares` and memory limits;

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

State Space S_t (for single-agent) L_t (for multi-agent)
Function SLO Preservation Ratio ($SP(t)$), Resource Utilization ($RU_{cpu}(t)$, $RU_{mem}(t)$), Function Request 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 ($AV(t)$) and Horizontal Actions ($AH(t)$), Average SLO Preservation Ratio ($MSP(t)$), Average Resource Utilization ($MRU(t)$)

both are configurable parameters in all commercial serverless platforms. A horizontal-scaling action in OpenWhisk corresponds to scaling either out or in the function containers, i.e., changing the number of created containers for a function (denoted by NC). The resource limit of each type for a function is initially over-provisioned and later managed by the RL agent. The decision made by the RL agent is then verified and passed by the horizontal and vertical scaler (i.e., 4) to the FaaS controller (i.e., 5) and finally changes the resource allocation of the function that the agent manages, and consequently the function performance. As a result, each function instance is deployed in a container with resource limits RLT_{cpu} and RLT_{mem}. Table 1 (the second row) defines the action space which includes available vertical-scaling actions that change RLT_{cpu}, RLT_{mem}, and horizontal-scaling actions that change NC.

State Space. We define the state space based on the five features listed in Table 1 (the first row). At each time step t, the average resource utilization $RU(t)$ of a function for each type of resource is retrieved from cgroups (i.e., 3) as telemetry data. The current resource allocations $RLT_{cpu}(t)$, $RLT_{mem}(t)$, and $NC(t)$ are kept as part of the state. In addition, the data store (i.e., 2) also collects function latency composition and request arrival rate. Based on these measurements, the RL agent calculates the remaining two state variables as described below:

a) *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.
b) *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 request 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 and convergence, we normalized the two variables by setting a predefined resource upper limit $\hat{R}_i$ and a lower limit $\underset{\smile}{R}_i$. 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 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.

Table 2: RL training hyperparameters in PPO [50]

Parameter	Value
Learning Rate	Actor (3×10^{-4}), Critic (3×10^{-4})
Discount Factor (γ)	0.99
Number of Hidden Layers × Units	Actor (2×64), Critic (2×64)
Mini-batch Size	10 (single-agent), 5 (multi-agent)
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 (T)	50 (per Episode)
Reward Coefficient (α)	0.3

Reward Function. The goal of the RL agent is, given a time duration T, to learn an optimal policy π_θ that results in fewer SLO violations (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 $r_t = \alpha \cdot SP(t) + (1-\alpha)/2 \cdot (RU_{cpu}(t) + RU_{mem}(t)) + penalty$, where $penalty$ is set to -1 in the following cases (and 0 otherwise): (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, an illegal action leads to a self-loop transition from a state to itself with a negative reward. (b) Undesired actions such as frequent oscillating decisions (e.g., scale down and up in two consecutive time steps), which are detected by comparing the actions of the current and last time step. Compared to masking the actions, i.e., excluding the undesired actions from the action space instead of giving negative rewards, our approach leads to faster convergence and is extensible (without modifying the action space).

3.2 S-RL Learning Framework

We use a policy-based method, PPO [50], to learn the optimal resource management policy under the RL problem formulation described above. We use PPO because it provides shorter training times and higher rewards after convergence in our setting, compared to other state-of-the-art RL algorithms (e.g., DDPG [32], DQN [62]), while being much simpler to tune. To stabilize the training process, clipping is used to prevent the policy and value functions from changing drastically between training iterations. It has been hypothesized [14] that the smooth policy updates (due to clipping) in PPO can help mitigate the non-stationarity issue in multi-agent RL. We implement both the policy π_θ and value function V_ϕ (parameterized by θ and ϕ) of PPO using neural networks. The value function evaluates the expected return of a given state [50]. The policy network maps the state to an action and its parameter θ is updated in the direction suggested by the value function. The hyperparameters used in our PPO implementation are listed in Table 2. We set those hyperparameters based on a grid search for the best results. We refer readers to [50] for detailed algorithms and rigorous derivations of PPO. The described S-RL solution is used in our characterization study (§5) and its performance in single-tenant scenarios is used as the baseline for assessing proposed solutions (as shown in §6) in multi-tenant scenarios.

4 Experimental Methodology

OpenWhisk Cluster Setup. We deployed OpenWhisk [18] 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. 2. 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 with Linux kernel version 4.15. We also deployed a larger OpenWhisk cluster (20 worker nodes) on IBM Cloud with 22 VMs in us-south-2 to study SIMPPO's scalability. Each node has 8 cores and 16–32 GB RAM, running Ubuntu 20.04. Docker containers were created on physical nodes in the local cluster and VMs in the IBM Cloud cluster. We disabled memory swapping for the Docker service. We ran the workload generator [52] and the RL module (either single- or multi-agent) from two separate nodes in the same cluster and used FaaSProfiler [52] to trace requests.

Serverless Benchmarks. We selected benchmarks from widely used open-source FaaS benchmark suites [11, 52, 76] (listed in Table 3). These benchmarks include web applications (`HTML-Gen`, `Uploader`), ML-model serving (`Sentiment-Anlys`,`Image-Inference`), multimedia (`Image-Resize`, `Compression`), scientific functions (`Primes`, `PageRank`, `Graph-BFT`, `Graph-MST`), and utilities (`Base64`, `Markdown2HTML`). Each of these function benchmarks has a different runtime behavior and resource demand in terms of CPU, memory, and I/O utilization. For example, `Image-Resize` and `Image-Inference` are compute-intensive functions; `Base64` and `Markdown2HTML` are memory-intensive functions; `Uploader` and `Compression` are I/O-bound functions; and `Page-Rank` and `Graph-BFT/MST` are data-intensive functions. The functions are written in either Python or Java. SLOs were defined on a per-function basis. In our experiments, we followed the common practice [7] and used the 99th-percentile latency as the SLO latency when running in isolation on the serverless platform. We added a 15% relaxation to the SLO latency to allow fluctuations and measurement errors.

Workloads. In the evaluation, we used both real-world and synthetic function invocation patterns. For real-world workloads, the function invocation patterns are from Azure Functions traces [53] collected over two weeks in 2019. We first constructed a dataset of invocation rates (i.e., RPS) based on the per-minute function invocation count in the traces (the intervals with no invocation will be treated as zero invocation rate). We then uniformly and randomly sampled the invocation rate at each RL step from the constructed dataset. Since our setup of 22 VMs is minimal compared to Azure Functions production setup and to allow the RL agent to explore as many invocation patterns as possible, the invocation pattern of a function in our experiment setup did not follow through with only one function trace but changed to multiple function traces during RL training. For synthetic workloads, we used common patterns [16] indicating flat and fluctuating loads, with a Poisson interarrival pattern whose parameter ranges from zero to the maximum value observed in the sampled Azure function traces. The change of request arrival rates was intended to evaluate whether RL agents could adapt to such workload changes.

Table 3: Serverless benchmarks [11, 52, 76].

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-Anlys	Generate a sentiment analysis score for the input text.
Image-Resize	Resize the Base64-coded image with new sizes.
HTML-Gen	Generate HTML files from templates.
Uploader	Upload a file from a given URL to Cloud storage.
Compression	Compress given images and upload to Cloud storage.
Image-Inference	Image recognition with a pre-trained ResNet-50 model.
Page-Rank	Calculates the Google PageRank for a specified graph.
Graph-BFT	Traverse the given graph with breadth-first search.
Graph-MST	Generate the minimum spanning tree given a graph.

5 Single-agent RL Characterization Study

5.1 S-RL in Single-tenant Cases

We implemented the single-agent RL (S-RL) solution described in §3, conducted training convergence analysis, and evaluated the function performance and resource utilization compared with those of a state-of-the-art heuristics-based approach, ENSURE [56].

S-RL Policy-training Convergence. To understand the convergence behavior of the S-RL agent in single-tenant environments, we used the workload described in §4 to train an agent for one function, with no other functions running on the platform. Since RL training proceeds in episodes (iterations), we then analyzed the per-episode reward evolution; Fig. 4 (the red curve above) shows the results for function `Primes` and we found that the agent-training progress is similar across different function benchmarks. We fixed the number of time steps per episode to 50 (i.e., T in Table 2). In the initial training stage, the agent policy was unable to mitigate SLO violations. Hence, we terminated the RL exploration early to reset the agent to the initial state and enter the next episode. As the training progressed, the agent improved its resource allocation policy and could mitigate SLO violations in less time. At that point (after around 70 episodes), we linearly increased the number of time steps to let the

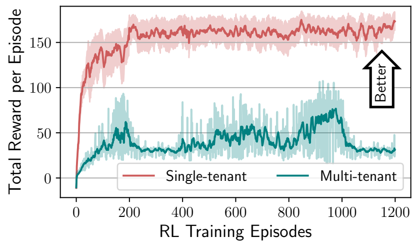

Figure 4: Training curves of the single-agent RL (for function `Primes`) in single- and multi-tenant environments.

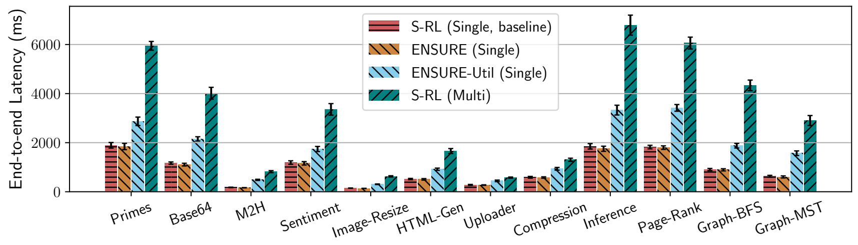

Figure 5: Single-agent RL 99th-percentile end-to-end function latency in single- and multi-tenant environments for all function benchmarks in comparison with the heuristics-based approach ENSURE [56] (in single-tenant environments).

agent interact with the environment for more 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 all benchmarks).

S-RL Policy-serving Performance. After convergence, we leveraged the trained agent (by using the saved checkpoints at the 1000th episode), and compared it with ENSURE [56], which is a state-of-the-art threshold-based autoscaler implemented on OpenWhisk. We set the parameters and thresholds according to author recommendations [56]. In this comparison, each function controlled by an agent is tested independently. Fig. 5 shows the online performance comparison. The S-RL agent is able to keep the CPU utilization at a higher level (around 24% higher than ENSURE, not shown in the figure) and achieves similar end-to-end latency. We found that the reason is that ENSURE over-provisions containers and resources when workload changes. After increasing the function latency threshold to a higher value (from 15% to 25%), we observed that the performance of the S-RL agent improved over that of ENSURE (labeled "ENSURE-Util") by at least 1.6× with respect to tail latencies at similar CPU utilizations. However, ENSURE does not require any training.

By interacting with dynamic serverless environments under diverse loads and resource allocation scenarios, S-RL agents gradually learn the policy that maximizes the expected rewards and hence outperform heuristics-based approaches. We regard S-RL in single-tenant cases as the baseline for assessing solutions in multi-tenant cases.

5.2 S-RL in Multi-tenant Cases

Serverless FaaS platforms are, in essence, multi-tenant, running different functions with various function characteristics, SLOs, and workload patterns from multiple customers [4, 21, 44, 66]. Each function managed 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 [44, 52, 66]. 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 explore the environment non-stationarity issue, and conduct training convergence analysis and policy-serving performance assessment of S-RL agents in multi-tenant environments.

Environment Non-stationarity. The shared environment in a many-agent setting is 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 [42, 57, 69] made by S-RL algorithms. In policy training or serving, S-RL requires that the isolated environment is affected only by the agent interacting with it. 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. From a system's perspective, the RL `step()` function (i.e., take an action and receive the next state and reward) can be viewed as a "transaction." An agent interacts with the environment at each time step and commits a transaction that may conflict with other transactions from different agents because of request-response function dependencies or shared-resource contention. The stationarity assumption means that an agent needs to obtain the "lock" to avoid race conditions where the environment ("critical section") is updated by two or more transactions. S-RL fails to provide any synchronization mechanism to address non-stationarity issues.

S-RL Policy-serving Performance Degradation. We assessed the policy-serving performance of the S-RL agent in multi-tenant cases in which each function is in control of an independent S-RL agent trained in isolation (oblivious of the other agents). Fig. 5 shows the performance degradation after multiple functions are introduced on the same platform (green bars compared to red bars). In this experiment, we created one function for each benchmark from Table 3 and trained one S-RL agent for each function in isolation until convergence. Then, we ran all 12 functions concurrently on OpenWhisk, with each function being managed by its trained S-RL agent. The evaluation results show that the degradation is up to 4.8× (for `Graph-BFS`) and as low as 2.2× (for `Compression`). The reason is that when the agent makes a decision, it is based on the states measured at the current time step; but at the same time, all other agents are also making their resource allocation decisions, which could affect the shared environment. Therefore, the state could have been changed and the estimated value function for an action by the S-RL model is no longer accurate.

S-RL Policy-training Convergence Failure. Multi-tenancy not only affects the online performance during policy serving for S-RL agents trained in isolation, but also leads to problems during training. We trained 12 S-RL agents together; each of them managed one function from the benchmarks listed in Table 3. Each S-RL agent was trained independently and did not consider the other agents in the same environment. Everything else was kept the same as before, with the S-RL agent trained in isolation. Fig. 4 (the green curve below) shows the per-episode reward evolution for the agent managing function `Primes`. Compared to the training curve of the S-RL agent trained in isolation (in red), the S-RL agent trained in a multi-tenant environment achieved a lower performance (a 55.9% drop in terms of per-episode reward) with higher variance and did not converge in a stable manner within the same training budget. We observed that other function benchmarks also failed to converge within the same training budget (not shown due to page limit).

While S-RL converges in isolated environments and provides a sufficient baseline compared with heuristics, system support for many-agent RL-based resource management that provides both training convergence and comparable policy-serving performance is needed to deal with environment non-stationarity.

6 SIMPPO Design and Implementation

We describe the design and implementation of SIMPPO in this section, which includes a remodeled resource management problem formulation (in §6.1) as a multi-agent extension of the Markov decision process (MDP), and a MARL learning framework based on PPO (in §6.2) under the problem formulation. All agents are jointly trained in SIMPPO by allowing each agent to peek into the behavior of the other agents, which potentially violates the end-to-end arguments [48]. However, the violation is necessary for performance gain [39] as we show in §5 that obeying the end-to-end arguments by handling non-stationarity at each agent's end sacrifices policy-serving performance. Our design choices for the multi-agent model were made to favor scalability and adaptability to agent churn caused by added/removed functions. SIMPPO enables the convergence behavior of all agents during training and provides online policy-serving performance comparable to that of the baseline, i.e., S-RL in single-tenant cases (as shown in §7).

6.1 SIMPPO's MARL Formulation

We extend the MDP formulation for single-agent RL (described in §3) to a Markov game (also known as a stochastic game [54]) for N agents, each of which controls the resource management for one particular function on the serverless platform. 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 action and reward of each agent are the same as defined in S-RL. The environment state then transitions to a new state that depends on the joint action of all the agents.

A Naive Approach. One can overcome the non-stationarity issue introduced by other agents in the shared environment by using centralized learning (e.g., joint action learners or JAL [10]). In such a centralized approach, the agents are jointly modeled, and a centralized policy for all the agents is trained. The exponential dependence on the number of agents N makes the centralized learning approach difficult to scale up beyond a few agents [10]. Therefore, given that there could be tens or even hundreds of function instances on a server [61], we do not proceed further with this approach.

Virtual Agent. The non-stationarity problem leads to significant scalability challenges, since each agent must reason about every other agent's internal state of information. In addition, existing MARL approaches explicitly model each function or agent (e.g., the JAL [10]) and thus the whole MARL algorithm needs to be retrained when there is any changes to the agent group, 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 SIMPPO, from the point of view of an agent i, the main idea is to treat all the other agents as part of the environment by creating a "virtual" agent that represents the environment and all the other agents. Introducing the virtual agent allows SIMPPO to be agnostic to the number of agents and the order of the agent sequence, which enables incremental training. We created *auxiliary global states* (which we will discuss in the next paragraph) by learning the collective and average behavior of the virtual agent instead of each of the other individual agents. Auxiliary global states are provided to each agent to help it adapt to varying agents in the environment. Reducing the interaction between an agent and the others to the interaction between the agent and the virtual agent greatly simplifies the scalability issue.

Value Function Inputs. In SIMPPO's MARL formulation, we resort to using the average to represent the behavior of the other agents (i.e., the virtual agent), and to providing each agent with an auxiliary global state in addition to its individual state. Each agent extracts its local state l_t^i and auxiliary global state g_t^i from the environment state s_t. The local state l_t^i of agent i is from the same state space as in S-RL (Table 1 the first row). Both local and auxiliary global states are used as the inputs to the value function, and we find that omitting any local state can be highly detrimental to learning of an optimal resource management policy. In addition, we select a subset of five state-action variables from all variables that can be used to represent the auxiliary global state g_t^i based on domain knowledge. First, aggregated actions and resource limits from all the other agents represent the collective action, since we view them as part of the environment. Second, as the goal is to achieve function SLO performance while maintaining high resource utilization, we select the mean SLO preservation ratio and resource utilization to represent how the other agents behave. Third, we observe that reducing the dimensionality of the value function inputs by removing redundant or repeated features further improves the RL agent performance and reduces training time. For example, a higher arrival rate or a lower horizontal concurrency could have been manifested by a lower SLO preservation ratio. Since each agent only cares about and optimizes its policy based on the reward function, redundant features can negatively affect RL training and make it hard for the neural network to learn to extract the real, useful features. Specifically, g_t^i consists of the following (the third row in Table 1):

a) *Aggregated resource limits*: $ARLT^i_{cpu}(t) = \sum_{j\neq i}^{N} RLT^j_{cpu}(t)$, and $ARLT^i_{mem}(t) = \sum_{j\neq i}^{N} RLT^j_{mem}(t)$
b) *Aggregated vertical actions*: $AV^i(t) = \sum_{j\neq i}^{N} \Delta RLT(t)$
c) *Aggregated horizontal actions*: $AH^i(t) = \sum_{j\neq i}^{N} \Delta NC^j(t)$
d) *Mean SLO preservation*: $MSP^i(t) = \sum_{j\neq i}^{N} SP^j(t)/(N-1)$
e) *Mean resource util*: $MRU^i(t) = \sum_{j\neq i}^{N} RU^j(t)/(N-1)$

In addition, to measure the volatility of agent performance and behavior, we also include the standard deviation of the selected variables across all the other agents.

Our design draws inspiration from the mean-field theory [46, 47], which has proved to be successful in fields like economics [2] and physics [70]. The underlying principle is to approximate the finite-agent system by summarizing the collective behavior of all agents as a population distribution, which is usually specified as the empirical distribution of the agents' states. Existing theory [46, 47] and our own work [36] show that the approximation error goes down as the number of agents increases because there is less influence from each agent on the overall system.

Rewards. The goal in the MARL setting is (see [78, 79]), given a time duration T, to determine an optimal collection of policies $\pi = \{\pi_{\theta^1}, \pi_{\theta^2}, ..., \pi_{\theta^N}\}$ that result 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 average resource utilization of each type 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 $r_t = \sum_{i=1}^{N} r_t^i/N$, where r_t^i is the reward for agent i as defined in §3. Our objective is to maximize the expected cumulative discounted reward $\mathbb{E}[\sum_{t=0}^{T}\gamma^t r_t] = \mathbb{E}[\sum_{t=0}^{T}\gamma^t \cdot \sum_{i=1}^{N} r_t^i/N]$.

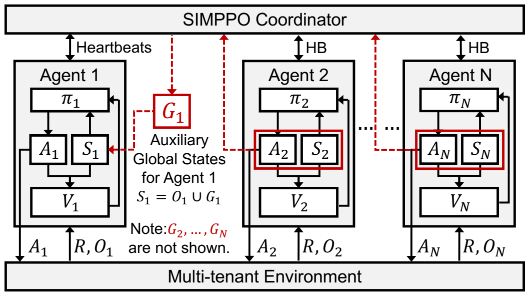

Figure 6: Overview of the multi-agent RL model SIMPPO.

6.2 SIMPPO's Learning Framework

Based on SIMPPO's scalable and incremental MARL formulation in §6.1, we introduce the learning framework of SIMPPO based on the PPO algorithm that we designed for S-RL (in §3.2). SIMPPO 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. Fig. 6 shows an overview of the SIMPPO algorithm. We concatenate the auxiliary global state g^i (described in §6.1) with each agent's local state l^i and feed them as the inputs to the value network V_{ϕ^i}. The auxiliary global state g^i is collected and calculated by the SIMPPO coordinator and sent to each SIMPPO agent. For example in Fig. 6, G_1 is sent to Agent 1. The policy network π_{θ^i} is the same as the policy network in S-RL, which maps the states to an action from the same action space. The other extension from S-RL is that the reward for each agent at time t is changed to the average reward across all agents: $r_t = \sum_{i=1}^{N} r_t^i/N$. The RL policy update step of each SIMPPO agent is done in parallel.

Training Data Reuse. PPO uses mini-batch gradient descent to perform several epochs of updates on a batch of training data. We find that the agent achieves the optimal performance when the mini-batch size is set to 10 in the single-agent setting. However, in the multi-agent setting, we find that a smaller mini-batch size (i.e., 5) results in a better performance. We attribute this to the negative effect that high data reuse brings in multi-agent settings [74]. The number of epochs implicitly determines the non-stationarity in MARL, as more training epochs will cause larger changes to the agents' policies, which exacerbates the non-stationarity issue. The other hyperparameters are kept the same as in Table 2 which are used by the S-RL agents.

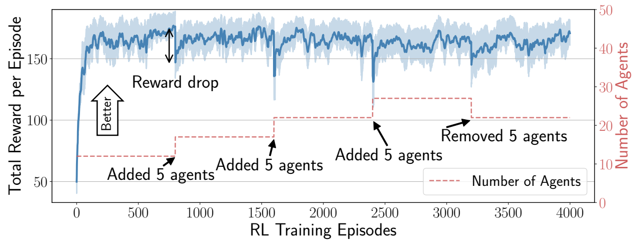

Figure 7: Incremental training of SIMPPO.

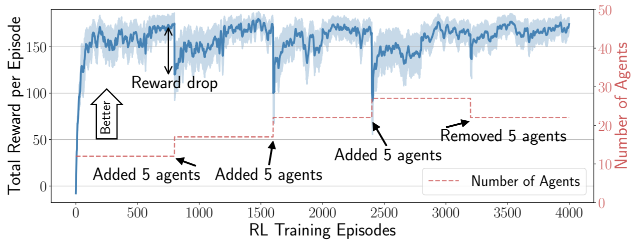

Figure 8: Incremental training (no parameter sharing).

Adding and Removing Agents. The SIMPPO coordinator and all SIMPPO agents follow the server-client communication model, and the coordinator is responsible for adding and removing an agent upon request (i.e., when registering or deleting a function). We use a heartbeat-based membership protocol between the SIMPPO coordinator and all SIMPPO agents. When a new function is being added to the serverless platform, a new SIMPPO agent will be initialized to control the resource management of the function. After a new SIMPPO agent is added or an existing function is removed, the auxiliary global state is updated by including or excluding the individual observations from that agent.

Network Parameter Sharing. We leverage neural network parameter sharing [58, 81] between SIMPPO agents that manage the same type of functions to shorten the incremental training time. Based on their sensitivity [15] to the allocation of each type of resource, we categorize the function benchmarks into three coarse-grained categories[2]: CPU-intensive, memory-intensive, and I/O-intensive. One trained RL model is used as the base model for each category of functions. The base model is obtained by training the RL agents that manage all functions that belong to the corresponding category. When a new function is added to the serverless platform, we first categorize the function and initialize the newly created SIMPPO agent with the network parameters from the base model of the same category. The base model can be slowly updated in the background by replacing its network parameters with the existing SIMPPO agents that manage the same type of functions.

7 Evaluation

We evaluated SIMPPO on OpenWhisk driven by widely used serverless benchmarks [11, 52, 76] with both synthetic and real-world workloads from production traces [53], as specified in §4. Our experiments addressed the following research questions:

§7.1 Does SIMPPO converge and support incremental training? What is the value of auxiliary global states?
§7.2 How does SIMPPO perform in multi-tenant environments compared to single-agent RL in single-tenant environments?
§7.3 What is the resource overhead of SIMPPO?
§7.4 Is SIMPPO scalable with respect to converged rewards, online policy-serving performance, and retraining time?

[2]We leave fine-grained function classification to future work.

7.1 Incremental Training

During the policy-training stage, we intentionally added and removed a few agents at random from the environment to evaluate the adaptability of the SIMPPO model to agent updates. Fig. 7 shows the training curves of SIMPPO in multi-tenant environments. To start with, we created 12 functions (one from each benchmark in Table 3); each function was then managed by an initialized SIMPPO agent. Since all agents used the team reward (i.e., the average reward across all agents), Fig. 7 shows the evolution of the average total reward per episode. All agents were able to reach a stable converged policy after around 500 episodes (around 2 hours). Then, at episode 800, we updated the multi-tenant environment by adding five functions (randomly selected from the benchmarks), each of which was managed by a separate SIMPPO agent. As shown in the figure, the total reward per episode dropped by 16.6%; that happened mainly because the five new SIMPPO agents started to learn the optimal policy, which initially leads to low reward values. After around 300 more episodes (around 1.2 hours), the learning curve of the SIMPPO agents was able to converge again. We updated the environment three more times (with one update after every 800 episodes) by either adding five (randomly chosen) new functions or removing five (randomly chosen) existing functions. We observed a similar reward drop and later convergence to about the same level after several hundred training episodes. When we removed five existing functions from the environment, the reward drop (i.e., 11.3%) was not as large as in the previous cases. We attribute the smaller reward drop to the fact that there was no added agent whose reward could start to become randomly lower than that of a trained agent. The team reward still dropped because of the fluctuation of the environment as five functions were removed.

Ablation Study. To study the benefit of neural network parameter sharing brought to incremental training, we implemented a variant of SIMPPO without parameter sharing between any two SIMPPO agents; Fig. 8 shows its training

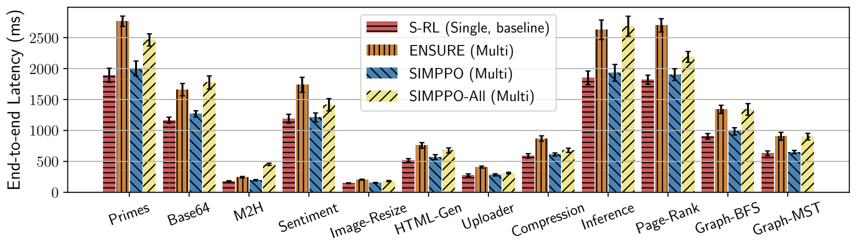

Figure 9: The 99th-percentile end-to-end latency comparison of functions managed by SIMPPO and the mean-field version of SIMPPO ("SIMPPO-All") agents. The comparison baseline is single-agent RL (S-RL) in single-tenant environments.

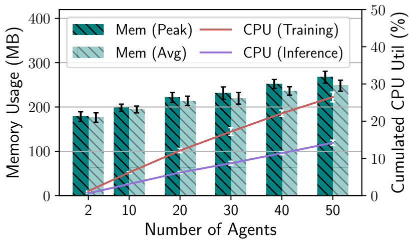

Figure 10: Memory usage and CPU utilization overhead analysis of SIMPPO with regard to the number of agents.

curves in the same experimental setup used for the vanilla version of SIMPPO. The benefit of network parameter sharing is manifested in the retraining phase when there is a significant change to the agent group. Compared to SIMPPO without network parameter sharing, the vanilla version has a 20–54.8% smaller reward drop each time functions are added to or removed from the platform. Consequently, the retraining time is reduced by about half (from around 600 episodes) using the learned based model of each type of functions.

7.2 Online Policy-serving Performance

We saved the checkpoints at the 3200th episode for all agents in §7.1 and used those checkpoints to evaluate the online performance during policy serving for all functions together on the serverless platform. We observed that the function performance was similar at different episodes when the agents' behavior converged. At the 3200th episode, there were 27 functions in total (all function benchmarks were used). Fig. 9 shows a function performance comparison between SIMPPO in the multi-tenant environment and S-RL trained in a single-tenant environment, i.e., the baseline. We average over all function instances of the same function benchmark. As shown in the figure, SIMPPO is able to provide online performance comparable to that of the baseline, with the performance degradation ranging from 1.8% (for `Sentiment-Anlys`, 1190.2 ms to 1211.5 ms) to 9.2% (for `Markdown2HTML`, 178.8 ms to 196.8 ms). In contrast, the performance degradation for ENSURE in multi-tenant environments ranged from 26.9% to 32.5%, up to 21.4× higher than for SIMPPO (for `Uploader`, SIMPPO improves the degradation from 31.6% to 1.5%). Compared to the single-agent RL trained in multi-tenant environments (as shown in §5.2 and Fig. 5), SIMPPO achieved from 2× (for `Uploader`, 283.4 ms to 576.3 ms) to 4.5× (for `Graph-MST`, 651.8 ms to 2902.6 ms) improvement in terms of the 99th-percentile function latency. In terms of application categories, we found that SIMPPO performs better for I/O-intensive (1.5–3.6%), then CPU-intensive (3.9–4.2%), and memory-intensive (7.9–9.2%) workloads. We attribute this to better performance predictability of I/O-intensive functions [44] which are less sensitive to memory-bandwidth/CPU contention compared to memory or CPU-intensive workloads.

Ablation Study. To study the importance of feature selection with domain knowledge when constructing the auxiliary global states, we implemented a variant of SIMPPO by replacing the selected auxiliary global states with the average of all individual states from each agent, as suggested in mean-field theory. Fig. 9 (in the yellow bars denoting SIMPPO-All) shows its online performance in the same experimental setup used for the vanilla version. We found that the performance degradation from the single-agent RL baseline in single-tenant settings increased by 2.6× (`HTML-Gen`) to 9.7× (`Graph-MST`) compared to the vanilla version of SIMPPO. The results indicate that our selection of auxiliary global states based on domain knowledge avoids redundant features that increase training time (not shown in this paper) and negatively affect policy-serving performance because it is hard for the neural network to learn to extract the real, useful features.

7.3 Policy Training and Serving Overhead

RL Model Overhead. The actor and critic networks for each SIMPPO agent consist of 4993 parameters (i.e., 28 KB) that are mostly from two neural-network layers with 64 hidden units per layer. Since SIMPPO is implemented in Python with several machine learning libraries such as PyTorch, the imported libraries account for about 166 MB of memory that is shared across all SIMPPO agents. The last part of memory consumption comes from the intermediate data (such as RL state-action transitions and reward vectors) used during training; the size of this memory is proportional to the number of agents and is about 2 MB per agent on average (as shown in Fig. 10).

RL Policy-training Overhead. To measure the overhead of RL training for learning an optimal resource management policy, we profile SIMPPO's training process, which proceeds in iterations. The agents all update their model parameters in parallel (using the training approach described in §6.2). The RL `step()` function takes 240 ± 13 ms, and the actor-critic network parameter update takes an average of 1.34 ± 0.21 s (i.e., five SGD epochs). Thus, in total, training from scratch takes up to around 2 hours, and incremental retraining takes around 0.2–1.8 hours (on Intel Xeon E5-2683). Retraining can be performed infrequently depending on environment stability. We note that since function execution and policy training are asynchronous, the training cost does not directly affect the execution time. However, agents during training are not able to produce optimal decisions although functions can still execute to completion. Most CPU time is spent on network parameter updating, and the overall CPU utilization during training is negligible (i.e., 0.6% per SIMPPO agent, as shown in Fig. 10), since the parameter update is performed only once per iteration.

RL Policy-serving Overhead. Mapping of a current state to derive an action requires about 240 ms on average (the same as the RL `step()` function during policy training). As the function container resource management is on an independent control plane and asynchronous with the function request scheduling or serving, the RL decision latency does not directly impose any overhead on function execution. The CPU utilization during policy serving is negligible (i.e., 0.3% per SIMPPO agent). With reasonable resource overhead, SIMPPO still improves utilization efficiency.

7.4 Scalability Analysis

Finally, we deployed an OpenWhisk cluster with SIMPPO on 22 VMs on IBM Cloud. One VM with 8 cores and 32 GB of memory hosted containers for the controller and other main components, including Nginx and Kafka. One VM with 16 cores and 16 GB of memory hosted all RL agents. Each of the remaining 20 VMs had 8 cores and 16 GB of memory, and hosted an invoker to run the functions in Docker containers. We first confirmed that SIMPPO's training convergence and online policy-serving performance were unchanged compared to those for the local cluster for settings specified in §7.1 and §7.2. We then further increased the number of registered functions from 20 to 130 (with the total available memory capacity and cores scaled linearly with the number of functions). We swept over the number of functions to add or remove functions (i.e., 2, 5, or 10 functions) each time function churn happened.

Incremental Training. At each function churn, we recorded the per-episode reward drop from the previously converged reward to the reward received at the first episode after the churn. To evaluate the effect of agent group size on SIMPPO's incremental training, we also recorded the retraining time required to reach convergence. Table 4 shows the reward drop percentage and the retraining time (in terms of the number of training episodes). We observe that as the number of functions increases from 5 to 110, the reward drop percentage first increases and then decreases after the number of functions is greater than 35. When we added 2 or 5 functions at a time, SIMPPO had the highest reward drop (i.e., 17.9% and 20.2%) if there were 5 existing functions; when we added 10 functions, SIMPPO had the highest drop (i.e., 42.7%) if starting at 20 functions. Accordingly, the retraining time has similar trends, as retraining is done until the per-episode reward converges to a stable value (not changing by more than 2%). The most expensive training cost ranged from 196.2 to 455.1 training episodes, on average, for each starting number of functions (about 0.8 to 1.8 hours). Removal of functions results in less reward drop and thus less retraining time for each setup, the same as what we describe in §7.1. As the number of functions increases to 110, the reward drop percentage and the retraining cost decrease to as little as 3.0% and 47.2 training episodes. We attribute this to the formulation of auxiliary global states for each agent that is used to model and approximate the collective behavior of all the other agents. As the number of agents in the system increases, the disturbance introduced at each function churn gets smaller and the approximation error goes down [36, 46, 47].

Table 4: Scalability analysis of SIMPPO in terms of retraining time in episodes (RT) and reward drop percentage (RDP) after a function churn (i.e., addition or removal of functions).

From	FN Churn	Add New Functions			Remove Functions		
		+2 FNs	+5 FNs	+ 10 FNs	-2 FNs	-5 FNs	-10 FNs
5 FNs	RDP (%)[†]	**17.9%**	**20.2%**	38.6%	**9.1%**	—	—
	RT (# Eps)[‡]	**288.5** ± 22	**369.9** ± 18	435.6 ± 24	**129.6** ± 19	—	—
20 FNs	RDP (%)	13.3%	16.6%	**42.7%**	9.3%	**10.3%**	**16.2%**
	RT (# Eps)	234.4 ± 21	325.6 ± 20	**455.1** ± 20	133.4 ± 18	**190.5** ± 17	**309.1** ± 20
35 FNs	RDP (%)	13.0%	15.4%	31.4%	7.1%	8.8%	14.4%
	RT (# Eps)	216.4 ± 20	284.8 ± 18	315.4 ± 19	120.7 ± 17	167.6 ± 18	268.7 ± 20
50 FNs	RDP (%)	9.3%	12.8%	24.2%	5.8%	6.6%	11.0%
	RT (# Eps)	145.8 ± 17	243 ± 16	289.7 ± 21	105.1 ± 14	149.3 ± 16	248.5 ± 18
65 FNs	RDP (%)	8.7%	10.3%	21.5%	4.9%	5.9%	11.3%
	RT (# Eps)	150.2 ± 15	208.6 ± 14	247.2 ± 17	82.6 ± 15	126.5 ± 15	208.3 ± 17
80 FNs	RDP (%)	6.9%	9.8%	18.5%	3.8%	5.1%	7.8%
	RT (# Eps)	140.0 ± 15	204.4 ± 13	241.8 ± 16	66.4 ± 17	117.5 ± 14	228.6 ± 21
95 FNs	RDP (%)	6.0%	8.5%	17.0%	3.2%	**4.7%**	7.4%
	RT (# Eps)	**135.8** ± 13	193.0 ± 12	215.3 ± 15	54.0 ± 15	100.6 ± 12	141.3 ± 17
110 FNs	RDP (%)	**5.9%**	**7.7%**	**14.9%**	**3.0%**	4.8%	**6.5%**
	RT (# Eps)	136.1 ± 11	**185.8** ± 13	**196.2** ± 15	**47.2** ± 11	**97.9** ± 13	**132.0** ± 17

[†]RDP = $(R_{before} - R_{after})/R_{before}$ [‡]RT: Num. of RL training episodes for SIMPPO to converge

Policy-serving Performance. We evaluated the online performance of the learned SIMPPO model when the number of agents was 20 and 110 by taking the model checkpoints after convergence. Fig. 11 shows that the distributions of both function end-to-end latency and CPU utilization for the model-serving benchmark `Sentiment-Anlys` are almost the same whether the number of agents is 20 or 110. (Similar trends were observed for other function benchmarks but are not shown in the figure.) The percentage difference of the p99 latency for each function between the 20- and 110-agent settings is smaller than 1.9%. More rigorously, we

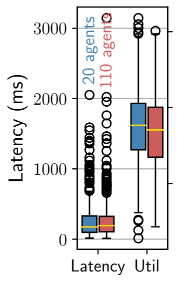

Figure 11: RL policy serving.

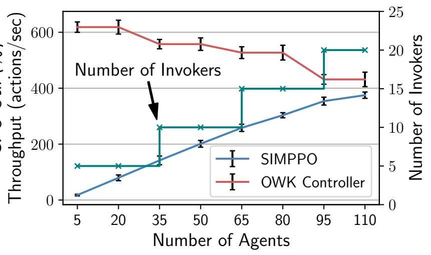

Figure 12: RL action throughput.

ran statistical testing to determine whether the two empirical data distributions are the same (i.e., the null hypothesis). The results show that the two-sided p-values for function latency and CPU utilization are 0.56 and 0.67 (> 0.05) so we cannot reject the null hypothesis. Therefore, we conclude that SIMPPO's online policy-serving performance is scalable in the number of agents.

Throughput. As the number of SIMPPO agents increases, the generated resource-scaling actions per time step grows almost linearly (as shown in Fig. 12), with the overhead mainly coming from the calculation of the auxiliary global states. Compared to the throughput that the OpenWhisk controller is able to handle, SIMPPO's throughput demand becomes smaller, but the difference shrinks as the number of invokers increases. In §8, we discuss a potential solution for scaling beyond the controller's limit.

8 Discussion

Performance SLOs. No commercial cloud provider offers SLOs on performance (only availability), which hinders the adoption of latency-critical services on serverless platforms. Though we infer SLOs by profiling applications, a mechanism to convey any SLO preference is needed. We also assume that all user requests sent to the same function correspond to the same SLO. However, the RL policy may not converge when an unachievable SLO is specified. SLO-aware serverless resource management could potentially (a) enable the serverless provider to offer SLO guarantees and change pricing models to be SLO-aware; (b) enable the adoption of latency-sensitive applications from traditional cloud computing to serverless computing.

Scalability Bottleneck. To scale scheduling and resource management, the most widely used approach [13, 51, 55, 63, 65] is to partition the cluster into several system pools and have one controller per system pool. This ensures that a controller does not become a scalability bottleneck. In that scenario, SIMPPO can be applied to make optimal decisions within each system pool.

Compatibility with Non-RL-based Approaches. SIMPPO can be applied to non-RL-based approaches by replacing the RL policy network with their static resource management policy. For example, in threshold-based scaling [29] for deployment with a target CPU utilization of 50%, if five pods are currently running and the mean CPU utilization is 75% (i.e., state), the controller will add 3 replicas (i.e., action) to move the pod average closer to the target. High-stakes applications or those with high function churn rates could use non-RL-based approaches and train the RL model in an offline manner [32, 62].

Function Chains. SIMPPO does not explicitly consider function dependencies. Since our approach is reactive (autoscaling based on observation), dependencies are indirectly addressed through performance or utilization measurements. To explicitly model dependencies, critical service localization in FIRM [43] could be potentially applied.

Extensibility. SIMPPO could be potentially applied to generic multi-dimensional autoscaling or resource management problems but not in the IaaS domain because each tenant manages resources, and maintains performance-SLOs, but cannot peek into others. To allow more than one objective among a wide range of users, a function with different objectives needs to be turned into different instances each of which is then managed by an agent. The RL reward function also needs to be updated if the objective is not about latency.

Multi-type Virtual Agents. Agents managing functions that have different behaviors (e.g., compute- or memory-intensive) can be represented using separate virtual agents. In addition, different percentile statistics, such as median or p99, can be used to help estimate the collective behavior more accurately. We leave the exploration of multi-type fine-grained virtual agents to future work.

9 Conclusion

This paper presents the issues involved in infusing multiple learning-based resource management agents in multi-tenant serverless platforms through a quantitative characterization study. We propose SIMPPO, a scalable and incremental MARL framework that (a) resolves the many-agent non-convergence problem while providing online policy-serving performance comparable to that of the baseline (i.e., S-RL in isolation), and (b) is scalable and adaptive to varying agent groups with reasonable resource overhead.

Acknowledgments

We thank the anonymous reviewers and our shepherd Mohammad Shahrad 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; and by the IBM-ILLINOIS Discovery Accelerator Institute (IIDAI). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or IBM.

References

[1] Amazon. 2022. AWS Lambda Concurrency Limit. https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html. Accessed: 2022-04-14.

[2] Andrea Angiuli, Jean-Pierre Fouque, and Mathieu Lauriere. 2021. Reinforcement Learning for Mean Field Games, with Applications to Economics. https://arxiv.org/abs/2106.13755

[3] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. 2017. A Brief Survey of Deep Reinforcement Learning. *CoRR* abs/1708.05866 (2017), 16 pages. http://arxiv.org/abs/1708.05866

[4] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, et al. 2017. *Serverless Computing: Current Trends and Open Problems.* Springer Singapore, Singapore, 1–20.

[5] Subho Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. Inductive-bias-driven Reinforcement Learning for Efficient Scheduling in Heterogeneous Clusters. In *Proceedings of the 37th International Conference on Machine Learning (ICML 2020).* PMLR, Cambridge, MA, USA, 629–641.

[6] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-scale Machines* (3rd ed.). Morgan & Claypool Publishers, San Rafael, CA, USA.

[7] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly Media, Inc., Sebastopol, CA, USA.

[8] Olivier Buffet, Alain Dutech, and François Charpillet. 2001. Incremental Reinforcement Learning for Designing Multi-agent Systems. In *Proceedings of the 5th International Conference on Autonomous Agents (AGENT 2001).* Association for Computing Machinery, New York, NY, USA, 31–32.

[9] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. 2010. Multi-agent Reinforcement Learning: An Overview. *Innovations in Multi-agent Systems and Applications* 331, 1 (2010), 183–221.

[10] Caroline Claus and Craig Boutilier. 1998. The Dynamics of Reinforcement Learning in Cooperative Multi-agent Systems. *Proceedings of the 15th National Conference on Artificial Intelligence and the 10th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 1998)* 2 (1998), 746–752.

[11] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference (Middleware 2021).* Association for Computing Machinery, New York, NY, USA, 64–78.

[12] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: Latency-aware Provisioning and Scaling for Prediction Serving Pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC 2020).* Association for Computing Machinery, New York, NY, USA, 477–491.

[13] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, et al. 2019. Hydra: A Federated Resource Manager for Data-center Scale Analytics. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2019).* USENIX Association, Berkeley, CA, USA, 177–192.

[14] 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:2011.09533* abs/2011.09533 (2020), 11 pages.

[15] Christina Delimitrou and Christos Kozyrakis. 2013. iBench: Quantifying Interference for Datacenter Applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC 2013).* IEEE Computer Society, Washington, DC, USA, 23–33.

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

[17] Sundar Dev, David Lo, Liqun Cheng, and Parthasarathy Ranganathan. 2020. Autonomous Warehouse-Scale Computers. In *2020 57th ACM/IEEE Design Automation Conference (DAC).* Association for Computing Machinery, New York, NY, USA, 1–6.

[18] Apache Software Foundation. 2022. OpenWhisk. https://github.com/apache/openwhisk. Accessed: 2022-04-14.

[19] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019).* Association for Computing Machinery, New York, NY, USA, 19–33.

[20] 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), 23 pages.

[21] 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 6th International Workshop on Serverless Computing (WoSC6) 2020.* Association for Computing Machinery, New York, NY, USA, 43–48.

[22] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2020).* USENIX Association, Berkeley, CA, USA, 443–462.

[23] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2011).* USENIX Association, Berkeley, CA, USA, 1–14.

[24] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)* (Virtual, USA). Association for Computing Machinery, New York, NY, USA, 152–166.

[25] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. Practical Scheduling for Real-World Serverless Computing. *arXiv preprint arXiv:2111.07226* (2021), 14 pages.

[26] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. 2018. Henge: Intent-Driven Multi-Tenant Stream Processing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC 2018)* (Carlsbad, CA, USA). Association for Computing Machinery, New York, NY, USA, 249–262.

[27] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. 2019. GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys 2019).* Association for Computing Machinery, New York, NY, USA, 16 pages.

[28] 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.

[29] Kubernetes. 2022. Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed: 2022-04-14.

[30] M. Lazuka, T. Parnell, A. Anghel, and H. Pozidis. 2022. Search-based Methods for Multi-Cloud Configuration. In *IEEE 15th International Conference on Cloud Computing (CLOUD 2022)*. IEEE Computer Society, Los Alamitos, CA, USA, 438–448. https://doi.org/10.1109/CLOUD55607.2022.00067

[31] Qian Li, Bin Li, Pietro Mercati, Ramesh Illikkal, Charlie Tai, Michael Kishinevsky, and Christos Kozyrakis. 2021. RAMBO: Resource Allocation for Microservices Using Bayesian Optimization. *IEEE Computer Architecture Letters* 20 (2021), 46–49.

[32] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous Control with Deep Reinforcement Learning. In *Proceedings of the 4th International Conference on Learning Representations (ICLR 2016)*. Open Review, Online, 14 pages. https://arxiv.org/abs/1509.02971

[33] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA 2015)*. Association for Computing Machinery, New York, NY, USA, 450–462.

[34] 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)*. Association for Computing Machinery, New York, NY, USA, 50–56.

[35] Hongzi Mao, Parimarjan Negi, Akshay Narayan, Hanrui Wang, Jiacheng Yang, Haonan Wang, Ryan Marcus, Mehrdad Khani Shirkoohi, Songtao He, Vikram Nathan, et al. 2019. Park: An Open Platform for Learning-augmented Computer Systems. *Advances in Neural Information Processing Systems (NIPS 2019)* 32 (2019), 13 pages.

[36] Weichao Mao, Haoran Qiu, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, and Tamer Başar. 2022. A Mean-Field Game Approach to Cloud Resource Management with Function Approximation. In *Proceedings of the 36th Conference on Advances in Neural Information Processing Systems (NIPS 2022)*, Vol. 36. Curran Associates, Inc., New Orleans, LA, USA, 1–12.

[37] 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 Computer Society, Washington, DC, USA, 583–592.

[38] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G Kulkarni, Dan Li, Jinho Hwang, KK Ramakrishnan, and Timothy Wood. 2021. Mu: An Efficient, Fair and Responsive Serverless Framework for Resource-constrained Edge Clouds. In *Proceedings of the 12th ACM Symposium on Cloud Computing (SoCC 2021)*. Association for Computing Machinery, New York, NY, USA, 168–181.

[39] Tim Moors. 2002. A Critical Review of "End-to-end Arguments in System Design". In *Proceedings of IEEE International Conference on Communications 2002 (ICC 2002)*, Vol. 2. IEEE Computer Society, Washington, DC, USA, 1214–1219.

[40] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, et al. 2018. Ray: A Distributed Framework for Emerging AI applications. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*. USENIX Association, Berkeley, CA, USA, 561–577.

[41] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. 2017. Hipster: Hybrid Task Manager for Latency-critical Cloud Workloads. In *IEEE International Symposium on High Performance Computer Architecture (HPCA 2017)*. IEEE Computer Society, Washington, DC, USA, 409–420.

[42] Georgios Papoudakis, Filippos Christianos, Arrasy Rahman, and Stefano V. Albrecht. 2019. Dealing with Non-stationarity in Multi-agent Deep Reinforcement Learning. https://arxiv.org/abs/1906.04737

[43] 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)*. USENIX Association, Berkeley, CA, USA, 805–825.

[44] 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 7th International Workshop on Serverless Computing (WoSC7) 2021*. Association for Computing Machinery, New York, NY, USA, 1–8.

[45] 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 15th European Conference on Computer Systems (EuroSys 2020)*. Association for Computing Machinery, New York, NY, USA, 16 pages.

[46] Naci Saldi, Tamer Basar, and Maxim Raginsky. 2018. Markov–Nash Equilibria in Mean-field Games with Discounted Cost. *SIAM Journal on Control and Optimization* 56, 6 (2018), 4256–4287.

[47] Naci Saldi, Tamer Başar, and Maxim Raginsky. 2019. Approximate Nash Equilibria in Partially Observed Stochastic Games with Mean-field Interactions. *Mathematics of Operations Research* 44, 3 (2019), 1006–1033.

[48] Jerome H Saltzer, David P Reed, and David D Clark. 1984. End-to-end Arguments in System Design. *ACM Transactions on Computer Systems (TOCS)* 2, 4 (1984), 277–288.

[49] 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)*. Association for Computing Machinery, New York, NY, USA, 804–811.

[50] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017), 12 pages. http://arxiv.org/abs/1707.06347

[51] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys 2013)*. Association for Computing Machinery, New York, NY, USA, 351–364.

[52] 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)*. Association for Computing Machinery, New York, NY, USA, 1063–1075.

[53] 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 *Proceedings of the USENIX 2020 Annual Technical Conference (ATC 2020)*. USENIX Association, Berkeley, CA, USA, 205–218.

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

[55] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A Scalable Low-latency Serverless Platform. In *Proceedings of the 12th ACM Symposium on Cloud Computing (SoCC 2021)*. Association for Computing Machinery, New York, NY, USA, 138–152.

[56] 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)*. IEEE Computer Society, Washington, DC, USA, 1–10.

[57] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, USA.

[58] Ming Tan. 1993. Multi-agent Reinforcement Learning: Independent vs. Cooperative Agents. In *Proceedings of the 10th International Conference on Machine Learning (ICML 1993)*. PMLR, Cambridge, MA, USA, 8.

[59] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling Quality-of-Service in Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC 2020)*. Association for Computing Machinery, New York, NY, USA, 311–327.

[60] Paul Turner, Bharata B Rao, and Nikhil Rao. 2010. CPU bandwidth control for CFS. In *Proceedings of the Linux Symposium.* Linux Symposium, Ottawa, Ontario, Canada, 245–254.

[61] 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)*. Association for Computing Machinery, New York, NY, USA, 559–572.

[62] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-learning. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI 2016)*, Vol. 30. AAAI Press, Palo Alto, CA, USA, 13 pages.

[63] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC 2013)*. Association for Computing Machinery, New York, NY, USA, Article 5, 16 pages.

[64] Tokala Veni and Mary Saira Bhanu. 2016. Auto-scale: Automatic Scaling of Virtualized Resources using Neuro-fuzzy Reinforcement Learning Approach. *International Journal of Big Data Intelligence* 3, 3 (2016), 145–153.

[65] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys 2015)*. Association for Computing Machinery, New York, NY, USA, 1–17.

[66] 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*. USENIX Association, Berkeley, CA, USA, 133–146.

[67] Yawen Wang, Daniel Crankshaw, Neeraja J. Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. 2022. SOL: Safe on-Node Learning in Cloud Platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 622–634.

[68] Zhaojie Wen, Yishuo Wang, and Fangming Liu. 2022. StepConf: SLO-aware Dynamic Resource Configuration for Serverless Function Workflows. In *IEEE Conference on Computer Communications (INFOCOM 2022)*. IEEE Computer Society, Washington, DC, USA, 1–10.

[69] Annie Xie, James Harrison, and Chelsea Finn. 2021. Deep Reinforcement Learning amidst Lifelong Non-stationarity. In *Proceedings of the 38th International Conference on Machine Learning (ICML 2021)*. PMLR, Cambridge, MA, 1–11.

[70] Yaodong Yang, Rui Luo, Minne Li, Ming Zhou, Weinan Zhang, and Jun Wang. 2018. Mean Field Multi-agent Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*. PMLR, Cambridge, MA, USA, 5571–5580.

[71] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A Native Serverless System for Low-latency, High-throughput Inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 768–781.

[72] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. 2019. MIRAS: Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows. In *IEEE 39th International Conference on Distributed Computing Systems (ICDCS 2019)*. IEEE Computer Society, Washington, DC, USA, 122–132.

[73] 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 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 167–181.

[74] Chao Yu, Akash Velu, Eugene Vinitsky, Yu Wang, Alexandre Bayen, and Yi Wu. 2021. The Surprising Effectiveness of PPO in Cooperative, Multi-agent Games. https://arxiv.org/abs/2103.01955

[75] Hanfei Yu, Athirai A. Irissappane, Hao Wang, and Wes J. Lloyd. 2021. FaaSRank: Learning to Schedule Functions in Serverless Platforms. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS 2021)*. IEEE Computer Society, Washington, DC, USA, 31–40.

[76] 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.

[77] Anastasios Zafeiropoulos, Eleni Fotopoulou, Nikos Filinis, and Symeon Papavassiliou. 2022. Reinforcement Learning-assisted Autoscaling Mechanisms for Serverless Computing Platforms. *Simulation Modelling Practice and Theory* 116 (2022), 17 pages.

[78] 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.

[79] 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* 325 (2021), 321–384.

[80] Jieyu Zhao and Jurgen Schmidhuber. 1996. Incremental Self-improvement for Life-time Multi-agent Reinforcement Learning. In *Proceedings of the 4th International Conference on Simulation of Adaptive Behavior.* Citeseer, Cambridge, MA, USA, 516–525.

[81] Lianmin Zheng, Jiacheng Yang, Han Cai, Ming Zhou, Weinan Zhang, Jun Wang, and Yong Yu. 2018. Magent: A Many-agent Reinforcement Learning Platform for Artificial Collective Intelligence. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI 2018)*, Vol. 32. AAAI Press, Palo Alto, CA, USA, 2 pages.