Future Generation Computer Systems 173 (2025) 107873

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A deep reinforcement learning based algorithm for time and cost optimized scaling of serverless applications

Anupama Mampage *, Shanika Karunasekera, Rajkumar Buyya

The Quantum Cloud Computing and Distributed Systems (qCLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

ARTICLE INFO

Keywords:
Serverless computing
Resource cost efficiency
Function scaling
Function latency
Reinforcement learning

ABSTRACT

Serverless computing has gained a strong traction in the cloud computing community in recent years. Among the many benefits of this novel computing model, the rapid auto-scaling capability of user applications takes prominence. However, the offer of adhoc scaling of user deployments at function level introduces many complications to serverless systems. The added delay and failures in function request executions caused by the time consumed for dynamically creating new resources to suit function workloads, known as the cold-start delay, is one such very prevalent shortcoming. Maintaining idle resource pools to alleviate this issue often results in wasted resources from the cloud provider perspective. Existing solutions to address this limitation mostly focus on predicting and understanding function load levels in order to proactively create required resources. Although these solutions improve function performance, the lack of understanding on the overall system characteristics in making these scaling decisions often leads to the sub-optimal usage of system resources. Further, the multi-tenant nature of serverless systems requires a scalable solution adaptable for multiple co-existing applications, a limitation seen in most current solutions. In this paper, we introduce a novel multi-agent Deep Reinforcement Learning based intelligent solution for both horizontal and vertical scaling of function resources, based on a comprehensive understanding on both function and system requirements. Our solution elevates function performance reducing cold starts, while also offering the flexibility for optimizing resource maintenance cost to the service providers. Experiments conducted considering varying workload scenarios show improvements of up to 23% and 34% in terms of application latency and request failures, or alternatively saving up to 45% in infrastructure cost for the service providers.

1. Introduction

Serverless computing has been embraced as an application deployment model favorable for many application domains in the current world [1]. The provider centric resource management model has succeeded in attaining the “serverless” nature of operations for the end user. However, the cloud provider is tasked with numerous added responsibilities as never before in achieving this seemingly “serverless” behavior of cloud systems. Rapid auto-scalability of user applications in line with load variations, is among the highly valued distinguishing properties of a serverless computing platform, which has proven useful under many application scenarios. The very fine-grained auto-scaling capabilities in serverless platforms require deployed functions to scale their resources just-in-time, as user demand varies. As such, function resources would scale to zero, when there is no request traffic and scale back up when needed, ensuring high resource efficiency. Setting up new resources in this manner on the go, results in a considerable start up time, widely known as the problem of the ’cold start delay’ in functions which hinders its performance. Cold start delay in essence, is a combination of the function runtime environment set up time (sandbox creation, runtime dependency installation) and the time spent on application specific code initialization. This initial delay becomes specially significant for serverless functions with very low execution times, which is the majority. The situation is further complicated by the existence of multiple user applications deployed on the same infrastructure, which require individual attention in their scaling decisions.

A number of existing works have studied the auto-scaling techniques employed by both the commercial and open source serverless computing platforms, and how they affect application performance [2,3]. [4] compares AWS Lambda, Google Cloud Functions and Microsoft Azure in terms of their function cold start delay. These platforms maintain idle function instances from previous executions for a particular time duration before recycling, in order to have more ready-to-serve warm

* Corresponding author.
E-mail address: mampage@student.unimelb.edu.au (A. Mampage).

https://doi.org/10.1016/j.future.2025.107873
Received 1 May 2024; Received in revised form 12 April 2025; Accepted 19 April 2025
Available online 5 May 2025
0167-739X/

instances for new executions. As per the study results, AWS and Google have relatively stable cold start delays while Azure platform showed more varying values at the time of their experimentation. Relationships also exist between factors such as the programming language used and the memory size of a function instance and the resulting resource start up delays. The majority of open source serverless frameworks including Fission [5], Kubeless [6], OpenFaas [7] and Knative [8] are built utilizing Kubernetes [9] as the function orchestrator [10]. The auto-scaling functionality of these frameworks is usually based on a set resource utilization threshold of the existing function instances or custom metrics such as the number of requests per second, which determines the required number of function replicas required to meet the current load.

Research works which address the issues related to serverless auto-scaling delays are identified under two categories. One set of solutions is directed towards reducing the frequency of the occurrence of cold start delays, while the other is focused on reducing the measured cold start delay of an individual function instance [3]. In order to reduce the delay itself, various techniques are presented to improve sandbox creation times, including the creation of the required network elements beforehand, utilizing snapshots of previously used containers and designing and developing customized sandbox environments [11–13]. On the other hand, minimizing the frequency of cold starts is achieved by employing techniques for creating pre-warmed containers, reusing warm containers and adjusting the level of concurrently served requests by a function instance. These approaches often times try to predict the arrival rates and demand levels for individual functions in order to proactively create the required resources [14–16]. The techniques used for such predictions mostly incorporate the resource consumption characteristics of the serverless functions in order to determine the size of the resource pool to be maintained. They rarely consider the cluster resource availability status or the cost of maintaining such idle resource pools to the serverless resource provider. The distinguished billing model in serverless platforms favors its end users by charging them only when the resources are actively being used with a millisecond level accuracy. This means that even though additional resource pools are maintained to meet Quality of Service (QoS) requirements of the user, the provider is able to recover the costs of such resources only to the extent of them being used, calculated at a very fine level. As such, careful calculations are required considering the status of the platform resources along with function characteristics, in order to make these scaling decisions. Moreover, the majority of solutions are limited in their capability of handling multi-tenancy in the function scaling process, since their solutions are designed to handle scaling for specific application functions.

Considering the above highlighted challenges, our work is focused on carrying out the scaling of function resources of multiple user applications in a way that would enhance application performance, while at the same time, preserving the optimum usage of cloud provider resources. Further, while existing solutions are designed to support only horizontal scaling of function instances, i.e., scaling in or out the number of function replicas, our solution approach encapsulates both horizontal and vertical scaling, for better optimizing our target objectives. Vertical scaling handles scaling up and down of the cpu and memory capacities of the function resources. In addition to varying the number of function instances to meet changing user request rates, adapting the resource configuration of existing function instances to handle the incoming traffic in this manner helps in balancing our dual objectives of function performance and provider cost optimization.

Deep Reinforcement Learning (DRL) techniques are being extensively explored for cloud resource management work from recent times. Experience based learning encouraged in the RL paradigm makes it a good candidate as a method of learning the behavior of dynamic serverless workloads with very short execution durations. In this work, we propose a DRL based solution which employs multiple learning agents to determine the optimum level of function scaling to suit changing demand levels. The key **contributions** of our work are as follows:

1. We formulate and present a RL based model of the function auto-scaling problem in a multi-tenant serverless computing environment.
2. We propose a novel multi-agent function scaling framework based on the policy gradient algorithm Asynchronous Advantage Actor Critic (A3C), which aims to attain a balance in optimizing application performance and provider resource cost. In essence we focus on the reduction in function response time and the infrastructure cost. We adapt the A3C algorithm to suit a multi-discrete action space required in making the horizontal and vertical scaling decisions for a multitude of user applications residing in the platform at a time.
3. We train and evaluate our DRL model in a python based simulator environment. We also design a practical testbed based on the open-source serverless platform Kubeless which is deployed on a Kubernetes cluster. The simulator replicates the characteristics and behavior of the practical testbed and utilizes function profiling data derived from the same, in all its experiments.
4. We evaluate and compare our approach with baseline scaling techniques using real world serverless applications, together with function traces captured from Microsoft Azure Functions.

The rest of the paper is organized as follows: Section 2 reviews relevant literature. Section 3 presents the system model and formulates the function scaling problem mathematically. Section 4 introduces the proposed DRL oriented scaling framework. Section 5 and Section 6 discuss the design and implementation details of the DRL agent training environment, evaluation of the proposed technique and the scope for future work.

2. Related work

2.1. Serverless resource scaling

Scaling of serverless functions could be discussed in terms of the horizontal and vertical scaling aspects. Horizontal scaling refers to varying the number of instances of a particular function that is available for request execution. As demand levels vary for an application with time, determining the optimum level of replica scaling required to meet the target objectives is a challenging task. [14] try to predict the required number of function instances in order to keep the new request waiting time below a set threshold, by using a non-cooperative heuristic technique for resource allocation. They propose pre-warming and reuse of containers to reduce response time. [15] use an exponentially weighted moving average model to estimate request arrival rates. Proactive allocation of sandboxes is done using this estimate. An oversubscribed static resource pool with pre-warmed containers of all resource sizes is proposed in [16]. [19] implement a lightweight middleware which uses the knowledge of function compositions to trigger cold starts, leading to provisioning of new containers before they are required. They embrace the principle that a process which undergoes cold start at one step is likely to encounter cold starts in the future too. A container management system with three distinct queues is introduced in [20]. First is occupied by cold containers, the second hosts the warm containers, while the third contains a copy of warm containers for each type of request currently being processed. [21] tries to mitigate the cascading effects of serverless cold starts. They propose just in time resource provisioning techniques specially focused on chained function executions. [17,18,24] propose maintaining a pool of function instances to face request demands. A heuristic solution is given in [27] to adjust the replica number without compromising on user budget. Time-series forecasting is used in [28] to determine the request workload to support the scaling decisions.

Q-Learning based approaches are used in [22,23,29] to determine the number of function containers to scale-up/down at each point in time in order to maintain low application latency and failure rates. [30]

Table 1
Summary of literature review.

Work	Application Model		Scaling	Scaling Type		Decision Parameters				Multi	VM
	Single Function	Function Chain	Technique	Horizontal	Vertical	Optimization Objective		Workload Awareness	Overall System Awareness	Tenancy	Heterogeneity
						Response Time	Provider Cost Efficiency				
[14]	✓		Heuristic	✓		✓	✓	✓		✓	
[16]	✓		Heuristic	✓		✓					
[17]	✓		Heuristic	✓		✓					
[18]		✓	ML	✓		✓	✓	✓			
[19]		✓	Heuristic	✓		✓		✓			✓
[20]	✓		Heuristic	✓		✓		✓		✓	
[21]		✓	Heuristic	✓		✓	✓	✓			
[22]	✓		Q-Learning	✓		✓					
[23]	✓		Q-Learning	✓		✓					
[15]	✓	✓	Mathematical modeling	✓		✓				✓	
[24]	✓		Mathematical modeling	✓		✓		✓		✓	
[25]	✓		PPO		✓	✓		✓		✓	
[26]	✓		Q-Learning		✓	✓					
[27]	✓		Heuristic	✓		✓		✓		✓	
[28]	✓		Mathematical Modeling	✓		✓		✓			
[29]	✓		Q-Learning/DQN	✓		✓					
[30]	✓		Q-Learning	✓		✓		✓			✓
[31]	✓		A2C/Mathematical Modeling	✓		✓		✓			
[32]	✓		MA-PPO	✓	✓	✓		✓		✓	
[33]	✓		Q-Learning/Heuristic	✓	✓	✓		✓			
Our proposed work	✓	✓	MA-A3C	✓	✓	✓	✓	✓	✓	✓	✓

use Q-Learning to decide the optimum level of maximum cpu usage in a function instance to trigger scaling. In [31] the DRL algorithm A2C is used to determine the idle time window for a used function instance and further a time series model is used to predict future invocations and thereby, create warm containers.

Vertical scaling deals with the up/down scale of the resource capacities of a function instance. This is seen as an alternative or used in conjunction with horizontal scaling in order to meet intended targets, in the face of changing traffic levels. An actor–critic architecture with Proximal Policy Optimization (PPO) is used in [25] to harvest idle resources from functions and direct them to under-provisioned instances. A Q-Learning based solution is given in [26] to identify the level of concurrency, i.e the number of concurrent requests served per instance, to optimize function latency and system throughput. A DRL based multi-agent (MA) solution is analyzed against a single agent implementation in [32] for the horizontal and vertical scaling of functions. They focus on function latency and resource efficiency for users and thereby lack focus on the overall platform resource utilization. A preliminary study is done in [33] on using Q-Learning for horizontal scaling decisions and a heuristic approach for vertical scaling.

2.2. RL solutions for serverless resource management

As also discussed above in section II(A), a number of recent works employ RL techniques for enhancing resource management in serverless environments. The target areas of improvement in this manner include resource scheduling, scaling and modeling of optimum resource configurations for functions.

A policy gradient algorithm is proposed in [34], to identify the best node for scheduling a function request. [35] uses a DRL approach to determine the percentage of user requests to be processed by the cloud and offloaded to the fog layer. A distributed task scheduling approach is presented in [36] for serverless edge computing networks. They explore a multi-agent dueling Deep Q Learning (DQN) architecture to assist the edge network in making resource allocation and scheduling decisions. A distributed, experience-sharing, function offloading framework for the edge is proposed in [37]. They suggest an improved actor–critic algorithm for deciding whether to execute functions on the IoT device or on an edge device. [38] introduce a multi-agent DRL solution for caching packages required for running serverless functions at edge nodes, based on their importance and popularity. They aim to improve per function response time while managing resources consumed while caching. A multi-step DQN based solution is proposed in [39] for function scheduling, in a multi-tenant serverless environment, which aims to optimize application performance as well as provider resource cost.

We summarize the reviewed works specifically in the area of serverless resource scaling in Table 1. This comparison considers the aspects of application model, used technique, type of scaling, optimization objective, workload-awareness (consideration for request arrival rate fluctuations), system awareness (knowledge on individual cluster VM resource usage metrics), multi-tenancy (adaptability to suit multiple concurrent applications) and VM heterogeneity. Majority of existing works propose solutions based on only one aspect of scaling, either horizontal or vertical, which is not ideal for optimizing resource cost. Thus such solutions lack attraction to service providers to be used in their serverless platforms. Further, many solutions lack overall system status awareness, and also are not scalable for decision making under a multi-tenant scenario. Our work contrasts with these existing works in a few major aspects: the consideration of the dynamic function workload characteristics as well as system parameters, rather than the simple function resource requirements, the adaptability of the developed solution to suit multi-tenant clusters, the dual optimization objectives concerning both the user and the provider. The ability to incorporate such a vast array of state parameters as decision vectors and also strike a balance between multiple conflicting objectives has been made possible by the multi-agent DRL model introduced in the next sections.

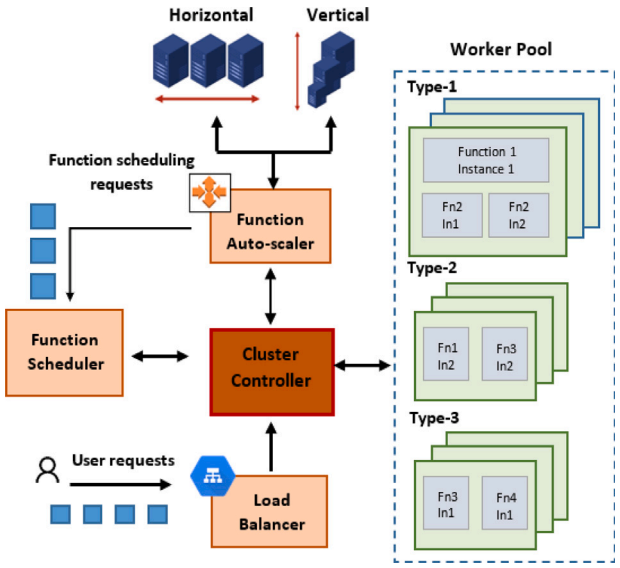


Fig. 1. The system model of the serverless application execution environment.

3. Adaptive function scaling

3.1. System model

Our system model represents the common serverless system architecture in use across a majority of open-source serverless frameworks [5,7,8]. The main components of this architecture include, a cluster of Virtual Machines (VMs), a load balancer acting as the entry point for user requests, a function auto-scaler, a function scheduler and a controller which coordinates the communication between all the other functional units. The highlevel system model is illustrated in Fig. 1.

We consider our worker pool to be composed of a set of heterogeneous VMs with varying compute and memory capacities. An instance of a function is deployed in a container and managed as a single resource unit called a pod. A pod is able to serve multiple concurrent function requests depending on its resource capacity. Creation of replicas of the same function type and adjusting the resource configuration of an existing instance is triggered by the auto-scalar depending on the implemented technique of scaling, which is the focus of this work. Subsequently, the function scheduler selects a suitable node and schedules the created instance.

User requests enter the system via the load balancer, which queues them and directs them to existing function replicas. In the absence of suitable resources, the requests are dropped after the passage of a certain time duration after arrival. The load balancer is considered to be distributing the incoming requests among the existing replicas in a round-robin manner. Serverless applications are formed of either a single or multiple functions. Multi function applications are composed of chained functions which are executed sequentially. Thus, an instance of a function may serve requests of multiple user applications. Requests are received at the deployed functions in a stochastic manner, and thus the demand levels for a function instance could vary rapidly in a very short time. Performance degradation caused by cold start delays are imminent, if the auto-scaling mechanism is not capable of pro-actively determining the scale up of resources required on time. At the same time, gross over-estimation of resources and over-provisioning of the same, lead to massive inefficiencies in resource maintenance cost for the provider. Thus at a given time, the auto-scaler needs to decide the ideal number of replicas and the resource configuration of each replica of a particular function, that would help reach a satisfactory balance in function performance and provider cost.

3.2. Problem formulation

Suppose N is the total number of VMs in a serverless cluster. Each VM is of varying size in terms of its CPU (number of vCPU cores) and memory (MB) capacity. Q is the total number of different function types deployed in the cluster. Let P^k and Req^k denote a pod/instance and a single request of the kth ($k \in [1, Q]$) function respectively, while $M^k(t)$ is the number of existing pods of the same at time t. Each function instance of type k has four attributes at time t, i.e., allocated pod CPU ($P^k_{cpu}(t)$), pod memory ($P^k_{mem}(t)$), average CPU and memory consumption of a single request (Req^k_{cpu} and Req^k_{mem}), standard response time (r^k_0) and the arrival rate of requests of the type. The standard response time for a request refers to the average request response time for a function when executed in a pre-created pod without any resource creation delays.

Compute power is often identified to be a main source of resource pressure in serverless functions, leading to poor application performance [40]. Based on this logic, the default horizontal auto-scaler in our system model triggers a new pod creation and scaling down of existing pods based on a target average CPU utilization of a pod of that type, $T^k_{cpu.util}(t)$ i.e., if the number of new pods of type k to be created is N^k_Δ and the maximum allowed number of pods of any type at any time is M^{max},

$$N^k_\Delta(t) = min[M^k(t) \times \frac{C^k_{cpu.util}(t)}{T^k_{cpu.util}(t)}, M^{max}] - M^k(t) \tag{1}$$

where $C^k_{cpu.util}(t)$ refers to the current average pod CPU utilization. The down scaling logic of pods is designed to terminate idle pods starting with those residing in active VMs with the lowest utilization levels in order to release them. Our task in terms of horizontal scaling is to determine the ideal $T^k_{cpu.util}(t)$ value at a given time. Pods which are not currently being used are scaled down as required where $N^k_\Delta(t)$ is a negative value.

Along with the action of the horizontal scaler, the vertical auto-scaler needs to determine the best suited levels of CPU and memory configurations for a function of type k at a given time t. The incremental/decremental CPU value $cpu^k_\Delta(t)$ and the memory value $mem^k_\Delta(t)$ which form the vertical scaling decision, need to meet a few constraints, i.e,

(1) the resulting resource allocation levels after action execution need to be within the upper and lower boundaries of applicable CPU and memory resource limits to a function instance,

$$\begin{aligned} P_{cpu.min} < P^k_{cpu}(t) + cpu^k_\Delta(t) < P_{cpu.max} \\ P_{mem.min} < P^k_{mem}(t) + mem^k_\Delta(t) < P_{mem.max} \end{aligned} \tag{2}$$

We consider these allocated resources for a function instance to be hard limits, i.e., these mark upper limits of resource consumption by a single pod, irrespective of the traffic levels.

(2) The chosen resource increments need to be compatible with the available resource levels of VMs holding the existing function replicas,

$$[P^k_{cpu}(t) + cpu^k_\Delta(t)] \times M^k_{vm^i}(t) < vm^i_{cpu}(t) \qquad \forall i \in [1, N] \tag{3}$$

$$[P^k_{mem}(t) + mem^k_\Delta(t)] \times M^k_{vm^i}(t) < vm^i_{mem}(t) \qquad \forall i \in [1, N] \tag{4}$$

where $M^k_{vm^i}(t)$ is the number of replicas of kth function residing in vm^i, while $vm^i_{cpu}(t)$ and $vm^i_{mem}(t)$ is the available CPU and memory of the same at time t.

(3) The resource configuration change in an instance should not affect the function requests already in execution. Thus the new resource allocation should not go below the current resource utilization levels of any pod of the type.

$$P^{kj}_{cpu.util}(t) < [P^k_{cpu}(t) + cpu^k_\Delta(t)] \qquad \forall j \in [1, M^k(t)] \tag{5}$$

$$P^{kj}_{mem.util}(t) < [P^k_{mem}(t) + cpu^k_\Delta(t)] \qquad \forall j \in [1, M^k(t)] \tag{6}$$

Table 2
Definition of symbols.

Symbol	Definition
N	Total number of available VMs
Q	Total number of deployed functions
P^k_{cpu}	Allocated CPU for a pod of the kth function, $k \in [1, Q]$
P^k_{mem}	Allocated memory for a pod of the kth function, $k \in [1, Q]$
$M^k_{vm_i}$	Number of existing pods of the kth function residing in vm_i
M^{max}	Maximum allowed number of pods of any single function type
$P^{kj}_{cpu.util}$	Cpu utilization of the jth pod of function k, $j \in [1, M^k]$
$P^{kj}_{mem.util}$	Memory utilization of the jth pod of function k, $j \in [1, M^k]$
$T^k_{cpu.util}$	Target average CPU utilization of a pod of type k
$C^k_{cpu.util}$	Current average CPU utilization of a pod of type k
N^k_Δ	Number of new pods of type k to be created
cpu^k_Δ	Change in allocated CPU to a pod of type k
mem^k_Δ	Change in allocated memory to a pod of type k
vm^i_{cpu}	Available cpu in vm_i
vm^i_{mem}	Available memory in vm_i
A	Total number of user applications deployed
V^b	Number of user requests received by application b, $b \in [1, A]$
R^b_q	The sum of response times of each function relevant to the q^{th} request of an application, $q \in [1, V^b]$
R^b_{q0}	Total standard response time of the constituent functions of the q^{th} request of an application
$price_i$	Unit price of VM, v_i
t_i	Total active time of VM, v_i

where $P^{kj}_{cpu.util}$ and $P^{kj}_{mem.util}$ are the cpu and memory utilization levels of the jth pod of function k. The time t in the above expressions: (1), (2), (3), (4), (5), and (6) indicates the time steps in which scaling decisions are taken.

3.3. Optimization objectives

One target objective of this work is to minimize sub-optimal application performance caused by the lack of a proper resource scaling strategy. As mentioned previously, the resources allocated to function instances are set as hard limits, which prevents them from causing resource contention in the host node, with increased traffic levels. Thus, we could consider that the performance degradation of applications under such a system model is a direct effect of the absence of enough ready resources to face request demand levels. Cold start delays introduced by new resource creation affect request response times and may also cause request failures. Hence we consider application response time latency and request failure rates to be metrics which directly reflect the effects of the platform scaling decisions.

Let A be the total number of user applications deployed in the platform. Consider V^b, $1 \le b \le A$ to be the total request traffic to bth application. The sum of response times of the constituent functions corresponding to the q^{th} request of application b ($1 \le q \le V^b$) is R^b_q. Also, the estimated total standard response time of the same is denoted by R^b_{q0}. We define the ratio of R^b_q and R^b_{q0} averaged over the total number of requests received by an application, as the average Relative Application Response Time (RART). We aim to minimize the average RART, calculated across all the deployed applications over the duration of a workload. Here we use an 'average' response time parameter in the calculation since we want to improve the overall performance of functions running on the platform. Also, we define RART instead of the response time itself, to eliminate any bias in our optimization objective arising from execution time variations in serverless functions.

$$Average\ RART = \frac{1}{A}\sum_{b=1}^{A}\frac{1}{V^b}\sum_{q=1}^{V^b}\frac{R^b_q}{R^b_{q0}} \tag{7}$$

The sum of response times of each function which form the considered execution sequence of an application, is used for calculating R^b_q and R^b_{q0}. The preceding function in a chained application simply evokes the next function in the sequence. Hence this calculation is possible as this process is devoid of any data communication delay. The total

standard response time for an application's request (R^b_{q0}) is expressed as a function of q, since the relevant function sequence is dependent on the request input.

Further, as part of performance optimization, we aim to minimize Request Failure Rates (RFR), i.e., the number of dropped function requests as a ratio of the total requests received. Accordingly we express our performance optimization objective as follows:

$$Minimize : [Average\ RART, RFR] \tag{8}$$

In this work we also plan to optimize the infrastructure cost of the provider. In the calculation of the resource costs we incorporate VM instance pricing, as we consider a heterogeneous serverless cluster composed of VMs of different CPU and memory sizes. The provider cost optimization objective is formulated as follows:

$$Minimize : Cost_{Total} = \sum_{i=1}^{N} price_i \times t_i \tag{9}$$

$price_i$ is the unit price of VM v_i and t_i is the total time that the ith VM was active during workload executions. A VM is considered to be active, when it is serving requests of at least one function. Thus resource efficiency is achieved when active VMs have high utilization levels.

Our primary objectives of minimizing function performance degradation and enabling high resource efficiency tend to be conflicting objectives. Therefore, we utilize a system parameter $\beta \in [0, 1]$, so that users can achieve a sufficient trade-off between the two. Accordingly, our overall target objective is as follows:

$$Minimize : \beta \times Sum\ (Average\ RART + RFR) + (1 - \beta) \times Cost_{Total} \tag{10}$$

The incorporation of both vertical and horizontal scaling decisions in to our solution space ensures that a balance is achieved between the user objectives. Vertical scaling of function resources allows for maintaining higher VM resource efficiency without the need for activating more VMs, when a scale up of resources is needed to elevate application performance. Similarly, as user traffic slows down, leading to a resource down scale, horizontal scaling enables saving up on VM resource cost by terminating idle pods and shutting down unused VMs.

Table 2 summarizes the various symbols introduced in this section.

4. Reinforcement learning model

4.1. Learning model for function scaling

RL is a branch of machine learning that encourages an experience based learning style. A RL agent interacts with its environment and at each time step takes an action a_t, based on the current policy $\pi(a_t|s_t)$, where s_t is the current state of the environment. A reward r_{t+1} is received in turn based on the 'goodness' of the action. The RL agent's final objective is to learn a policy which maximizes the cumulative reward over a sequence of actions.

In this work we explore the applicability of the concept of RL in developing an adaptive scaling policy for applications in a multi-tenant serverless computing environment. The RL agent takes the role of forming the basis of scaling each function, either horizontally or vertically, with variations in user demand levels. The serverless platform forms the environment with which the agent communicates and derives state information at each time step. Time steps in which these scaling configuration changes are executed, are considered to happen at regular intervals. The received reward after each scaling action implementation is dependent on the target level of optimization of each of the dual objectives discussed above. The key aspects of the RL model in the context of our problem are discussed below.

State space: The state information needs to encapsulate both the resource metrics of the serverless platform infrastructure as well as the resource requirements, traffic levels and the current performance of the function to be scaled. Accordingly, the state in our environment could be presented as a 1-dimensional vector, where the first part describes the cluster VM specifications: [$vm^i_{cpu.util}$, $vm^i_{mem.util}$, $vm^i_{cpu.alloc}$, $vm^i_{mem.alloc}$, $vm^i_{cpu.cap}$, $vm^i_{mem.cap}$, $vm^i_{replicas}$]. $vm^i_{cpu.util}$ and $vm^i_{mem.util}$ refer to the actual cpu and memory utilization levels of the VMs, $vm^i_{cpu.alloc}$ and $vm^i_{mem.alloc}$ refer to the percentage of cpu and memory that is allocated to pods, $vm^i_{cpu.cap}$ and $vm^i_{mem.cap}$ denote the cpu and memory capacities (this is representative of the VM unit prices), while $vm^i_{replicas}$ represent the number of replicas of the scaling function, that is currently present in the VM. These VM resource metrics are gathered for all the cluster VMs to form the state space. Note that the resource utilization and allocation levels identify as two separate metrics since although resources are allocated to function pods, the VM resources actually utilized depend on the function requests in execution in those pods. The second part of the state vector is composed of the function specifications: [P^k_{cpu}, P^k_{mem}, Req^k_{cpu}, Req^k_{mem}, P^k_{rate}, P^k_{RFRT}, P^k_{RFR}, $C^k_{cpu.util}$, $C^k_{mem.util}$]. P^k_{cpu} and P^k_{mem} represent the requested cpu and memory by a function instance, Req^k_{cpu} and Req^k_{mem} represent the average resource consumption of a single request of the type, P^k_{rate} represents the current request rate, P^k_{RFRT} represent the Relative Function Response Time (RFRT), which is the ratio of the actual function response time to the standard response time, P^k_{RFR} represent the function request failure rate, while $C^k_{cpu.util}$ and $C^k_{mem.util}$ represent the average cpu and memory utilization of all the pods of type k. These metrics when consolidated, give the RL agent a comprehensive understanding on the current system status, in order to reach the best scaling policy with time. At the start of each time step, the agent gathers the required data and forms this state vector before determining the scaling action.

Action space: We model the action space in our environment as a novel multi-discrete action space where we need to determine three decision parameters namely, the target average cpu utilization value for triggering horizontal function scaling (a_1), the change in allocated cpu (a_2), and memory (a_3) values for pods of the considered function type. Since combining the three actions to formulate an action space with all possible combinations leads to an explosion in the action space size, we consider the three to be independent decision variables. Further we discretize each variable to suit the scale of our modeled environment, where each action would reflect either an increase, decrease or maintaining the same level in the particular variable. Accordingly, a complete action generated by the DRL agent could be presented as [a_1, a_2, a_3].

Reward: The reward assigned to the agent at each step immediately after an action, needs to resonate with the target objectives of optimization. Since the considered objectives of performance and provider cost optimization in this work usually compete with each other and thus could be conflicting, we define two separate reward structures for the two. Accordingly, the reward for action a_t is:

1. R_1: The sum of the average RFRT and RFR of all the deployed functions in the cluster a set time interval after the implementation of action a_t. Since application response time is a function of that of its constituent functions, RFRT acts as a proxy to measure our performance optimization objective of RART in Eq. (7). In addition, it is more closely identifiable with each function scaling decision of the DRL agent.
2. R_2: The difference in the total cluster VM up time cost (Eq. (9)) just before and a set time interval after the implementation of action a_t.

Since the cumulative of both these reward values at the end of an episode needs to be minimized in order to reach our target improvements, we insert a negative sign to motivate reduction in latency and cost over time. Further, at each step we normalize the three values of RFRT, RFR, and VM cost which are in different scales in order to remove any notion of being biased towards one value, in the process of DRL model training. We derive the minimum and maximum values for each of these step rewards after running and observing these values over many workload scenarios. As such, the awarded reward to the agent after each scaling decision is as follows:

$$Reward = -((\beta \times R_{1normalized}) + ((1 - \beta) \times R_{2normalized})) \tag{11}$$

Algorithm 1 Actor–Critic based Multi-agent Scaling Algorithm

Initialize the global shared actor and critic network parameters θ and ϕ
for worker = 1 to N **do**
  Initialize the local actor and critic network parameters θ' and ϕ'
  Initialize the local step counter t = 0
  Initialize the training parameters α, γ and network update frequency f
  Initialize the local training environment for the worker agent
  for episode = 1 to E **do**
    Reset the environment
    for step = 1 to T **do**
      Input the state s of the environment to actor network $\pi_{\theta'}(a|s)$
      for i = 1 to 3 **do**
        Select action a_i using the marginal distribution $\pi_{\theta'_i}(a|s)$
      Execute the combined action $(a = a_1, a_2, a_3)$, move to the next state s' and observe the reward r
      Store the transition (s, a, r, s') in memory D
      if $t\%f == 0$ or step = T **then**
        for j = 1 to K **do**
          Compute the advantage estimates $\hat{A_1}$ to $\hat{A_K}$
          Compute the loss and the gradients of the loss of actor $\nabla_{\theta'} J(\theta')$ and critic $\nabla_{\phi'} J(\phi')$ networks
        Perform asynchronous update of global actor and critic network parameters θ and ϕ
        Synchronize the local actor and critic network parameters θ' and ϕ' with θ and ϕ
        Clear memory D
      $t \leftarrow t + 1$
return

4.2. Actor–critic based multi-agent scaling framework

The objective of a RL algorithm is to find the optimal policy to take actions, which maximizes the cumulative reward over time. The two fundamental methods in RL to find the optimal policy are the value based and policy based methods. The value based methods work by observing the 'Quality' or how good a particular state–action pair is, i.e., by using the Q function. In policy based methods, we find the optimal policy without calculating the Q function. Actor–critic methods take advantage of both the value and policy based methods in finding the optimal policy. In fact, they are proven to be able to overcome many shortcomings of vanilla policy gradient methods. Thus we form the basis of our scaling framework using the actor–critic algorithm.

Actor–critic technique makes use of two neural networks, the actor network and the critic network. The actor helps find the optimal policy $\pi_\theta(a_t|s_t)$, which leads to taking the best action in each state in order to achieve the desired objectives. The critic works in a feedback loop evaluating the policy generated by the actor, leading it to finding the best policy. In essence, the actor network is a policy network which uses a policy gradient method to find the optimal policy, while the critic network is a value network which is trained to estimate the state-value function, $v_\pi(s_t|\phi)$. θ and ϕ are the adjustable parameters of the actor and critic networks respectively.

Actor and critic networks learn by either maximizing their objective functions or by minimizing the loss functions. Accordingly, the actor learns the optimal policy by calculating the policy gradient, i.e., the gradient of the network and periodically updating the network parameter θ using gradient ascent (Eq. (12)).

$$\nabla_\theta J(\theta) = \nabla_\theta log\pi_\theta(a_t|s_t)A(s,a)$$
$$\theta = \theta + \alpha\nabla_\theta J(\theta) \tag{12}$$

where $J(\theta)$ is the objective function which aims to increase the probability of occurrence of the actions which maximize the expected return of a given trajectory. As seen in Eq. (12) above, we calculate the policy gradient in the actor–critic methods using $A(s,a)$, the advantage function, hence the name Advantage Actor Critic (A2C). Expanding the advantage function;

$$\nabla_\theta J(\theta) = \nabla_\theta log\pi_\theta(a_t|s_t)(r + \gamma V_\phi(s'_t) - V_\phi(s_t)) \quad s.t. : (2), (3) \tag{13}$$

Thus, the advantage function reveals how good action a is compared to the average actions in state s. This essentially helps actor–critic methods to overcome inefficiencies of vanilla policy gradient algorithms by reducing the high variance of policy networks and stabilizing the model. Similarly, the critic learns by minimizing the loss of the critic network, i.e., the Temporary Difference (TD) error, which is the difference between the target value of the state $(r + \gamma V_\phi(s'_t))$ and the value of the state predicted by the network. During the course of training, the gradient of the critic network is calculated and the network parameter is updated using gradient descent (Eq. (14)), thus allowing the critic to learn the actual state-value function.

$$J(\phi) = r + \gamma V_\phi(s'_t) - V_\phi(s_t)$$
$$\phi = \phi - \alpha\nabla_\phi J(\phi) \tag{14}$$

DRL techniques trained with a single agent have proven to be able to provide effective solutions for many single function scaling scenarios [22,23,33]. But serverless platforms are usually multi-tenant environments with a number of deployed functions with various resource characteristics co-existing with each other. Further, these different functions have dynamically changing workload patterns, lowering the sample efficiency of many single agent RL solutions in the context of the multi-tenant scaling problem considered in our work. Thus in this work, we explore the applicability of the DRL technique A3C [41], which employs several DRL agents who engage in learning in parallel, and aggregate the overall experience. The process of parallel learning helps explore the combination of state and action spaces much faster.

In A3C we work with two types of networks, the global network and the local or worker networks. Each worker agent interacts independently with its own copy of the environment, and shares the gathered experiences with the global agent asynchronously. Both the worker agents and the global agent follow an actor–critic architecture. Under A3C, in order to encourage sufficient exploration and reaching a global optimum, we add the term 'entropy' to the previously discussed (Eq. (13)) actor loss, i.e.;

$$J(\theta) = log\pi_\theta(a_t|s_t)(r + \gamma V_\phi(s'_t) - V_\phi(s_t)) + \beta H(\pi(s)) \tag{15}$$

where $H(\pi)$ refers to the entropy of the policy while β controls the significance of the entropy. As discussed under section 4.1, we express our action space for scaling as a novel multi-dimensional discrete action space. We then adapt the technique described for discretized multi-dimensional action spaces in [42], to design our actor network architecture.

We assume our normalized initial action space to be $A = [-1, 1]^M$, where M represents the number of action dimensions. If we discretize

Table 3
Worker cluster resource details.

Instance Type	vCPU cores	Memory (GB)	Quantity	Price ($/hr)
m6 g.medium	1	4	5	0.048
t4 g.large	2	8	5	0.0848
t4 g.xlarge	4	16	5	0.1696
t4 g.2xlarge	8	32	5	0.3392

each of these dimensions into K equally spaced actions, the set of atomic actions we get for each dimension i is, $A_i = \{\frac{2j}{K-1} - 1\}_{j=0}^{K-1}$. Then we present the distribution of action space as factorized across dimensions, in order to tackle the curse of dimensionality. As such, we consider a marginal distribution $\pi_{\theta_i}(a_i|s)$ for each dimension i, over the set of actions $a_i \in A_i$, where θ_i is the parameter of the distribution. Accordingly we get a joint discrete policy $\pi_\theta(a|s) = \prod_{i=1}^{M} \pi_{\theta_i}(a_i|s)$, where θ represents the parameter of the actor network which takes state s as input. After layers of transformation, the network outputs the log probability L_{ij} for the jth action in the ith dimension, where $i \in [1, M]$ and $j \in [1, K]$. Finally, for each dimension i, the K logits are combined with soft-max to derive the probability of choosing action j, i.e., $p_{ij} = softmax(L_{ij})$. Note that as per the scaling problem space defined in our work, $M = 3$ and K is chosen suitably for each action dimension. Each actor in our multi-agent framework follows this network architecture.

Algorithm 1 presents the pseudo-code for the multi-agent scaling framework training process flow. We first initialize the global actor and critic network parameters which would be shared among and updated by the worker agents during the training process. Each worker would have its own copy of the environment and separate actor and critic networks (lines 3–6). At the start of each episode, workers reset their local environment. At each time step, the agent retrieves the state information with regard to the platform and the function to be scaled, and feed it to the local actor network. Next, the marginal probability distribution for each action dimension is used to determine the combined action for the current step (lines 11–12). Upon execution of the generated action, the environment transitions to a new state, and the agent receives a reward. All the transition information which includes the environmental state, executed action, awarded reward, and the next state are stored in memory (line 14). If the network update frequency or the maximum step count for an episode is reached, the agent starts the network parameter sharing and update process. First the advantage estimates, the loss and the network gradients are calculated for each transition stored in memory (lines 16–18). Then each worker agent asynchronously updates the global actor and critic network parameters using the calculated gradients. Finally, the local networks are updated with new weights pulled from the global model. After each network update, the local memory is cleared (lines 19–21).

5. Performance evaluation

5.1. RL environment design and implementation

We implement a practical experimental serverless framework on the Melbourne Research Cloud [43] for preliminary data collection related to profiling serverless functions and also for deriving realistic system parameters exhibited during function executions. The testbed comprises of the Kubeless [6] open source serverless framework deployed on a 20 node Kubernetes [9] cluster on top of which the Prometheus [44] monitoring tool is installed for monitoring cluster metrics.

Following the architecture of this practical testbed, we have developed a simulation environment for serverless function execution in *Python*, which also represents the system model presented in section III(A). This environment is integrated with Tensorflow-agents in the backend, which are developed using Keras and Tensorflow(TF) libraries. The key features of our developed simulator environment and the agent training process flow are summarized below:

1. Requests arriving at deployed function instances are loaded to an event queue in the order of arrival at the start of the simulation.
2. In the event that a suitable function instance is unavailable to accommodate an incoming request, the request is queued and subsequently dropped, after multiple scheduling retries at set time intervals.
3. Time steps for scaling decision making for each agent are scheduled at regular time intervals so that the agent's learned policy is capable of supporting proactive scaling of function resources independent of any workload specifics.
4. At each time step of the DRL agent, the serverless environment exposes the cluster state metrics which include the VM resource usage statistics and the workload nature of the function to be scaled.
5. After the execution of each of the combined horizontal and vertical scaling actions, the agent waits for a set time duration for the environment to reflect the action consequences, before deriving the step reward.
6. Each agent in the implemented multi-agent model, follows these steps in parallel, on copies of the same cluster environment.

Although our implemented TF-agents are tasked with optimizing function performance and cluster resource cost, the developed simulation environment is capable of exposing monitoring metrics required for any other extended objectives and facilitating training for continuous action spaces or modified DRL agent architectures. Our RL based serverless environment implementation with TF agents as the back end called 'Serverless_DRL', is available as an opensource software.[1]

5.2. Experimental settings

5.2.1. Cluster setup

Our simulated VM cluster comprises of 20 heterogeneous VMs of 4 different vCPU and memory configurations. The clock speeds of the CPU cores were set to be similar to that of VMs in AWS Lambda serverless platform as identified in [4]. We use the AWS instance pricing of EC2 VMs (in Australia) [45] closely matching the clock speed, vCPU and RAM configurations, as our pricing model. These cluster resource details are summarized in Table 3. Our practical testbed too follows these VM cpu and memory configurations. We conduct our experiments under two scenarios, letting the multi-agent model to be comprised of 3 and 5 parallel actor-learners (agents) under each scenario, in order to observe the training time and data efficiency in state and action space exploration with more agents. Each individual agent works in a cluster environment of similar configuration as above during training.

5.2.2. Workload specifications

Serverless Applications: We choose 12 benchmark applications from ServiBench [46] and FunctionBench [47] benchmark suites, which are formed of either a single or a chain of functions. Each of these applications have varying demands on CPU and memory resources based on their constituent functions. Thus their diverse sensitivities to different horizontal and vertical actions in the action space provide a good learning experience for the DRL agents. We use our practical setup for conducting function profiling for all the selected applications. An instance of each individual function is deployed on a VM in isolation and the JMeter [48] load generation tool is used for sending a series of user requests to this instance. The results obtained from this tool and the cluster data recorded by Prometheus are averaged across multiple such workload executions to determine the resource consumption of a single function request ($Req^k_{cpu}(t) and\ Req^k_{mem}(t)$), standard response time (r^k_0) of a request and the instance creation time. In the absence of resources for profiling an application, knowledge on proxy functions with similar resource requirements could be used. However, the more accurate the function profiling is, the more optimal the model results would be. P^k_{cpu} and P^k_{mem} for each function is initially set as the resources required to handle a defined number of requests. Table 4 captures the resource utilization profile of these benchmark applications.

[1] https://github.com/Cloudslab/Serverless_DRL

Table 4
Serverless application details.

Name	Resource Sensitivity		# of Functions
	CPU	Memory	
Primary	High	High	1
Float	High	High	1
Matrix Multiplication	High	High	1
Linpack	High	High	1
Load	low	low	1
Dd	High	Medium	1
Gzip-compression	High	Medium	1
Thumbnail Generator	Low	Medium	2
Facial Recognition	Medium	Medium	5
Todo API	Low	Low	5
Image Processing	Medium	Medium	2
Video Processing	High	High	2

Workload Creation: We leverage function traces from the publicly available data set from Microsoft Azure's Serverless Platform [49] in order to derive request arrival patterns when creating the function workloads for both training and evaluating the DRL agents. In all the experiments, we maintain request arrival rates at 10–60 requests per second, maximum compute power and memory allocated to a single function instance at 1 vCPU core and 3 GB respectively and the execution time of a request below 10 s. These thresholds are defined to suit the scale of our cluster setup. Accordingly, we analyze the Azure function data collected over a 24 h period and filter a set of multi and single function applications with these characteristics and extract their request arrival patterns in the workload creation. For each function the per minute request arrival rates recorded in Azure traces for a given function is considered as a per second rate. In a given workload, the function arrival rates for a single function is fluctuated over time using these traces. Multiple such function request loads are combined to form a single workload. A workload consumed by a single agent is incorporated with traffic from no more than 4 functions at a time in order to maintain a sufficient load in the cluster for the training process. During each time step, the function subjected to scaling configuration changes is decided arbitrarily during the training process while the function with the highest RFRT is chosen during model evaluation, in order to attain optimum application performance.

5.2.3. Hyper-parameter configurations

Hyper-parameters for the actor–critic networks of each worker agent are decided on a trial and error basis. The discount factor is maintained at a lower value since the rapidly fluctuating nature of serverless workloads reduce the relevance of distant rewards towards current actions and thus a higher discount factor would force irrelevant information on the agent hindering the learning process. The learning rate for the actor and critic networks is maintained low enough so as not to cause a gradient blowup or lead to a sub-optimal solution too fast, and high enough so as the model converges with sufficient training. Each action dimension is discretized in to 11 actions as described under section IV(B). This was arrived at after a series of initial experiments in order to maintain action space exploration costs at a manageable level while reaching good optimization levels for the target metrics. The maximum number of replicas for a single function at a time was restricted in order to suit the capacity of a 20 VM cluster while an upper limit of CPU scaling threshold was also set to ensure proactive scaling for all functions even at very low traffic levels. The hyper-parameter settings for the A3C agents along with these environmental parameters in use for all the experiments are listed in Table 5.

5.3. Performance metrics

We use three metrics to evaluate the effectiveness of our solution noted below:

Table 5
Hyper-parameters used for DRL model training.

Parameter	Value
General	
Optimization parameter (β)	[0.0, 0.25, 0.50, 0.75, 1.00]
Maximum number of concurrent replicas of a function	80
Maximum pod CPU utilization for horizontal scaling	90%
Neural network parameters	
Discount factor (γ)	0.6
Learning rate (α)	0.0001
No. of input layers	1
No. of output layers	1
No. of hidden layers	2
No. of neurons in each hidden layer	150
Optimizer	Adam
Network update frequency	30
Action space size for each dimension	11

1. **Average Relative Application Response Time ratio (RART)**: The sum of the average, relative response times of all the applications in a workload during an episode, divided by the number of applications, calculated using Eq. (7).
2. **Request Failure Rate (RFT)**: The ratio of the number of dropped function requests to the total number of requests received in an episode.
3. **VM Usage Cost**: The cost of maintaining the VMs active during an episode. The calculation of this metric is as in Eq. (9).

5.4. Baseline scaling techniques

We use four baseline scaling techniques to compare the performance of our proposed solution.

DQN: We use the value based DRL algorithm Deep Q Learning to arrive at a solution for the function scaling problem. Here we consider each combination of the actions from the three discretized action spaces for horizontal scaling, CPU and memory vertical scaling as a compound action that the agent chooses. Due to the state–action space explosion that results from combining actions in this way, we limit the granularity of action discretization to four actions per dimension resulting in 64 compound actions in total.

Knative: The opensource serverless platform Knative [50] allows users to set a target pod concurrency value, limiting the number of concurrent requests handled by a function instance. Further a target utilization value is set determining the actual percentage of the target that we should meet. Horizontal scaling of functions is triggered in order to maintain the set level of request concurrency and the target utilization (we set these at 4 and 75%).

Kube-cpu: Kubernetes default horizontal pod auto-scaler scales function instances based on a set threshold on function level resource usage metrics. Here we consider scaling function replicas in order to maintain the average CPU utilization across all instances of a function at or below the set value (we set the CPU utilization threshold at 50%).

OpenFaaS: The opensource serverless platform OpenFaas [51] offers a mix of three modes of scaling to be used based on the function requirements. For long-running functions which can handle only a limited number of requests at a time, the 'capacity' mode triggers scaling based on the number of in-flight requests (we consider a threshold set at 4). For functions which execute quickly and have a high throughput, the 'rps' mode enables scaling based on the number of requests per second completed by a function replica (threshold set at 8). All the other workloads which do not support the 'capacity' and 'rps' scaling profiles use the 'cpu' mode which triggers scaling based on the average CPU utilization (set at 50%) across pods, similar to Kube-cpu. We dynamically decide on the scaling mode used for each function type

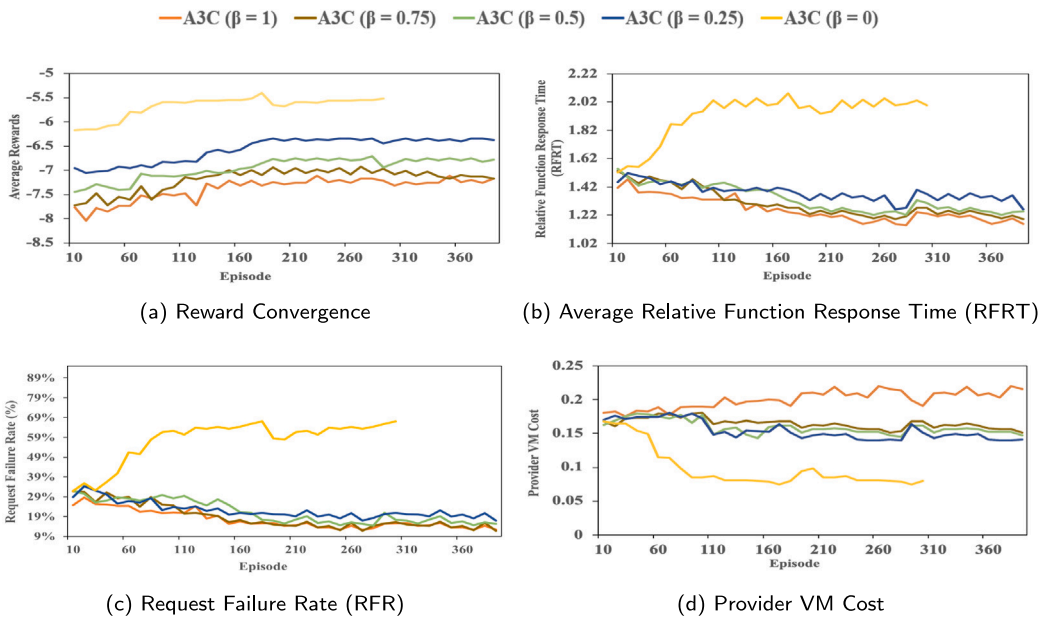


Fig. 2. Training progress of the 3 worker A3C models in terms of reward, average RFRT, request failure rate, and the total VM cost.

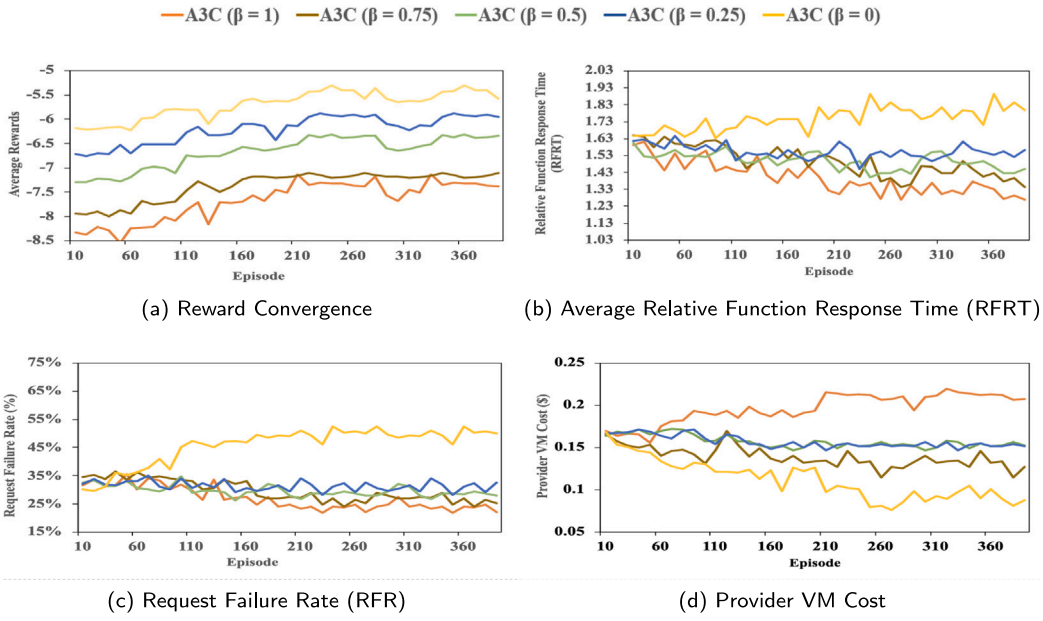


Fig. 3. Training progress of the 5 worker A3C models in terms of reward, average RFRT, request failure rate, and the total VM cost.

based on their execution times and request rates. Accordingly, functions with execution times greater than 2 s are scaled using the 'capacity' mode, functions with less than 2 s execution time and request rate higher than 20 requests per second at the time, use 'rps' mode and the rest are scaled using the 'cpu' mode.

5.5. Convergence of the DRL model

Under each of the multi-agent model scenarios comprising of different number of parallel workers, we train 5 variations of the A3C model considering the significance given to each optimization objective. We use the β parameter to signify the priority assigned to each objective in the agent reward structure. Accordingly, $\beta = 1$ refers to a 100% focus on improving function performance while $\beta = 0$ indicates model training leading to infrastructure cost optimization only. The graphs in Figs. 2 and 3 display the training progress achieved over time as the agents gradually learn to optimize the cumulative reward over an episode. The progress is demonstrated in terms of the episodic reward and the target optimization metrics themselves, i.e: average relative function response time (RFRT), request failure rate (RFR) and the VM cost over each episode. Note that the marked values on the graphs are averaged values over 10 episodes for clarity in presentation.

In the first scenario with 3 actor-learners working in parallel, we train the model with 9 different functions, deployed in the cluster in total. These are chosen to be a mix of functions that are common to multiple applications from Table 4. In the next scenario, we employee 5 actor-learners in the learning process, in order to analyze the achieved efficiency in state space exploration with more worker agents. In this second set of experiments, we deploy 15 different functions

altogether in the cluster and observe the agent behavior leading to model convergence.

Figs. 2(a) and 3(a) show the convergence of the episodic reward under each scenario with varying β parameters. As seen from the graphs, all of the models achieve convergence around the 300th episode, despite the considerably expanded state space size in the second scenario. This is due to the speed-up in data exploration achieved by having more workers learning in parallel, which results in the global model reaching its maximum optimization levels faster, effectively improving both time and data efficiency. Fig. 2(a) also shows that when $\beta = 0$, the model convergence happens relatively faster compared to other scenarios in the set of 3 worker models. Since $\beta = 0$ only incentivizes reducing the VM cost, the agent seems to easily learn to take actions that lead to maintaining the lowest number of replicas possible in the cluster while also limiting their CPU and memory capacities. On the other hand, improving function performance is not as straightforward for the agent to learn, since expanding the pool of function replicas or vertically scaling pod resources would not always lead to the optimum solution. This is because while horizontal scaling creates new resources for request execution, it also adds a resource set up delay which causes increased latency and request failures. Thus a more intelligent strategy needs to be learned depending on the cluster state at each scaling step.

Figs. 2(b), 2(c), 2(d) and 3(b), 3(c), 3(d) represent the average function latency, failure rates and the VM costs incurred over the corresponding training episodes. The $\beta = 1$ graphs in both 2(b) and 3(b) maintain a steady decrease in RFRT with each episode. The $\beta = 0.75$, $\beta = 0.5$, $\beta = 0.25$ graphs too show a gradual decrease in function response time latency since they too are partially motivated to improve function performance in the reward. But understandably, they converge at a higher latency level than when solely focused on optimizing this feature alone. The $\beta = 0$ models on the other hand display a completely opposite trend of increasing latency with each episode as they simply target resource efficiency only, and this easily compromises the performance parameter. The RFR graphs too display a similar trend in all the scenarios. The VM cost graphs show a clear decrease in VM cost over time when $\beta = 0$. The $\beta = 0.25$, $\beta = 0.5$, and $\beta = 0.75$ graphs too show a moderate decline in overall cost for the provider with time. The $\beta = 1$ model converges at a high VM cost as expected, but here we observe considerably lesser prominence than the decline in function performance we earlier saw with the $\beta = 0$ model. A probable cause for this behavior is likely to be the indirect effect that actions leading to improved performance seem to have on enhancing resource efficiency too.

5.6. Analysis of model performance on the evaluation data sets

The performance of our trained multi-agent models is evaluated and discussed mainly in terms of our target optimization objectives of serverless application performance and resource cost efficiency. We extract 1800 function traces in total from Azure function traces using the procedure described in section V(B)(2) in creating the evaluation data set. Our model evaluation is conducted under three request traffic levels as 5–20, 20–40 and 40–60 requests per second, for both the 3 and 5 parallel agent scenarios. Accordingly, for both these scenarios, we create 60 workloads each for the 3 load levels, i.e. a total of 360 workloads. When creating each workload for evaluating the variations of the models trained with 3 agents, we include simultaneous user requests from 5 different serverless applications (single and multi-function) created using the 9 functions used during the training process. Similarly workloads are created for the 5 agent models incorporating requests from applications created using 15 different functions. Further, the request arrival rates for a single application are varied over time in a given workload, each of which spans over five minutes.

Figs. 4 and 5 demonstrate the performance of our A3C models against the baseline scaling techniques under the two agent scenarios. Each bar graph corresponds to the achieved performance metric derived by averaging over the 60 workload runs under each load level.

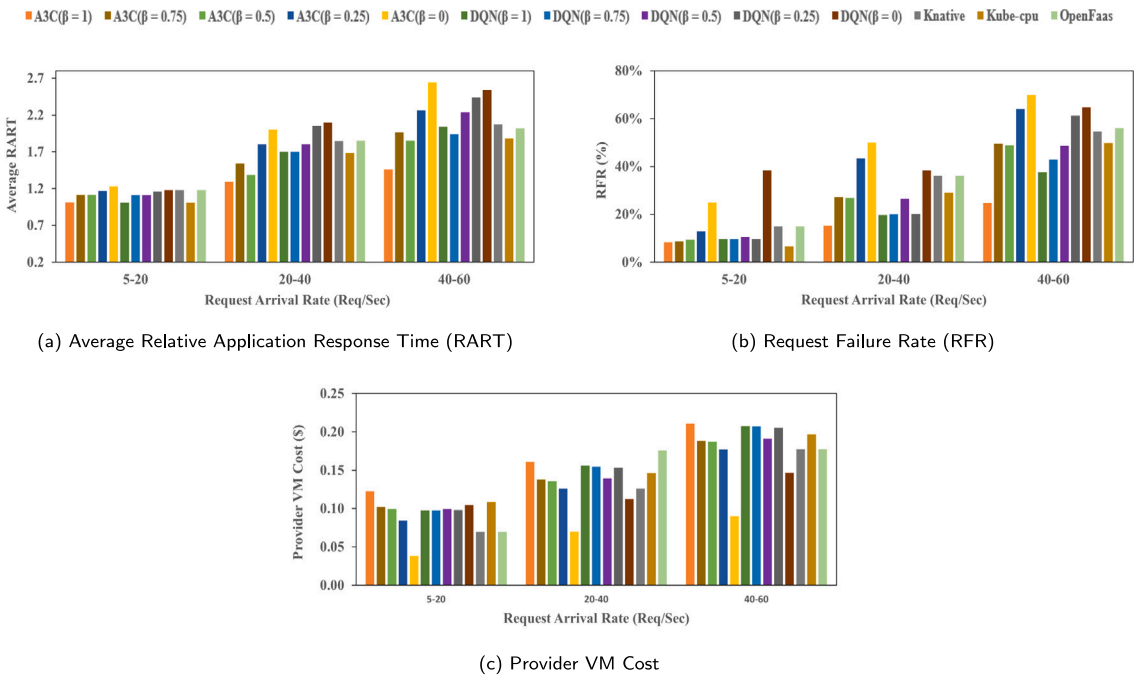
5.6.1. Evaluation of application performance

Application performance is evaluated in terms of RFRT and RFR performance as shown in graphs 4(a), 5(a) and 4(b), 5(b). Overall we can see that the latency and request failure rates increase gradually with the rise in request rates due to increased wait times for request executions arising from resource limitations in the cluster. Also, it is evident from the plotted graphs that the behavior of the models trained using both 3 and 5 actor-learner architectures, is similar in most aspects and thus our discussion below would entail a common analysis for both scenarios for the most part.

At the lowest traffic level of 5–20 req/sec, we do not observe a significant improvement from the A3C($\beta = 1$) model compared to the rest, where the DQN($\beta = 1$) and Kube-cpu models exhibit almost similar or better performance. This is because, at lower traffic levels, the cluster is less congested and thus an intelligent function scaling strategy adds less value to overall performance. However, at high β values, the A3C as well as the DQN models show better function performance as their learned policy favors performance more than cost.

As request rates increase to 20–40 req/sec, a more distinct performance upgrade is seen to be achieved by the trained models. In both the graphs for RFRT, 4(a) and 5(a) and for RFR, 4(b), 5(b), we observe the best performance from the A3C($\beta = 1$) model. The A3C model outperforms the next best performing baseline of Kube-cpu by up to 23% in RFRT and 24% in RFR. Since our models in this case are purely rewarded for better function performance during the training process, they learn to maintain lower cpu thresholds for function scaling, leading to more proactive instance creation. Further, when an existing instance is reaching its maximum utilization levels, the agent learns to vertically scale its capacity after which it could immediately accommodate more requests without any additional wait times. In this process, the agent also learns to weigh between horizontal and vertical scaling as although vertical scaling expands capacity immediately, it limits future resource expansions. Thus if the current cluster load could sustain some delays in resource creation without excessive request failures, horizontal scaling could lead to long term performance benefits. The DQN($\beta = 1$) model exhibits next best performance as it too follows an intelligent scaling policy in contrast to other baselines. However, the DQN model lacks the fine grained learning capability of the A3C model for many reasons. As a single agent model, it lacks the state space exploration capability even when trained for longer periods of time as we observed during model training. Also, since we had to create compound actions out of the 3 action dimensions with high level discretization, the extensiveness of action space exploration too was far weaker compared to the A3C model. The Knative, Kube-cpu and OpenFaas techniques which follow fixed threshold base scaling, do not perform well in a congested resource constrained cluster. They apply a blanket threshold for all the application functions facing varying request rates, which lead to increasingly poor performance as the cluster load rises.

At 40–60 req/sec we see even more distinguished performance improvements in the A3C models with high β values, with up to 34% reduction in request failures when $\beta = 1$. We also note that at times, the A3C($\beta = 0.5$) model shows slightly better latency performance than the A3C($\beta = 0.75$) model under high traffic levels. When the agent is rewarded equally to improve both latency and cost ($\beta = 0.5$), it has indirectly resulted in better function latency than when focused more on latency itself. This is because, scaling actions which lead to efficient cluster resource usage could also result in reduced request wait times and thus latency, which is an added advantage. The DQN model too shows similar behavior in the latency and request failure graphs for DQN($\beta = 1$) and DQN($\beta = 0.75$). The A3C($\beta = 0$) models show worst performance in terms of application performance.

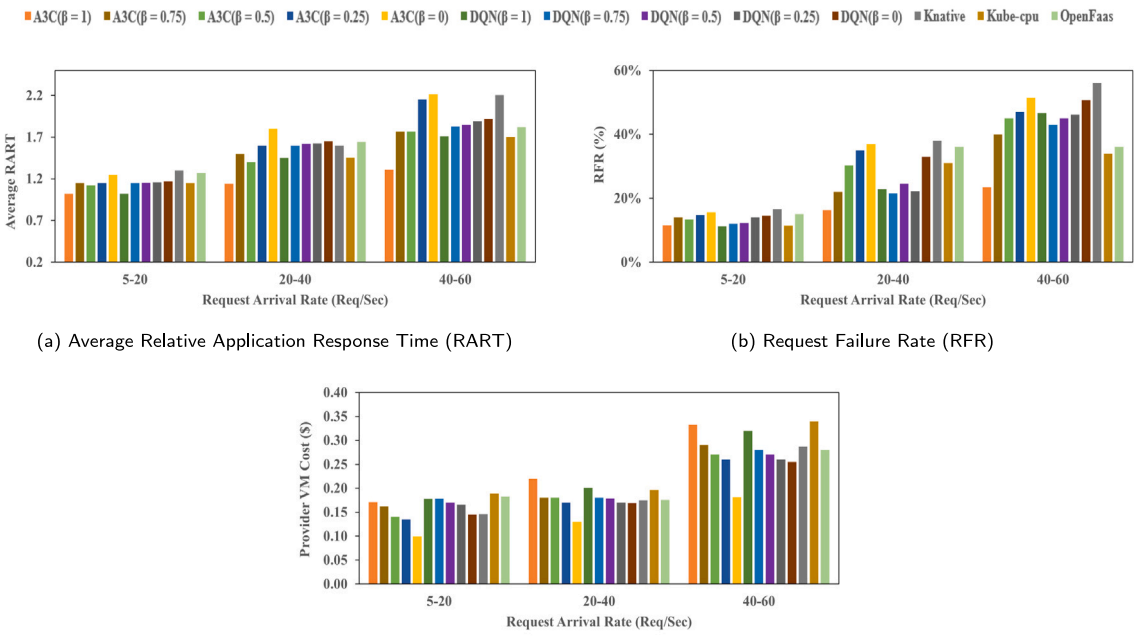


(a) Average Relative Application Response Time (RART)

(b) Request Failure Rate (RFR)

(c) Provider VM Cost

Fig. 4. Comparison of the Average RART, RFR and provider VM cost in the system during an episode, by the 3 worker A3C models and the baseline algorithms.



(a) Average Relative Application Response Time (RART)

(b) Request Failure Rate (RFR)

(c) Provider VM Cost

Fig. 5. Comparison of the Average RART, RFR and provider VM cost in the system during an episode, by the 5 worker A3C models and the baseline algorithms.

5.6.2. *Evaluation of resource cost efficiency*

Resource cost efficiency is evaluated in terms of the cost incurred by the provider to maintain the VMs while they contain running function instances. The overall cost of infrastructure increases as the load levels rise.

In contrast to our observations for latency performance at lower traffic levels, we see clear cost improvements of up to 45% in the A3C($\beta = 0$) models trained for that purpose. This is because with lesser load, if cost is not a concern (i.e. at higher β values), horizontal scaling is encouraged and the cluster could maintain a lot of idling instances. This leads to high VM maintenance costs. On other hand, where resource efficiency is rewarded, the agent learns to take vertical scaling actions more, which leads to higher utilization levels for the active VMs. Subsequently, any idling VMs could be switched off, which saves resource costs. Although not as efficient, the DQN models too show a decreasing trend in cost with β at low load levels, in the second scenario (Fig. 5(c)), which has higher multi-tenancy in the cluster with more applications. Kube-cpu scaling style triggers proactive horizontal scaling without a deeper understanding on the workload patterns, thus leading to large resource inefficiencies.

At 20–40 and 40–60 load levels too we observe a significant improvement in our A3C($\beta = 0$) model, although the opportunity for

gaining a huge resource efficiency level reduces as the cluster utilization levels increase. As expected, the next best performance is seen in the DQN($\beta = 0$) model, as it closely follows the reward structure of the A3C model, falling only short of the state and action space exploration capabilities of the actor–critic architecture. A3C($\beta = 1$), DQN($\beta = 1$) agents exhibit worst performance in terms of cost, closely followed by the Knative, Kube-cpu and OpenFaas techniques which are unable to handle complex load scenarios to achieve a particular target.

5.7. DRL model inference overhead

Model inferencing for our DRL algorithm simply refers to the mapping of current environmental state to an available action based on the state–action values of the pre-trained model. This mapping process is observed to consume a negligible duration of around 40–50 ms on average, which would be the overhead on the function response time.

6. Conclusions and future work

The abstract form of application resource management in serverless computing completely relieves the end users from operational responsibilities. However, cloud providers are still in the process of developing the best strategies to fulfill this new set of responsibilities. As such, attaining an optimum level of scaling for function resources of different applications is still a challenge requiring attention.

In this paper, we proposed a DRL based adaptive solution using the actor–critic architecture, for taking the horizontal and vertical resource scaling decisions for applications in a multi-tenant serverless environment. A successfully scaled application satisfies both the application owner and the infrastructure provider. Accordingly, application performance and the infrastructure maintenance cost for the provider, were considered as our target optimization objectives for DRL model training. Our solution offers flexibility for prioritizing either of these objectives, depending on the user requirements. We conducted and presented details of two sets of experiments in order to observe data and time efficiency improvements achieved, when using different numbers of parallel actor-learners in training our A3C model. Our trained DRL agents were able to take effective scaling decisions for functions deployed in serverless platforms, which led to reduced application latency, request failures and provider side resource wastage. We employed a trained DQN model, along with other baselines to evaluate our presented solution. The results obtained show that our presented intelligent scaling solution greatly benefits all user categories in meeting their objectives.

As part of future work, we plan to explore the efficiency of integrating a time-series regression model with the developed DRL agent architecture for function scaling, in order to forecast request traffic patterns. In the current work, the function applications considered in our experiments are relatively simple and composed of chained functions which run in sequence. When they are run in isolation on a VM, a single request will have a very small resource consumption. Also, they are not designed to utilize multiple cores available in a VM, heterogeneity of VMs has no direct impact on the execution time of a request. By running an application in isolation, we ensure that the VMs are not overloaded. This removes any dependency on the application workload execution time on the CPU or memory availability on a VM. Therefore, as part of future work, we will explore more complex applications in the experiments and consider VM heterogeneity (that is, the capability of VMs with multicores) and their influence on the execution time of functions. Also, we plan to extend our scaling techniques to enable vertical scaling at the individual pod level instead of at function level. Further, we aim to develop a framework for performance and cost modeling of functions, incorporating the performance data obtained in our scaling experiments by varying the allocated resources to function replicas.

CRediT authorship contribution statement

Anupama Mampage: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Shanika Karunasekera:** Writing – review & editing, Supervision, Conceptualization. **Rajkumar Buyya:** Supervision, Resources, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] A. Mampage, S. Karunasekera, R. Buyya, A holistic view on resource management in serverless computing environments: Taxonomy and future directions, ACM Comput. Surv. 54 (11s) (2022) 1–36.
[2] J. Manner, M. Endreß, T. Heckel, G. Wirtz, Cold start influencing factors in function as a service, in: Proceedings of the 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), IEEE, 2018, pp. 181–188.
[3] P. Vahidinia, B. Farahani, F.S. Aliee, Cold start in serverless computing: Current trends and mitigation strategies, in: Proceedings of the 2020 International Conference on Omni-Layer Intelligent Systems, COINS, IEEE, 2020, pp. 1–7.
[4] L. Wang, M. Li, Y. Zhang, T. Ristenpart, M. Swift, Peeking behind the curtains of serverless platforms, in: Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18), 2018, pp. 133–146.
[5] Fission Project, Fission, 2022, (Accessed on 08 April 2022) https://fission.io/.
[6] Kubeless, Kubeless, 2021, (Accessed on 13 Jan 2022) https://kubeless.io/.
[7] OpenFaaS, Home | OpenFaaS - serverless functions made simple, 2022, (Accessed on 08 April 2022) https://www.openfaas.com/.
[8] The Knative Authors, Home - knative, 2022, (Accessed on 08 April 2022) https://knative.dev/docs/.
[9] Kubernetes, 2022, (Accessed on 08 April 2022) https://kubernetes.io/.
[10] S.K. Mohanty, G. Premsankar, M. Di Francesco, et al., An evaluation of open source serverless computing frameworks., CloudCom 2018 (2018) 115–120.
[11] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, V. Sukhomlinov, Agile cold starts for scalable serverless, in: Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19), 2019.
[12] P. Silva, D. Fireman, T.E. Pereira, Prebaking functions to warm the serverless cold start, in: Proceedings of the 21st International Middleware Conference, 2020, pp. 1–13.
[13] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, R. Arpaci-Dusseau, SOCK: Rapid task provisioning with serverless-optimized containers, in: Proceedings of the USENIX Annual Technical Conference, 2018, pp. 57–70.
[14] M. Stein, The serverless scheduling problem and NOAH, 2018, arXiv preprint arXiv:1809.06100.
[15] A. Singhvi, A. Balasubramanian, K. Houck, M.D. Shaikh, S. Venkataraman, A. Akella, Atoll: A scalable low-latency serverless platform, in: Proceedings of the ACM Symposium on Cloud Computing, 2021, pp. 138–152.
[16] W. Ling, L. Ma, C. Tian, Z. Hu, Pigeon: A dynamic and efficient serverless and faas framework for private cloud, in: Proceedings of the International Conference on Computational Science and Computational Intelligence, CSCI, IEEE, 2019, pp. 1416–1421.
[17] P.-M. Lin, A. Glikson, Mitigating cold starts in serverless platforms: A pool-based approach, 2019, arXiv preprint arXiv:1903.12221.
[18] Z. Xu, H. Zhang, X. Geng, Q. Wu, H. Ma, Adaptive function launching acceleration in serverless computing platforms, in: Proceedings of the 25th International Conference on Parallel and Distributed Systems, ICPADS, IEEE, 2019, pp. 9–16.
[19] D. Bermbach, A.-S. Karakaya, S. Buchholz, Using application knowledge to reduce cold starts in faas services, in: Proceedings of the 35th Annual ACM Symposium on Applied Computing, 2020, pp. 134–143.
[20] K. Solaiman, M.A. Adnan, WLEC: A not so cold architecture to mitigate cold start problem in serverless computing, in: Proceedings of the IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2020, pp. 144–153.
[21] N. Daw, U. Bellur, P. Kulkarni, Xanadu: Mitigating cascading cold starts in serverless function chain deployments, in: Proceedings of the 21st International Middleware Conference, 2020, pp. 356–370.

[22] G. Somma, C. Ayimba, P. Casari, S.P. Romano, V. Mancuso, When less is more: Core-restricted container provisioning for serverless computing, in: Proceedings of the IEEE INFOCOM - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), IEEE, 2020, pp. 1153–1159.
[23] S. Agarwal, M.A. Rodriguez, R. Buyya, A reinforcement learning approach to reduce serverless function cold start frequency, in: Proceedings of the 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), IEEE, 2021, pp. 797–803.
[24] K. Suo, J. Son, D. Cheng, W. Chen, S. Baidya, Tackling cold start of serverless applications by efficient and adaptive container runtime reusing, in: Proceedings of the 2021 IEEE International Conference on Cluster Computing, CLUSTER, IEEE, 2021, pp. 433–443.
[25] H. Yu, H. Wang, J. Li, S.-J. Park, Harvesting idle resources in serverless computing via reinforcement learning, 2021, arXiv preprint arXiv:2108.12717.
[26] L. Schuler, S. Jamil, N. Kühl, AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments, in: Proceedings of the IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), IEEE, 2021, pp. 804–811.
[27] X. Li, P. Kang, J. Molone, W. Wang, P. Lama, KneeScale: Efficient resource scaling for serverless computing at the edge, in: Proceedings of the 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), IEEE, 2022, pp. 180–189.
[28] H.-D. Phung, Y. Kim, A prediction based autoscaling in serverless computing, in: Proceedings of the 2022 13th International Conference on Information and Communication Technology Convergence, ICTC, IEEE, 2022, pp. 763–766.
[29] A. Zafeiropoulos, E. Fotopoulou, N. Filinis, S. Papavassiliou, Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms, Simul. Model. Pr. Theory 116 (2022) 102461.
[30] P. Benedetti, M. Femminella, G. Reali, K. Steenhaut, Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications, in: Proceedings of the 2022 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events (PerCom Workshops), IEEE, 2022, pp. 674–679.
[31] P. Vahidinia, B. Farahani, F.S. Aliee, Mitigating cold start problem in serverless computing: A reinforcement learning approach, IEEE Internet Things J. (2022).
[32] H. Qiu, W. Mao, A. Patke, C. Wang, H. Franke, Z.T. Kalbarczyk, T. Başar, R.K. Iyer, Reinforcement learning for resource management in multi-tenant serverless platforms, in: Proceedings of the 2nd European Workshop on Machine Learning and Systems, 2022, pp. 20–28.
[33] Z. Zhang, T. Wang, A. Li, W. Zhang, Adaptive auto-scaling of delay-sensitive serverless services with reinforcement learning, in: Proceedings of the 2022 IEEE 46th Annual Computers, Software, and Applications Conference, COMPSAC, IEEE, 2022, pp. 866–871.
[34] H. Yu, A.A. Irissappane, H. Wang, W.J. Lloyd, FaaSRank: Learning to schedule functions in serverless platforms, in: Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS, IEEE, 2021, pp. 31–40.
[35] C.K. Dehury, S. Poojara, S.N. Srirama, DeF-DReL: Systematic deployment of serverless functions in fog and cloud environments using deep reinforcement learning, 2021, arXiv preprint arXiv:2110.15702.
[36] Q. Tang, R. Xie, F.R. Yu, T. Chen, R. Zhang, T. Huang, Y. Liu, Distributed task scheduling in serverless edge computing networks for the internet of things: A learning approach, IEEE Internet Things J. 9 (20) (2022) 19634–19648.
[37] X. Yao, N. Chen, X. Yuan, P. Ou, Performance optimization of serverless edge computing function offloading based on deep reinforcement learning, Future Gener. Comput. Syst. 139 (2023) 74–86.
[38] H. Jeon, S. Shin, C. Cho, S. Yoon, Deep reinforcement learning for qos-aware package caching in serverless edge computing, in: Proceedings of the 2021 IEEE Global Communications Conference, GLOBECOM, IEEE, 2021, pp. 1–6.
[39] A. Mampage, S. Karunasekera, R. Buyya, Deep reinforcement learning for application scheduling in resource-constrained, multi-tenant serverless computing environments, Future Gener. Comput. Syst. 143 (2023) 277–292.
[40] A. Suresh, G. Somashekar, A. Varadarajan, V.R. Kakarla, H. Upadhyay, A. Gandhi, ENSURE: Efficient scheduling and autonomous resource management in serverless environments, in: Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS, IEEE, 2020, pp. 1–10.
[41] V. Mnih, A.P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning, in: Proceedings of the International Conference on Machine Learning, PMLR, 2016, pp. 1928–1937.
[42] Y. Tang, S. Agrawal, Discretizing continuous action space for on-policy optimization, in: Proceedings of the Aaai Conference on Artificial Intelligence, 34, (04) 2020, pp. 5981–5988.
[43] Melbourne research cloud documentation, 2022, (Accessed on 22 Jul 2022) https://docs.cloud.unimelb.edu.au/.
[44] Prometheus - monitoring system & time series database, 2022, (Accessed on 08 Sep 2022) https://prometheus.io/.
[45] AWS pricing calculator, 2022, (Accessed on 25 Jul 2022) https://calculator.aws/#/addService/EC2.
[46] J. Scheuner, S. Eismann, S. Talluri, E. Van Eyk, C. Abad, P. Leitner, A. Iosup, Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications, 2022, arXiv preprint arXiv:2205.07696.
[47] J. Kim, K. Lee, Functionbench: A suite of workloads for serverless cloud function service, in: Proceedings of the IEEE 12th International Conference on Cloud Computing, CLOUD, IEEE, 2019, pp. 502–504.
[48] Apache Software Foundation, Apache jmeter - apache jmeter™, 2021, (Accessed on 08 Nov 2021) https://jmeter.apache.org/.
[49] M. Shahrad, R. Fonseca, Í.n. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, R. Bianchini, Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider, in: Proceedings of the USENIX Annual Technical Conference, ATC, 2020, pp. 205–218.
[50] Knative, About autoscaling - knative, 2023, (Accessed on 18 April 2023) https://knative.dev/docs/serving/autoscaling/.
[51] Autoscaling - OpenFaaS, 2022, (Accessed on 19 April 2023) https://docs.openfaas.com/architecture/autoscaling/.

Anupama Mampage is a Ph.D. student at the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne, Australia. She received her B.Sc. Engineering (Hons) degree, specialized in Electronic and Telecommunication Engineering from the University of Moratuwa, Sri Lanka, in 2017. Her research interests include Serverless Computing, Internet of Things (IoT), Distributed Systems and Reinforcement Learning.

Shanika Karunasekera is currently a Professor in the School of Computing and Information Systems and the Deputy Dean (Academic) in the Faculty of Engineering and IT, University of Melbourne, Australia. She received her B.Sc. degree in Electronic and Telecommunications Engineering from the University of Moratuwa, Sri Lanka, in 1990 and the Ph.D. degree in electrical engineering from the University of Cambridge, U.K., in 1995. Her research interests include Distributed Computing, Mobile Computing, and Social Media Analytics.

Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He has authored over 825 publications and seven text books including "Mastering Cloud Computing" published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese and international markets respectively. He is one of the highly cited authors in computer science and software engineering worldwide (h-index=167, g-index=360, 146900+ citations).