Future Generation Computer Systems 143 (2023) 277–292

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Deep reinforcement learning for application scheduling in resource-constrained, multi-tenant serverless computing environments

Anupama Mampage [*], Shanika Karunasekera, Rajkumar Buyya

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

A R T I C L E I N F O

Article history:
Received 25 September 2022
Received in revised form 26 December 2022
Accepted 6 February 2023
Available online 9 February 2023

Keywords:
Serverless computing
Resource cost efficiency
Function scheduling
Resource contention
Reinforcement learning
Practical experiments

A B S T R A C T

Serverless computing has sparked a massive interest in both the cloud service providers and their clientele in recent years. This model entails the shift of the entire matter of resource management of user applications to the service provider. In serverless systems, the provider is highly motivated to attain cost efficient usage of their infrastructure, given the granular billing modules involved. However, due to the dynamic and multi-tenant nature of the serverless workloads and systems, achieving efficient resource management while maintaining function performance is a challenging task. Rapid changes in demand levels for applications cause variations in actual resource usage patterns of function instances. This leads to performance variations in co-located functions which compete for similar resources, due to resource contentions. Most existing serverless scheduling works offer heuristic techniques for function scheduling, which are unable to capture the true dynamism in these systems caused by multi-tenancy and varying user request patterns. Further, they rarely consider the often contradicting dual objectives of achieving provider resource efficiency along with application performance. In this article, we propose a novel technique incorporating Deep Reinforcement Learning (DRL) to overcome the aforementioned challenges for function scheduling in a highly dynamic serverless computing environment with heterogeneous computing resources. We train and evaluate our model in a practical setting incorporating Kubeless, an open-source serverless framework, deployed on a 23-node Kubernetes cluster setup. Extensive experiments done on this testbed environment show promising results with improvements of up to 24% and 34% in terms of application response time and resource usage cost respectively, compared to baseline techniques.

1. Introduction

The paradigm shift in cloud computing caused by the serverless computing concept implies that the provider handles all the operational tasks related to application resource management. This include instance selection, resource allocation and scaling of cloud resources for multiple applications belonging to multiple end users. Further, in contrast to managing resources for long-running applications in traditional cloud computing environments, the ephemeral nature of serverless functions poses a unique set of challenges to the providers. Function instances need to be created and scaled up and down in an adhoc manner based on request arrivals, where a majority of function requests would only last a maximum execution duration of one second, which creates complex system dynamics.

Serverless systems are also multi-tenant environments where multiple applications of different users could reside on the same server or more specifically on the same Virtual Machine (VM). Thus, resource contention among these applications, when competing for the same set of resources, is quite a prevalent issue. Moreover, the majority of open-source serverless platforms [1–3], consist of a system architecture where a single function instance serves multiple concurrent requests, which is the serverless environment in consideration for our work. Under such a system model, the situation is further aggravated when rapid changes in request rates cause the resource consumption of individual function instances to fluctuate over time. Regardless of these factors, each end user expects the cloud provider to guarantee satisfactory performance for their applications.

Early research works in this area, highlight performance limitations caused by contention among co-resident function instances on the same VM on commercial serverless platforms [4].

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

https://doi.org/10.1016/j.future.2023.02.006
0167-739X/© 2023 Elsevier B.V. All rights reserved.

In subsequent research works, a few have studied resource congestion which is a major barrier on achieving the desired performance objectives in serverless systems [5,6]. On the other hand, an often disregarded factor when focusing on application performance on serverless systems is the cost efficiency of the underlying resources on the provider side. The serverless billing model charges the user only for the resource-time consumed during function execution with a millisecond level granularity. Regardless of that, cloud vendors maintain their infrastructure throughout, even with partial utilization. Thus, this billing model necessitates the cloud vendors to focus heavily on the optimum utilization of their resources [7,8].

Existing commercial serverless platforms mostly follow simple heuristics in function scheduling. AWS Lambda treats the function placement decision as a bin packing problem which maximizes VM memory utilization [4]. Azure Functions follow a spread placement policy to avoid co-location of concurrent instances of the same function on the same VM [4]. A hash-based first-fit heuristic is employed by IBM OpenWhisk, aimed at a better cache hit rate and instance re-use [9]. In research literature many have attempted at presenting techniques for function scheduling. Most of the existing works focus primarily on satisfying application latency requirements of users and managing the resource cost for the end user [10,11], but not on optimizing cloud provider infrastructure costs. Further, their efforts are mostly directed towards articulating heuristic solutions for the simpler problem of individual request scheduling, based on a system model which serves only a single concurrent request per function instance. Many works also fail to attain overall workload and system awareness, which is detrimental to the effectiveness of the provided solutions in a highly dynamic serverless system. In contrast, our work addresses the complex problem of scheduling function instances which serve multiple concurrent requests, in a cost efficient manner with complete awareness of workload patterns and system dynamics, so that application performance is not hindered by resource contention.

Reinforcement Learning (RL) techniques are increasingly being used for solving problems related to serverless resource management as seen from a few contemporary research works [12,13]. The approach of learning through experience suits well, the unpredictable nature of serverless workloads and systems where an individual function request would have a millisecond level duration [14] and the co-residency of different applications on a VM would change swiftly over time with changing request arrival rates for deployed functions. Further, although RL techniques have been extensively explored for general cloud scheduling problems in literature, almost all these works incorporate simulator environments for training and testing their models, which have only a limited capability in capturing the actual resource congestion situation in a practical setting. Thus in this work we design an actual test-bed to train and evaluate our DRL models, which capture the fine details of application resource characteristics, workload patterns and the environment. Our model evaluations show promising results which outperform other baseline techniques. The key **contributions** of our work are as follows:

1. We formulate and present a RL oriented model of the problem of function instance scheduling in a resource constrained, multi-tenant serverless computing environment.
2. We propose a multi-step Deep Q Learning (DQN) model for developing a workload and system aware scheduling framework for serverless functions, aimed at optimizing application response time latency and provider cost efficiency. Since these two are conflicting goals, we add flexibility to the model to establish a trade-off between these goals as desired by the users.
3. We design a practical training environment for the DRL agent, integrated with the open-source serverless platform Kubeless [15], which is deployed on a Kubernetes [16] cluster composed of heterogeneous VMs.
4. We conduct extensive experiments using real world single and multi-function serverless applications [17,18] and function traces captured from Microsoft Azure Functions [19], to evaluate the performance and scalability of the proposed DRL model and compare it with baseline schedulers.

The rest of the paper is organized as follows: Section 2 reviews existing related works. Section 3 presents the system model and the mathematical formulation of the problem. Section 4 introduces the DRL oriented framework for function scheduling, followed by the design details of the agent training environment in Section 5. Sections 6 and 7 discuss the performance evaluation of the proposed technique and the potential for future work, respectively.

2. Related work

We focus our discussion on related works under two key areas as, serverless function scheduling and the application of RL techniques for resource management in serverless computing environments.

2.1. Serverless function scheduling

The problem space of serverless function scheduling has emerged as a new research area in recent times. Various solutions are presented in existing literature for the problem of finding a suitable host node for scheduling a function instance, which may accommodate either a single request or multiple concurrent requests, based on the system architecture.

A package-aware scheduler for serverless functions is proposed in [10]. They try to bundle function requests requiring similar packages to the same node, with a focus on reducing function cold start latency. Other than the package dependencies, they do not consider any other workload characteristics in the scheduling decision. [26] presents a locality-aware scheduler to reduce function latencies. A preliminary design for a centralized scheduler is presented in [20], which assigns each function execution to an individual CPU core. They aim to reduce overloading of cores and co-located function interference. A similar scheduling policy coupled with request queuing is evaluated in [25]. [21] uses a first-fit heuristic for request load balancing in their serverless setup. A supervised Machine Learning (ML) based approach is presented in [22], for selecting a VM for scheduling single function applications. Their objectives are to reduce function execution time and user cost by improving function throughput. The presented approach requires the platform to possess a comprehensive prior understanding of the behavior of an application and thus will not have the flexibility to adapt to dynamic workloads. [23] also explores a greedy scheduling approach to improve cluster utilization. A heuristic based on function latency in each VM is used in [5] to schedule function requests. A request priority and a deadline based greedy heuristic is proposed in [6] to choose a VM. [14] studies an architecture with semi-global schedulers in a serverless system using a spread-placement approach for function instance placement. A cost, function load, and locality-aware heuristic solution for function scheduling is proposed in [27], where the solution lacks overall system awareness. Another heuristic solution which includes an excessively time consuming manual profiling of function co-location patterns based on their resource usages is discussed in [29]. An input sensitive container allocation and request scheduling policy is

Table 1
Summary of literature review.

Work	Application model		Scheduling technique	Decision parameters				Request concurrency		VM
	Single function	Function chain		Optimization objective		Workload awareness	Overall system awareness	Single request	Multiple requests	Heterogeneity
				Response time	Provider cost efficiency					
[10]	✓		Heuristic	✓				✓		
[20]	✓		Heuristic	✓				✓		
[21]	✓		Heuristic	✓					✓	
[22]	✓		ML	✓			✓		✓	
[11]		✓	Heuristic	✓			✓		✓	
[23]		✓	Heuristic		✓			✓		
[5]	✓		Heuristic	✓	✓			✓		
[6]	✓		Heuristic	✓	✓			✓		
[14]		✓	Heuristic	✓				✓		
[24]	✓		DRL	✓			✓	✓		
[25]	✓		Heuristic	✓				✓		
[26]	✓		Heuristic	✓		✓	✓	✓		
[27]	✓		Heuristic	✓	✓	✓		✓		
[28]		✓	Heuristic	✓			✓	✓		✓
[29]	✓		Heuristic	✓	✓	✓		✓		
Our proposed work	✓	✓	DRL	✓	✓	✓	✓		✓	✓

presented in [30], where their focus is mostly on request batching and reordering to minimize SLO (Service Level Objective) violations.

A hybrid scheduling framework is presented in [11], which uses a greedy algorithm to determine the order and placement of functions in either the private or the public cloud. [28] presents a heuristic approach for scheduling function workflows in a federated serverless environment. Their focus is limited to improving the makespan of function executions.

2.2. Application of RL for serverless resource management

A number of works have explored RL techniques for task scheduling in traditional cloud computing environments. [31–35]. All of these existing works present experimentation done solely based on simulator environments. As opposed to experiments designed on a practical setting, training a model on a simulator environment often times incorporates assumptions such as fixed execution times for tasks on a given machine irrespective of resource pressure, uniform resource consumption by applications throughout the experiment etc. These assumptions pose limitations in creating a realistic image of the actual behavior of a cloud environment, specially under resource constrained scenarios which is the focus of our work. Further, unlike the traditional long running monolithic application workloads in the cloud, serverless functions are designed to have very short run times which result in the level of resource contention among applications to change rapidly within seconds. Thus, solutions presented for generic cloud applications have little or no usability in serverless computing environments in achieving satisfactory results [23]. For these reasons, here we extensively focus on reviewing existing works utilizing RL solutions in the context of serverless computing environments. A few recent research works have demonstrated the applicability of RL techniques for serverless resource management as discussed below.

In [12] the authors present a Q-learning based RL approach to determine the best level of function request concurrency per container in order to achieve better performance in terms of system throughput and mean function latency. A Proximal Policy Optimization (PPO) algorithm is leveraged in [13] to dynamically manage resource configurations of each function container. CPU and memory resources from idle functions are harvested and allocated to under-provisioned functions, after assessing the cluster state with each function request arrival. A Q-learning based approach is used in [36] to minimize serverless function cold start frequency. A multi-agent Proximal Policy Optimization (PPO) approach is studied in [37] for horizontal and vertical scaling of serverless functions. In [24], a policy gradient algorithm is used to calculate a score function for each server, in order to determine a suitable node for scheduling an individual function request. They focus only on reducing the completion time for each function and also does not pay attention to workload dynamics. Except for this work, all other existing works exploring RL techniques for serverless resource management focus on resource scaling and not function scheduling.

Table 1 summarizes the reviewed works related specifically to serverless function scheduling, in terms of the application model, technique used, optimization objective, workload-awareness (awareness on request arrival patterns), overall system awareness (complete awareness on the cluster VM resource usage metrics related to CPU, memory, network and disk I/O), request concurrency (ability to serve multiple concurrent requests by a function instance) and VM heterogeneity. Most of the existing works focus only on a specific aspect of the application or the system, in the scheduling decision making. Our work in contrast is focused on gaining a comprehensive understanding on the status of the system and the dynamic function workload parameters at any given time. This knowledge is then used in determining the VM node most capable of hosting a function instance. We also strive to achieve a balance between the two conflicting objectives of application performance in terms of function response time, and provider side cost efficiency, which was generally seen to be ignored in prior works.

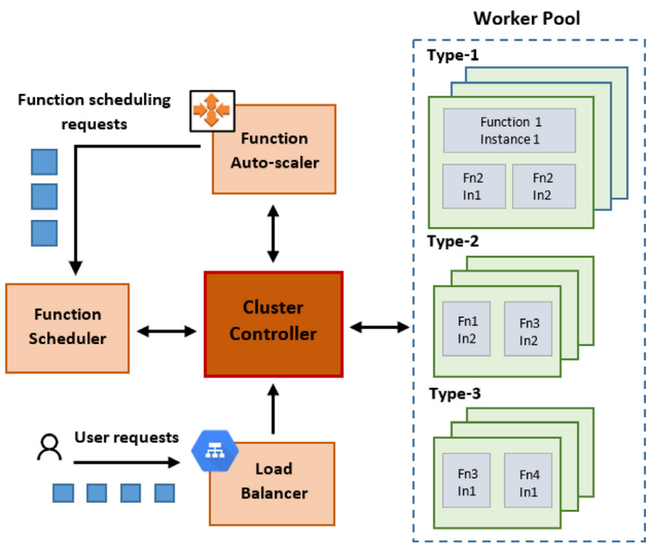


Fig. 1. The system model of the serverless application scheduling environment.

3. Time and cost optimized function scheduling

This section discusses the system model and formulate the problem of application scheduling in a serverless computing environment with a flexible trade-off between response time and provider cost optimization.

3.1. System model

We formulate our system model around the system architecture of majority of the existing open-source serverless frameworks serving many enterprise users [1–3]. In this work, we consider a serverless application to be composed of either a single or multiple functions. Multi function applications are composed of chained functions whose execution sequences are determined by user input.

Fig. 1 illustrates the high level system model used in this work. We consider a cluster made up of heterogeneous VMs with varying compute and memory capacities as the set of worker nodes. An instance of a single function is the smallest resource unit of computing, that could be scheduled and managed, also referred to as a pod. A pod consists of a single container holding the function code and its dependencies. We consider at least a single instance of a function to be always present in the system. A function instance can handle multiple concurrent requests received from end users.

Additional instances of the same function (replicas) are deployed to the cluster depending on the request demand. This process is called function auto-scaling and is handled by the function auto-scaler. Auto-scaling of a particular function is triggered whenever the average CPU utilization level across all of its instances goes above a particular set utilization threshold. The number of new replicas to be created is decided based on the current and the desired CPU utilization levels. Each new function replica needs to be scheduled on a suitable VM for meeting the request load for that function. This is handled by the function scheduler, which is the focus of our work.

The load balancer is responsible for distributing the incoming function requests among the existing function replicas. We consider that these requests are forwarded to the relevant deployed function instances in a round robin manner. An instance of a particular function could receive requests originating from multiple user applications, depending on the nature of function chaining. Arrival times of user requests for each application are stochastic and the cluster will have no prior knowledge of the workload arrival patterns. This means that the request arrival rate for each function can vary randomly within short periods of time. Depending on the nature of its operation, each function instance would compete for different levels of resources in terms of CPU, memory, network and disk I/O bandwidth. The actual resource consumption of a function instance at a given time is determined by the total arrival rate of dependent requests and also on the number of instances of the function present in the system at the time. Based on this actual level of resource consumption of each function instance running on a node at a time, the performance of user requests will vary depending on the extent of resource pressure. Thus the placement decision of a scaled function needs to incorporate the workload and the system resource usage dynamics, while also focusing on the resource cost efficiency of the worker nodes in the cluster. The cluster controller coordinates the actions of the function auto-scaler, scheduler, and the load balancer, whilst maintaining communication with the worker pool.

3.2. Problem formulation

Consider a set $V = \{v_1, v_2, \dots, v_N\}$, to be the set of available VMs in a serverless cluster environment, where N is the total number of VMs and v_i, $1 \le i \le N$ is the ith VM. Each VM is defined by a two-dimensional vector representing the resource capacities in terms of CPU and memory denoted as v_i^c and v_i^m respectively. Hence we have, $v_i =< v_i^c, v_i^m >$. The total CPU capacity in a VM is determined by the number of virtual cores (vCPUs) and the memory capacity is measured in Mega Bytes (MBs). The available free CPU and memory resources in VM, v_i at time t is denoted by, $v_i^C(t)$ and $v_i^M(t)$ respectively.

Consider $\varepsilon = \{1, 2, 3, \dots, Q\}$ to be the index set of all the different functions deployed in the cluster. Let $P^k = \{p_1^k, p_2^k, \dots, p_{M_k}^k\}$ be the sequence of pod (function instance) scheduling requests received at the scheduler for the kth function, where $1 \le k \le Q$ and M_k is the total number of scheduling requests. Also p_j^k, $1 \le j \le M_k$ is the jth request. Each pod request carries four attributes, i.e., $p_j^k =< p_j^{kc}, p_j^{km}, p_j^{kt}, r_0^k >$. p_j^{kc} and p_j^{km} denote the requested minimum CPU and memory resources for the pod. p_j^{kt} refers to the pod request arrival time and r_0^k is the standard response time for a request of the function k. Note that here p_j^{kc} and p_j^{km} are set as soft resource requests which denote the minimum guaranteed resources a pod of a particular function needs to be allocated with, in order to handle a defined number of function requests at a time. In line with the Docker CPU shares [38] policy for resource allocation to containers, these values determine the proportion of CPU and memory each pod would get when faced with resource contention in a node. But when at ease without resource pressure, a pod is free to use as much CPU and memory of the nodes, as it requires. The standard response time for a function request, r_0^k is the average request response time obtained by running a function pod in isolation on a dedicated VM.

When scheduling a function instance on a worker node, the following CPU and memory resource demand and capacity constraints have to be considered.

$$p_j^{kc} \le v_i^C(t) \tag{1}$$

$$p_j^{km} \le v_i^M(t) \tag{2}$$

i.e., the CPU and memory request of pod p_j^k should not exceed the available (unallocated) CPU and memory of the VM at time t.

Table 2
Definition of symbols.

Symbol	Definition
v	A VM or compute node available for function execution
N	Total number of available VMs
δ	Index set of all the available VMs, $\delta = \{1, 2, \ldots, N\}$
v_i^c	Total CPU capacity of a VM, $i \in \delta$
v_i^m	Total memory capacity of a VM, $i \in \delta$
v_i^C	Available CPU in a VM, $i \in \delta$
v_i^M	Available memory in a VM, $i \in \delta$
ε	Index set of different functions, $\varepsilon = \{1, 2, \ldots, Q\}$
M_k	Total number of instance scheduling requests for a function, $k \in \varepsilon$
p_j^k	jth instance scheduling request of function p^k, $k \in \varepsilon$
p_j^{kc}	Requested minimum CPU for the function instance, p_j^k
p_j^{km}	Requested minimum memory for the function instance, p_j^k
p_j^{kt}	Arrival time of the function instance, p_j^k for scheduling
p_{Con}^k	Request concurrency on a pod belonging to function p^k, $k \in \varepsilon$
k_r	Request arrival rate for a function, $k \in \varepsilon$
k_n	Deployed number of replicas for a function, $k \in \varepsilon$
r_0^k	Standard response time for a function request, $k \in \varepsilon$
γ	Index set of different applications, $\gamma = \{1, 2, \ldots, A\}$
L^b	Number of user requests received by an application, $b \in \gamma$
R_q^b	Total response time of the constituent functions of the qth request of an application, $b \in \gamma$
R_{q0}^b	Total standard response time of the constituent functions of the qth request of an application, $b \in \gamma$
$price_i$	Unit price of VM, v_i
t_i	Total active time of VM, v_i

We identify a VM's available CPU and memory resource levels as follows:

$$v_i^C(t) = v_i^c - \sum_{k=1}^{Q}\sum_{j=1}^{M_k} u_{kji}(t)p_j^{kc}(t) \tag{3}$$

$$v_i^M(t) = v_i^m - \sum_{k=1}^{Q}\sum_{j=1}^{M_k} u_{kji}(t)p_j^{km}(t) \tag{4}$$

where we define a binary variable u_{kji} to indicate whether pod p_j^k is currently placed in v_i or not, i.e., $\forall i \in \delta$, we have;

$$u_{kji}(t) = \begin{cases} 1, & \text{if pod } p_j^k \text{ is deployed on } v_i \text{ at time } t \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

Even though p_j^{kc} and p_j^{km} represent the resource constraints to be met when assigning a pod to a host node, the actual resource consumption of a single function instance at time t will depend on the number of concurrent requests that it accommodates at the time. Request concurrency on a pod belonging to the kth function, $p_{Con}^k(t)$ is determined by the request arrival rate $k_r(t)$ and the deployed number of replicas $k_n(t)$ at time t. This value of request concurrency is an important parameter for modeling the level of resource contention on host nodes. Due to the round robin nature of request distribution among replicas, we derive $p_{Con}^k(t)$ as follows:

$$p_{Con}^k(t) = \frac{k_r(t)}{k_n(t)} \tag{6}$$

Note that $k_r(t)$ above is a resultant of the cumulative arrival rate of requests of all user applications consuming the kth function. The time t in the above expressions: (1), (2), (3), (4), (5) and (6) refers to the time that a pod is taken in for scheduling.

A primary objective of this work is to minimize the performance degradation of the execution of user requests, caused by resource contention in multi-tenant host nodes. We consider the overall application response time latency to be the metric most reflective of the performance of an application workload. Consider $\gamma = \{1, 2, 3, \ldots, A\}$ to be the index set of all the different applications receiving user requests and L^b, $1 \leq b \leq A$ be the number of requests received by bth application. R_q^b is the total response time of the constituent functions of the qth request of application b, $1 \leq q \leq L^b$. R_{q0}^b is the total standard response time of the constituent functions of the same. Thus we define the ratio of these two values averaged over the total requests for a particular application as the relative application response time (RART). For a given workload, our target is to minimize the sum of the RART across all the user applications over the duration of the workload. Considering RART instead of the response time itself, removes any bias in our target objective due to varying execution times of serverless functions when working in a multi-tenant environment. Accordingly we formulate the application performance optimization objective as follows:

$$Minimize : Sum\ RART = \sum_{b=1}^{A} \frac{1}{L^b} \sum_{q=1}^{L^b} \frac{R_q^b}{R_{q0}^b} \tag{7}$$

When calculating R_q^b and R_{q0}^b we consider only the sum of response times of the individual functions involved with the particular execution sequence of the application. This is possible since chained functions simply act as triggers for the next function in sequence, and hence this process does not involve any communication delay. Note that we denote the total standard response time for an application's request (R_{q0}^b) as a function of q, since the relevant constituent functions will depend on the request input.

Further, we aim to minimize the provider expenses incurred for the execution of serverless workloads. Since our deployed cluster is formed of heterogeneous VMs with varying CPU and memory capacities, the cost incurred depends on the VM instance pricing. Thus, the provider cost optimization objective could be expressed as follows:

$$Minimize : Cost_{Total} = \sum_{i=1}^{N} price_i \times t_i \tag{8}$$

where $price_i$ is the unit price of VM v_i and t_i is the total active time of the ith VM over the duration of workload executions. A VM is considered to be in active mode when it is serving requests of at least a single function. Thus our target is to release cluster infrastructure after experiencing high utilization levels during their active life time. In the rest of the paper at times, we use the term resource efficiency to refer to the cost optimization objective.

Overall, the focus of this study is to minimize both the performance degradation of functions and to enable efficient utilization of VMs. These two are generally known to be conflicting objectives. Therefore, we introduce a system parameter $\beta \in [0, 1]$, which is adjustable by users to prioritize each optimization objective as required. Accordingly, we present our target objective as follows:

$$Minimize : \beta \times Sum\ RART + (1 - \beta) \times Cost_{Total} \tag{9}$$

Table 2 summarizes the important notations and descriptions presented in this section.

4. Deep reinforcement learning model

In this section we first introduce the RL paradigm and discuss the application of RL in the context of the serverless function scheduling problem discussed above. Then we elaborate on the specific RL techniques we have incorporated in this work.

4.1. Application of RL for function scheduling

RL is a form of machine learning, which is quite distinct from the traditional machine learning techniques identified as supervised and unsupervised learning. The primary goal of supervised and unsupervised learning is to find and comprehend patterns or a hidden structure in collections of labeled or unlabeled training data. In contrast, a RL agent learns to map actions in the action space, to different states from the environment, in the best way possible in order to maximize a reward signal over time. During the learning process, the agent interacts with the environment and at each time step, takes an action based on the current policy $\pi(a_t|s_t)$ and in turn receives a reward r_{t+1}. s_t is the current state of the environment and a_t is the action taken.

In this paper, we apply the concepts of RL to solve the problem of scheduling function instances in a serverless computing environment with dynamic incoming workloads. In the context of our problem, the function scheduler acts as the RL agent and each time-step of our agent training model corresponds to scheduling a function instance from the function scheduling request queue. The cluster environment composed of the worker nodes form the environment with which the agent interacts. The state is a combination of all the resource usage statistics of each worker node in the cluster and the workload nature of the function instance to be scheduled. The set of VMs form the action space from which the agent chooses a suitable action. The reward that the agent receives for each action is based on the scheduling objectives of application performance and provider cost optimization. The task assigned to the scheduling agent is to choose the best VM to schedule a function instance while satisfying the basic resource demand and capacity constraints of the system. Below we define the key components of our RL model.

State Space: The state metrics that are considered in the formation of the state space s_t at time t with function instance p_j^k waiting to be scheduled, are as follows:

1. The actual CPU, memory, network (sum of network bytes received and transmitted) and disk I/O (sum of disk read and write bytes) bandwidth utilization of each of the nodes in the cluster at time t
2. The CPU and memory capacities of each of the nodes in the cluster
3. Unit price of each cluster node
4. The total of minimum CPU and memory requested by function instances running in each node at time t
5. The active status of each node. A node is considered to be active at time t if it contains instances of functions for which user requests are received at the cluster at the time
6. The number of replicas of function of type p^k already deployed on each node at time t
7. The minimum CPU and memory requested by the function instance, p_j^k
8. Sum of network bytes received and transmitted during a single request execution of kth function on average
9. Sum of disk read and write bytes during a single request execution of kth function on average
10. Request concurrency on each function instance of type p^k calculated using Eq. (6)
11. Relative function response time (RFRT) of function of type p^k in the cluster at time t. This is the ratio between the actual and standard response time (r_0^k) for the function
12. The request arrival rates for each different function deployed in the cluster at time t

Action space: The action space represents the index set of the VMs available for scheduling the function instance.

Reward: As per the optimization objectives discussed in Section 3, we define the reward r_{t+1} for each action a_t as follows:

1. R_1: The sum of RFRT calculated across all the deployed functions in the cluster, just before implementation of the next action, a_{t+1}. This is a good measure of our performance optimization objective of RART in Eq. (7), since application response time is directly dependent on that of its constituent functions. Also, it presents a reward more identifiable with each function scheduling action of the DRL agent.
2. R_2: The difference in the cumulative cost of cluster VM usage just before the implementation of the action, a_t and just before the implementation of the next action, a_{t+1}, calculated as in Eq. (8).

For training purposes we take normalized values of both these rewards at each time step so that the scale of each parameter does not bias the training process. The minimum and maximum values for normalizing are arrived at by observing samples across time steps in multiple episodes. Accordingly, the reward awarded to the agent after each scheduling decision is:

$$Reward = -(\beta \times (\frac{R_1 - R_{1min}}{R_{1max} - R_{1min}}) + (1-\beta) \times (\frac{R_2 - R_{2min}}{R_{2max} - R_{min}})) \quad (10)$$

The negative sign is required to encourage minimization of both the function response time and VM usage cost.

4.2. Proposed DRL technique for function scheduling

We adapt a variation of the DRL based algorithm, DQN to solve the problem of scheduling function instances in the proposed RL environment.

Background: The objective of a reinforcement learning agent is to find the optimal policy, which is the policy that maximizes the expected cumulative reward over time.

$$\mathbf{E}[\sum_{t=0}^{\infty} \gamma^t r_t] = \mathbf{E}[r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \cdots] \quad (11)$$

Here, γ is the discounting factor, which determines the significance of future rewards. r is the reward received at each step by following the policy $\pi(a_t|s_t)$.

Q-Learning: Q-Learning is a temporary difference algorithm in RL, and it works by assessing the 'Quality', or how good a particular action is, with regard to gaining future rewards. This is represented by means of the Q-function for a state–action pair, $Q(s,a)$. The optimal Q-function, $Q^*(s,a)$ denotes the maximum reward that can be obtained by following the optimal policy at each step. The Bellman optimality equation for the optimal Q-function is defined as follows:

$$Q^*(s,a) = \mathbf{E}[r + \gamma \max_{a'} Q^*(s',a')] \quad (12)$$

Deep Q Learning (DQN): Due to the high-dimensional nature of the environment modeled in our work, it is infeasible to incorporate tabular Q-learning solutions. This is owing to the computational and space restrictions associated with maintaining the data and also the difficulty in exploring all the state–action pairs by the agent during the training process. We can overcome these shortcomings by training a neural network and using it as a function approximator to determine the Q values. Accordingly, we parameterize our Q function by an adjustable parameter θ, i.e., $Q(s,a;\theta) \approx Q^*(s,a)$. We then feed the environmental state to the neural network, which in turn returns the Q value of all the possible actions for that state. Subsequently, the action with the maximum Q-value is selected by the agent.

Thus in DQN, the objective is to predict the Q value, which is basically a regression task. Mean Squared Error (MSE) is generally used as the loss function for performing regression.

$$L(\theta) = \frac{1}{K} \sum_{i=1}^{K} (y_i - \hat{y}_i)^2 \quad (13)$$

Algorithm 1 DQN Based Function Scheduling Algorithm

```
1:  Initialize the main network parameter $\theta$ with random weights
2:  Initialize the target network parameter $\theta'$ by copying the weights from the main network
3:  Initialize the N-step buffer $D'$ and replay buffer $D$
4:  Initialize the training parameters $\epsilon$, $\alpha$, $\gamma$
5:  for episode = 1 to E do
6:      Reset the environment
7:      for step = 1 to T do
8:          Observe the state $s$
9:          Select an action $a$ using the $\epsilon$-greedy policy
10:         Execute the action $a$, move to the next state $s'$ and observe the reward $r$
11:         Store the transition $(s, a, r, s')$ in the buffer $D'$
12:         if size of $D' = N$ then
13:             Translate and move the N-step transition data to $D$
14:         Randomly sample a mini-batch of $K$ transitions from $D$
15:         Compute the loss $L(\theta)$
16:         Update the main network:
17:                 $\theta = \theta - \alpha \nabla_\theta L(\theta)$
18:         Update the target network every $P$ steps
    return
```

where y is the target value, $\hat{y}$ is the predicted value and K is the number of training samples involved. The target value is the optimum Q value. Accordingly,

$$y = r + \gamma \max_{a'} Q^*(s', a')$$

$$\hat{y} = Q_\theta(s', a')$$

As per the dynamic nature of our function scheduling environment and unprecedented delays in action executions at each time step in a practical training environment, we observed that the straightforward application of vanilla DQN algorithm did not work well for our problem. Hence we used its variant, multi-step DQN to train the agent. This involves a multi-step buffer which considers longer trajectories in storing the state transitions in memory, resulting in a more effective and efficient learning process for the agent. In contrast to a single step buffer, a multi-step buffer is known to give the agent a better view of the future rewards and also helps to propagate newly observed rewards to earlier visited states faster [39]. This technique is summarized in Algorithm 1.

The scheduling environment is reset at the beginning of each episode (line 6). Each time step corresponds to scheduling a function instance from the pod queue. At the start of each step, the environmental state is retrieved and the agent selects an action (line 9). After performing the selected action and receiving the reward, we store the agent's experience in memory. Due to the multi-step nature, every transition is first stored in a temporary buffer D' and then the most recent N transitions are summarized and moved to the replay buffer D (lines 11–13). Once the experiences are stored, we randomly sample a mini-batch of transitions from the buffer and train the network. The neural network training is done by finding the optimal network parameter θ which minimizes the loss function. Accordingly, we compute the gradient of our loss function $\nabla_\theta L(\theta)$ and update our network parameter θ (lines 14–18).

If we use the same neural network to calculate the target Q value of the next state–action pair, and also the predicted Q values, this causes instability in the loss function and the network learns poorly. To avoid this issue, we use a separate neural network for calculating the target values, keep its network parameter static for a while and periodically update its value referring to the main network.

Algorithm 2 Online Scheduling

```
1:  upon event Submission of a new poddo
2:       Enqueue pod in pod-waiting queue
3:  while P dood-waiting queue is not empty
4:      Dequeue a pod from queue
5:      Retrieve current cluster state info
6:      Retrieve function resource requirements and behavioral status
7:      Compose the state space
8:      Action $a$ = Agent(state)
9:      Create pod in the selected worker node
    return
```

Algorithm 2 presents the specific steps of the agent's behavior during online scheduling of function instances in the context of our modeled environment.

5. DRL agent training environment design and implementation

We investigate the problem space of time and cost optimized scheduling of serverless functions by designing and implementing an experimental framework using the serverless framework Kubeless, deployed on a Kubernetes cluster. From among the many existing open-source frameworks, we chose Kubeless for our work since it works with minimal changes to the underlying Kubernetes core components, and thus makes our entire setup compatible for easy resuse with any other framework utilizing Kubernetes for container orchestration, such as OpenFaas [1], Knative [2], Fission [3]. In this section we discuss the fundamental architectural setup of the designed system.

5.1. System architecture

Fig. 2 presents the overall architecture of our system. We have deployed this framework using 23 VM nodes on the Melbourne Research Cloud [40] which is part of the ARDC Nectar Research Cloud, the national research cloud of Australia [41]. Kubernetes is initially deployed on the cluster nodes, on top of which we deploy the serverless framework, Kubeless. As illustrated in the figure, our setup consists primarily of a control cluster, a worker cluster and the DRL agent which communicates with the control cluster.

The control cluster is made up of two nodes, each with 4 vCPUs and 16 GB of memory. One control cluster node contains all the core components of the Kubernetes control plane, which are responsible for the creation, management and auto-scaling of pods in a basic Kubernetes cluster. The controller component of the Kubeless framework resides on the second control plane node. It communicates with the Kubernetes controller manager in order to handle the function deployment, scheduling and auto-scaling processes. It also acts as the gateway for new application deployments and incoming function requests. Kubeless uses a Kubernetes Custom Resource Definition (CRD) to be able to create functions as custom Kubernetes resources. Given the application logic of a function via the CLI, the Kubeless controller coordinates with Kubernetes' components and automatically manages the deployment of an instance of the function in the cluster, as a pod.

We have also deployed Apache CouchDB [42] database as a cluster on the control plane nodes for persisting function data as required. CouchDB is an open source NoSQL database with fast querying and scaling capabilities which suit the requirements of a serverless environment. The database consumes the disk space

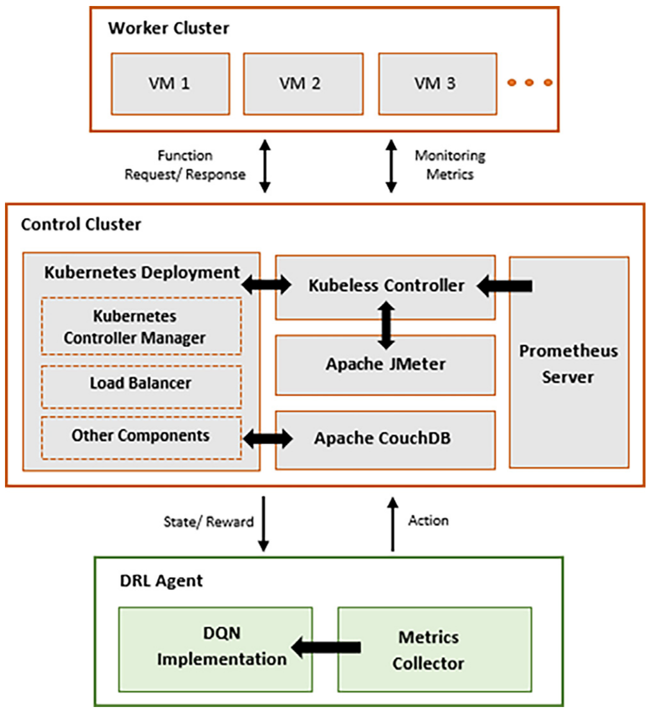


Fig. 2. The proposed system architecture of the practical testbed for training and evaluating the DRL agent.

(30 GB each) of the control plane nodes. The Prometheus metrics monitoring tool [43] is installed in our cluster setup, and the Prometheus server is deployed on the second node along with the Kubeless components. Prometheus periodically scrapes the configured set of metrics from the cluster and aggregates them on the Prometheus server. We have configured it to scrape system metrics associated with resource usage levels of each node and pod. We also gather metrics related to the incoming function request workloads such as the request arrival rates and request execution times, by observing the Kubeless controller gateway and the Kubernetes core components. We have also installed Apache JMeter [44], a load testing tool, in order to simulate a large number of user requests to functions as required. This tool too resides on the control cluster and is able to generate HTTP requests to multiple destinations simultaneously, at a given rate for a given time duration. At the start of each episode, a function request workload is created and sent to the worker cluster using this tool.

The worker cluster consists of 20 VM instances, each with varying number of vCPUs and memory capacities, described further under experimental settings. As per the nodes selected by the agent, function instances are deployed on worker nodes. Incoming requests for a function are forwarded to the relevant deployed function instances in a round robin manner as discussed under the system model. The response for each request is received at the control cluster. Each worker node also exposes scraped metrics to the Prometheus server.

The DRL agent which executes Algorithm 1 in our framework, is implemented in Python using Keras [45] and Tensorflow2 [46], on a VM with 8 vCPUs and 32 GB of memory. We have replaced the default Kubernetes pod scheduler with our custom scheduler which is incorporated into the agent's implementation. The custom scheduler uses a python client for the Kubernetes API, which watches for new pod requests. During each time step, the agent retrieves the state and reward metrics composed of the system and workload characteristics from the Prometheus server

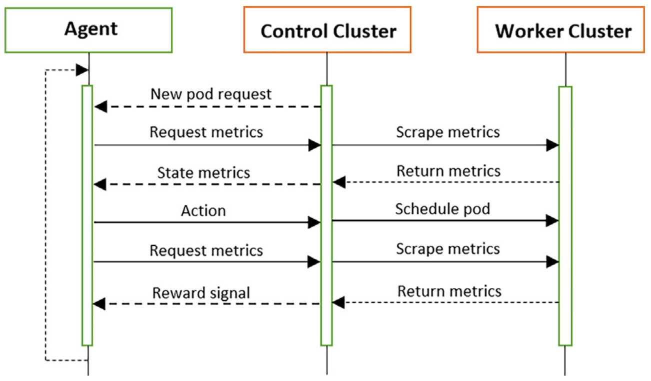


Fig. 3. The communication process flow of the DRL agent with the cluster during the training phase.

via HTTP APIs. The agent's selected action is communicated to the control cluster for implementation. The agent's process flow is explained in detail in the next section.

5.2. DRL agent's process flow

At the start of an episode, a concurrent request workload to multiple applications is created and sent to the worker cluster using the JMeter tool. These requests are served by the existing instances of the function in a round-robin manner. Once the auto-scaler triggers scaling up of function instances, the agent's process flow commences. Fig. 3 illustrates the sequence of actions that takes place at each subsequent time step. A new time step for the agent is triggered once a new pod scheduling request is seen by the Kubernetes watch API. The monitoring tool periodically scrapes and stores cluster metrics. At each new time step, the agent crawls the state metrics from the Prometheus server via HTTP APIs. Next a node is selected for scheduling the pod as per the agent's logic, and the control cluster is notified of the decision. Once the pod is scheduled on the selected node, we wait for a few seconds for the environment to react to the implementation done. Next the agent retrieves the reward metrics and moves on to the next pod scheduling request.

6. Performance evaluation

In this section we discuss the evaluation process of our proposed DRL framework for scheduling serverless function instances. We compare our solution with several state-of-the-art baseline algorithms under different scenarios.

6.1. Experimental settings

6.1.1. Cluster setup

We use the cluster setup described in Section 5 for both the training and evaluation experiments of our DRL model. As the set of worker nodes, we have used 20 VM instances with various pricing models, in line with the AWS EC2 instance pricing (in Australia) [47]. This enables us to recreate a real-life public cloud setting in order to train our agents to optimize provider cost. We conduct experiments under two cluster sizes of 10 VMs and 20 VMs in order to test model scalability. Table 3 summarizes the overall resource details of the worker cluster. The 10 VM cluster is composed of 2, 6 and 2 VMs with 2, 4 and 8 vCPUs respectively. We maintain the scrape interval of monitoring metrics at two seconds, in order to maintain the accuracy and relevance of the stored metrics.

Table 3
Worker cluster resource details.

Instance type	vCPU cores	Memory (GB)	Quantity	Price (AUD/h)
t4g.large	2	8	6	0.086
t4g.xlarge	4	16	10	0.172
t4g.2xlarge	8	32	4	0.344

6.1.2. Workload specifications

Serverless Applications: We refer to the ServiBench [18] and FunctionBench [17] benchmarking suites and choose 12 different single and multi-function real-world serverless applications and use them in all our experiments. The selected applications have a varying demand on CPU, memory, network and disk I/O bandwidth resources and thus different sensitivities to contention on node resources. After the deployment of an instance of each new function of an application in the cluster for the first time, we send multiple requests to the function in isolation on a VM to determine the average response time for a single request when not subject to resource pressure. This is used as the standard response time r_0, for the function in model training and for application performance evaluation. Further, we obtain approximate values for p_j^{kc}, p_j^{km} introduced in Section 3 and the network and disk I/O bandwidth consumed by a single request using this profiling step, to be used as reference values in deriving state parameters during agent training. Table 4 presents details on the nature of these applications.

Workload Creation: In addition to the inherent resource sensitivities of these applications, we also use function inputs to create additional variations of resource usage by them. Further, applications with chained functions would have varied execution paths based on the input values. We leverage the publicly available function traces from Microsoft Azure's serverless platform [19] to derive average function response times and request arrival rates when formulating the workloads for both training and evaluation of the model. Since Azure functions are already grouped in to applications, for multi-function applications we filter and use traces of matching applications.

The input parameters to individual functions are varied as required to attain the execution times extracted from these traces. For the 10 and 20 VM cluster scenarios, we maintain the request arrival rates at 5–20 and 5–60 requests per second and the maximum pod replicas for scaling at 4 and 6 respectively, for each function. Further, we configure the standard response time r_0 for a function request to be below one second, pod CPU requirements between 0.05–0.5 vCPUs and pod memory requirements within 50–500 MBs. These parameters were chosen so as to create enough request traffic in each of the VM cluster scenarios, while not overloading the system. We combine Azure Function traces spanning over two days and filter function traces that fall within these specified ranges of the function response time and request arrival rate parameters. We use the average function execution times from the traces as the function response times for our experiments. This could be done without causing any inaccuracy, since we have observed that the instance creation time for all our applications is quite similar and thus the evaluation of relative application performance is not affected by this delay. The request arrival rates are obtained by translating the per day total invocations for a function in the data set to a per second value. Accordingly, multiple variations of the selected benchmark applications are created by adjusting their function input values and request loads are created. A single episode consists of a set of different applications receiving simultaneous requests at a time for a particular duration, and each application would have requests arriving at different arrival rates.

The request load is generated in real time by the JMeter HTTP load generator. The R_{1min}, R_{1max}, R_{2min} and R_{2max} values for training the DRL models, are determined after running the created workloads multiple times and recording the calculated R_1 and R_2 values at each time step.

6.1.3. Hyper-parameter configurations

Table 5 highlights the hyper-parameters used in training the DRL agents. All the parameters for both the cluster scenarios were decided on a trial and error basis. The size of the N-step buffer was chosen so as to improve the agent's convergence speed without breaking the training progress. We use 600 function traces in total, derived from Azure Functions data set, in creating the workloads required for model training. The number of neurons in each hidden layer in the neural network for the 10 VM and 20 VM cluster scenarios are 100 and 200 respectively.

6.2. Performance metrics

We use the below metrics to evaluate the performance of our model.

Relative Application Response Time ratio: The sum of the relative response times of all the user applications during the span of the experiment, calculated using Eq. (7). At the end of a workload execution, we use the request response times recorded in the JMeter test report for each function, to arrive at this value.

Average Number of Nodes: The average number of VMs actively involved in request execution during a single episode. This is calculated by retrieving the number of active VMs in the cluster every two seconds and taking the average over the total duration of the workload.

VM Usage Cost: The total cost incurred for keeping the VMs active during a scheduling episode. This is calculated as shown in Eq. (8). We use the active nodes parameter together with the instance pricing given in Table 3 to arrive at this value.

Throughput: The average number of successfully served requests per second during an episode.

6.3. Baselines schedulers

We compare the performance of our DRL based scheduling framework with six baseline algorithms.

Round Robin (RR): Each incoming function instance is scheduled in a different VM with sufficient resources, in a cyclic manner.

Bin packing First-Fit (BPFF): This is a greedy scheduler similar to AWS Lambda's strategy of packing function invocations to improve VM resource utilization [4]. Nodes are numbered from 1–12 and pod requests are directed to the first VM which satisfies the minimum resource requirements.

Static Time Cost Aware (STCA): A scheduler which uses state parameters derived by the DRL agent in a static manner to select a VM. A separate rank and accordingly a score is given to each VM based on each parameter. Then the node with the highest or lowest overall score would be selected for function placement, based on the target objective. The state parameters taken into consideration are, CPU, memory, network and disk I/O utilization of each node, ratio of CPU and memory requests of running functions against their capacity in each node and the active status of the nodes. Based on the nature of these parameters choosing a VM with lower overall score (STCA-L) resembles better function performance while a higher score (STCA-H) promotes higher

Table 4
Serverless application details.

Name	Resource sensitivity				# of functions
	CPU	Memory	Disk I/O	Network	
Primary	High	High	–	–	1
Float	High	High	–	–	1
Matrix multiplication	High	High	–	–	1
Linpack	High	High	–	–	1
Load	Low	Low	–	High	1
Dd	High	Medium	High	–	1
Gzip-compression	High	Medium	High	–	1
Thumbnail generator	Low	Medium	Low	Low	2
Facial recognition	Medium	Medium	Low	Low	5
Todo API	Low	Low	Low	Low	5
Image processing	Medium	Medium	Low	Low	2
Video processing	High	High	Medium	High	2

Table 5
Hyper-parameters used for DRL model training.

Parameter	Value
General	
Discount factor (γ)	0.95
Mini-batch size	64
Replay buffer size	2000
N-step buffer size	5
Target network update rate	100
Replay memory size to start training	100
Epsilon max (ϵ_{max})	1
Epsilon min (ϵ_{min})	0.04
Epsilon decay factor at each time step	0.999
Neural network parameters	
Learning rate (α)	0.001
No. of input layers	1
No. of output layers	1
No. of hidden layers	2
No. of neurons in each hidden layer	100/200
Optimizer	Adam

resource efficiency. This is an approach often followed in cluster scheduling scenarios to help resolve congestion [48].

Dynamic Time Cost Aware (DTCA): A scheduler similar to STCA but uses state parameters derived by the DRL agent in a dynamic manner to select a VM. Ranking and scoring of VMs is done as in STCA but the decision of choosing the highest or lowest overall score is taken based on the Relative Function Response Time (RFRT) and p^k_{con} calculated using Eq. (6), at the time of scheduling the function instance. If either RFRT or p^k_{con} is higher than the average value of all the deployed functions, we choose the VM with the lowest overall score (DTCA-L) and else, the highest one (DTCA-H).

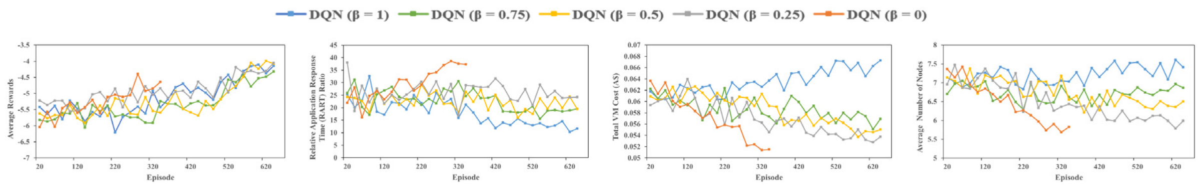
LZ-based: We adapt ENSURE's [5] latency zone based request scheduling policy for our instance scheduling problem. Accordingly, if RFRT of the function in consideration is lower than a given latency threshold (we consider a value of 1.25), the cluster is considered to be in a safe/prewarning zone (with regard to that function) and the maximum number of replicas of that function scheduled on a VM is limited to the number of vCPU cores it has. If RFRT is higher than that, the cluster is pushed to a warning zone and only a maximum of a single replica of that function is scheduled on each VM.

KC: We use the K-means ++ unsupervised machine learning algorithm [49] to derive a cluster interpretation of function instances based on their resource consumption. During scheduling, we avoid co-locating those belonging to the same cluster on a VM. We collect the CPU, memory, network and disk I/O resource utilization metrics of function instances of the selected applications under varying request arrival rates and input parameters, normalize them and use this data to perform clustering. The number of clusters is determined using the elbow method.

6.4. Convergence of the DRL model

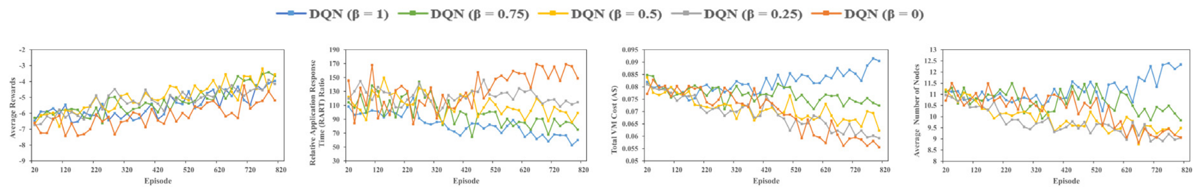
For each cluster size, we train the DRL model under five scenarios defined by the parameter β (as given in Eq. (9)), which identifies the level of trade-off between the two optimization objectives. A higher value of β indicates that the agent is incentivized more for improving the function response time, while a lower value indicates increased reward for the agent for optimizing VM usage cost. Accordingly, $\beta = 1$ implies that the awarded reward is solely dependent on function response time while the focus is only on improving VM cost efficiency when $\beta = 0$. Figs. 4 and 5 illustrate the step by step progress achieved by the DRL agents under each scenario, in the process of learning to take actions which lead to the accomplishment of the desired objectives. The training progress is demonstrated in terms of episodic reward, sum of relative application response time ratio, VM usage cost and the average number of nodes used during an episode. Note that in each of these graphs we have plotted the average value over 20 iterations for ease of observation of the training progress. We train the model for five times under each scenario using the hyper-parameters stated in Table 5 and select the model that gives the best results for conducting the evaluation experiments.

Figs. 4(a) and 5(a) show how the total reward captured during an episode improves and gradually converges under varying β parameters. In the 10 VM cluster, when $\beta = 1$, the model converged around the 600th episode, and the training took about 60 h. In the 20 VM cluster, the same model converged around the 800th episode, requiring approximately 80 h of training due to the expanded state space. Since here the full focus is on improving the function response time, we see a steady decrease in RART in the corresponding graphs in 4(b) and 5(b) as training progresses. In contrast, in the corresponding graphs in 4(c), 5(c) and 4(d), 5(d) we see a gradual increase in the VM usage cost and the average number of nodes used. This is because the agent learns through experience that using more nodes with higher resource availability to host different function instances leads to lesser resource congestion. This results in higher VM costs with the partial usage of nodes with higher resource capacities and hourly charges. Similarly, in the $\beta = 0$ scenario we observe a steady reduction in VM usage costs and the number of nodes (Figs. 4(c), 5(c) and 4(d)), 5(d) while the RART deteriorates visibly (Figs. 4(b) and 5(b)). It is also seen that during this scenario in the 10 VM cluster, the model convergence is relatively faster, with reward getting stabilized around the 300th episode which required about 30 h of training. This is because, the cost efficiency objective is easily achieved by primarily learning to use already active nodes more frequently. In comparison, finding the best



(a) Reward Convergence (b) RART Ratio (c) Total VM Cost (d) Average Number of Nodes

Fig. 4. Convergence process of the trained DRL models in the 10 VM cluster in terms of reward, RART ratio, total VM cost, and the average node number.



(a) Reward Convergence (b) RART Ratio (c) Total VM Cost (d) Average Number of Nodes

Fig. 5. Convergence process of the trained DRL models in the 20 VM cluster in terms of reward, RART ratio, total VM cost, and the average node number.

policy to improve application performance requires the agent to learn the different congestion levels created by the co-location of different functions with dynamic workload patterns, and also the effects of various environmental parameters. The three models focused on improving both the target objectives ($\beta = 0.75$, $\beta = 0.5$, $\beta = 0.25$) too converge around the 600th episode for the 10 VM scenario while the 20 VM cluster requires training for approximately 800 iterations for the same models. Since these two are conflicting objectives, giving a higher significance to one, impedes the training progress of the other as seen in the convergence graphs for these scenarios in 4(b), (c), (d) and 5(b), (c), (d). When $\beta = 0.75$, a significant improvement is seen in RART at convergence, while the improvement in VM cost is marginal. In contrast, the $\beta = 0.25$ scenarios record a notable improvement in VM cost, while the corresponding optimization of response time is minimal. The models converge with average improvements in both the parameters when $\beta = 0.5$.

6.5. *Analysis of model performance on the evaluation data sets*

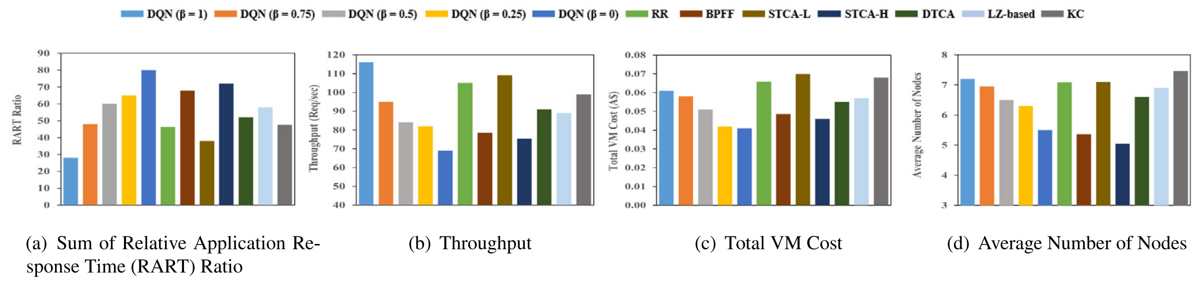
The performance evaluation of the trained models under the two clusters of VMs, is conducted across different request traffic levels. We create dynamic workloads for model evaluation by using 900 function traces from Azure Functions data set, extracted using the same mechanism as described in Section 6.1.2. These traces are used to create 150 different workloads in total with 50 workloads each having request arrival rates ranging between 5–20, 20–40 and 40–60 requests per second respectively. The 10 VM cluster is evaluated using the set of workloads with arrival rates between 5–20, while the 20 VM cluster is tested with all three sets of workloads. Each individual workload comprises of concurrent requests arriving for multiple applications (comprising of single or multiple functions), at varying arrival rates (ranging between 5–60 requests/s overall), for a duration of 5 min. All the evaluation parameters in Figs. 6 and 7 represent averaged values over runs of the 50 different workloads under each scenario. The separate analyses of model performance under the two cluster setups demonstrate the scalability and robustness of the proposed model across expanded state parameters. Overall, the performance of our proposed model in comparison with the baseline algorithms, is discussed under the two optimization objectives of application response time and resource cost efficiency.

6.5.1. *Evaluation of application response time*

We discuss application response time performance in association with the RART ratio and system throughput.

10 VM Cluster: Fig. 6(a) demonstrates the comparison of the performance of our trained models with the baselines in terms of the total RART ratio. The DQN ($\beta = 1$) model shows the best performance in terms of application performance among all the algorithms, with a 24% improvement in RART ratio over the next best performing algorithm STCA-L. This is also reflected in the corresponding throughput graphs in Fig. 6(b). Under the $\beta = 1$ scenario, the agent is constantly incentivized to avoid performance degradation caused by resource pressure. Thus it has developed a superior understanding of the congestion levels caused by each function instance on the host node at different request traffic levels and various node resource conditions. This has led to establishing the best policy to choose the host node with minimum contention.

As expected, our DQN ($\beta = 0$) model performs worst in terms of response time since the agent is trained to fully focus on improving resource cost efficiency and thus largely compromises on application performance. This is demonstrated by the RART ratio in Fig. 6(a) and the throughput graph in Fig. 6(b). BPFF and STCA-H algorithms too show poor performance in terms of response time and STCA-H has the lowest system throughput next to DQN ($\beta = 0$). Both these methods tend to place new function instances on VMs that are most congested, causing increased competition for node resources. RR algorithm performs relatively better as each consecutive function instance is spread among the cluster VMs. But since this only leads to randomly balancing the load among the nodes without an understanding on specific function or system characteristics, the achieved results are sub-optimal. KC shows similar performance to RR. At lower load levels, we observed that most data points in the K-means clustered data based on resource usage, belonged to the same cluster, thus resulting in a RR like function scheduling pattern. The STCA-L algorithm depends on static state parameters of the system in taking scheduling decisions. Although the decisions made under this method leads to relatively good results, this technique is not competitive enough to find the most optimum solution since it possesses no overall understanding on the complex system dynamics. The performance of DTCA and LZ-based strategies are mostly comparable with that of DQN($\beta = 0.75$, $\beta = 0.5$, $\beta = 0.25$) models since they try to balance both the objectives. Out of these DQN($\beta = 0.75$) outperforms the rest but is closely followed



(a) Sum of Relative Application Response Time (RART) Ratio (b) Throughput (c) Total VM Cost (d) Average Number of Nodes

Fig. 6. Comparison of the RART ratio, throughput, total VM cost and the average number of used nodes in the system during an episode, by the DRL model and the baseline algorithms in the 10 VM cluster.

by DTCA and LZ-based since the response time delays still get priority in their scheduling decisions whereas DQN($\beta = 0.5$) is incentivized to optimize both equally.

20 VM Cluster: Fig. 7 exhibits the relative performance of our trained models on the 20 VM cluster in comparison to baseline algorithms. On this cluster we conduct experiments under three levels of request arrival rates to applications as shown. As the user request rates increase, an overall increase in resource congestion and as a result, a degradation of application response times is seen (Fig. 7(a)).

At the lowest request traffic level of 5–20 req/s, the cluster is able to serve all the requests with minimum pressure on its resources. In this situation when the cluster is relatively relaxed, a complex understanding on the underlying application characteristics seem to provide only minimal added benefits. As a result, the STCA-L algorithm gives the best performance in terms of RART, while the DQN ($\beta = 1$) model closely follows. DTCA and RR algorithms show similar performance to DQN ($\beta = 0.75$) and DQN ($\beta = 0.5$) models, followed by KC. LZ-based algorithm shows poor performance since a fixed latency threshold for applications regardless of the request arrival rates, is not able to make good decisions under dynamic load conditions. DQN ($\beta = 0$) model shows worst performance since it only focuses on resource cost efficiency. The throughput graph for the 5–20 request range too reflect the RART performance, but since the request arrivals are sparse, the difference seen among the scheduling algorithms is less significant.

With the increase in the load level at 20–40 req/s, DQN ($\beta = 1$) model shows a 17% improvement in RART over the next best performing algorithm STCA-L. Since the DQN agent is trained to identify application resource characteristics at a given load level and the cluster status, it is able to avoid resource contention on host nodes in the most optimum way. STCA-L algorithm performs well due to its inherent tendency to choose host nodes with least request traffic. LZ-based algorithm too performs fairly well in this scenario since with the increased load level, the considered latency threshold has been able to make comparatively better decisions. Results indicate that scheduling functions based on identified cluster patterns in the KC algorithm is not granular or robust enough to understand system reactions to resource pressure well, and hence is not able to manage the resulting impact on application performance. DTCA algorithm suffers from poor decision making when the overall cluster resource pressure increases, due to its dependency on average cluster RFRT and request concurrency. It is also observed that with increased traffic levels, DQN ($\beta = 0.75$), ($\beta = 0.5$) and DQN ($\beta = 0.25$) models which aim at balancing the dual objectives, show relatively distinct performances with regard to application performance. Throughput graphs for this scenario too show more significant improvements in line with the response time performance, compared to the previously discussed low load level scenario.

At the highest level of request rates, DQN ($\beta = 1$) model demonstrates a 20% improvement in RFRT compared to STCA-L. The response time behavior of the other baseline scheduling algorithms under this scenario is mostly similar to that of the 20–40 load level.

6.5.2. Evaluation of resource cost efficiency

The efficiency in resource usage is primarily measured in terms of the total VM usage cost.

10 VM Cluster: Fig. 6(c) illustrates the performance of our trained models when compared with the baseline solutions in terms of resource cost. When $\beta = 0$, the DQN agent is encouraged solely to use low cost resources and maintain higher utilization levels of the used resources, which results in overall lower VM usage cost. The derived policy from training the agent, tries to strategically place new functions to already used, low cost VMs as much as possible. The results from DQN ($\beta = 0.25$) model too closely resonates with that of DQN ($\beta = 0$), and together they show the best performance. DQN ($\beta = 0$) model results in a 11% and 15% lesser VM usage cost compared to the next best performing non-DRL techniques of STCA-H and BPFF. The lower resource consumption is also reflected in the average number of used VMs as shown in Fig. 6(d), where the average number of used VMs for DQN ($\beta = 0$) is among the lowest.

STCA-L algorithm shows the highest VM usage cost and also rank high in terms of the average number of nodes used, which reflect worst performance. That is because its strategy is to use the system parameters to determine high capacity nodes with least number of running functions and minimum resource utilization, and use them for function scheduling. This leads to more cluster nodes often operating drastically below their capacities. The next highest resource cost is seen in RR, KC and in DQN ($\beta = 1$) algorithms. RR algorithm understandably results in low resource efficiency since it is not sensitive to any variations in incoming workloads or cluster resource conditions. It simply schedules functions on VMs cyclically, and this inadvertently results in most VMs being active throughout the experiment. KC algorithm behaves mostly in a similar manner due to irregularities in cluster formations at low load levels. DQN ($\beta = 1$) on the other hand is trained to focus fully on avoiding resource contention among functions and thus consumes more resources in the process. BPFF naturally tries to pack as many functions as possible to one VM before moving on to the next one, while STCA-H manoeuvres system parameters to find low cost VMs that already have a high utilization. The result is lower VM usage cost overall since this minimizes under-utilization of VMs, specially with high capacities. STCA-H and BPFF also result in the lowest average number of VMs being used, even lower than that of DQN ($\beta = 0$). Even though in comparison to DQN ($\beta = 0$), these techniques incur a higher resource cost, this could still occur because the lower number of used nodes could be having higher unit time cost. DQN ($\beta = 0.75$) is high in VM cost due

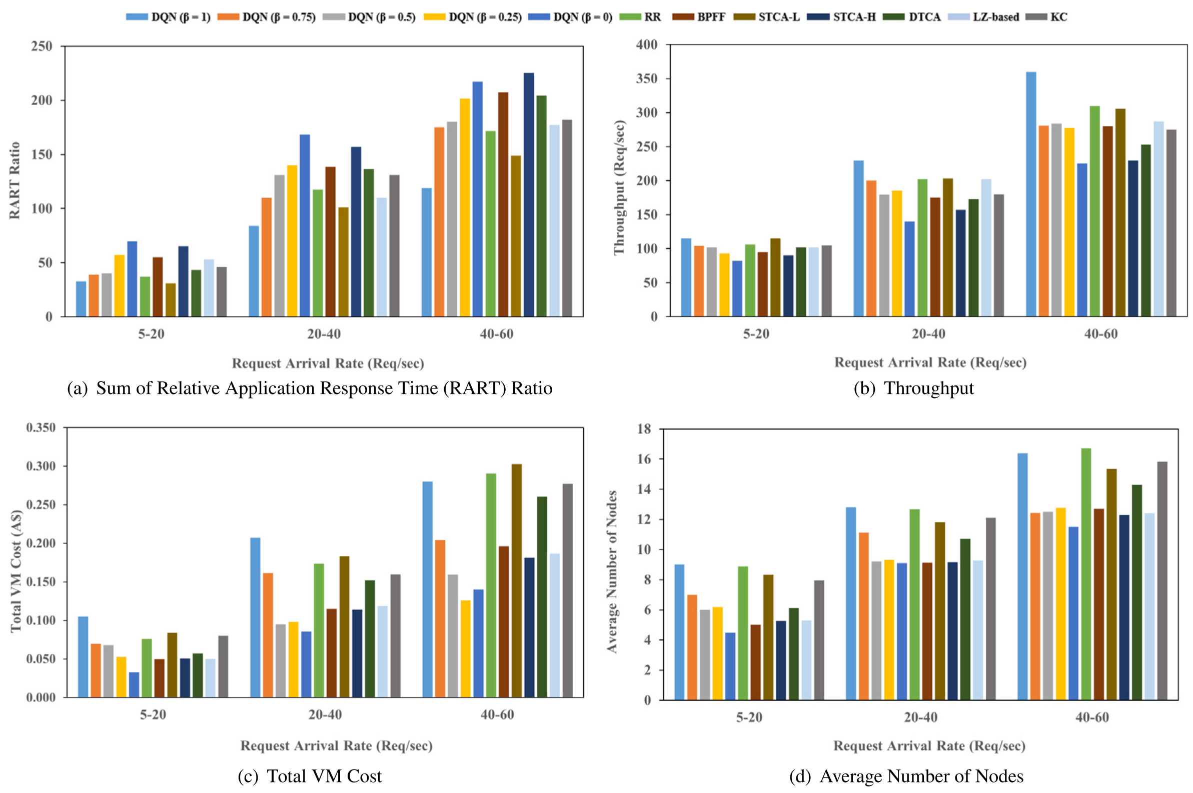


Fig. 7. Comparison of the RART ratio, throughput, total VM cost and the average number of used nodes in the system during an episode, by the DRL model and the baseline algorithms in the 20 VM cluster.

to being biased towards response time improvement, while DQN ($\beta = 0.5$) is the best at balancing both the objectives, performing better than the other non-DRL dual objective oriented techniques of DTCA and LZ-based algorithms in terms of cost efficiency.

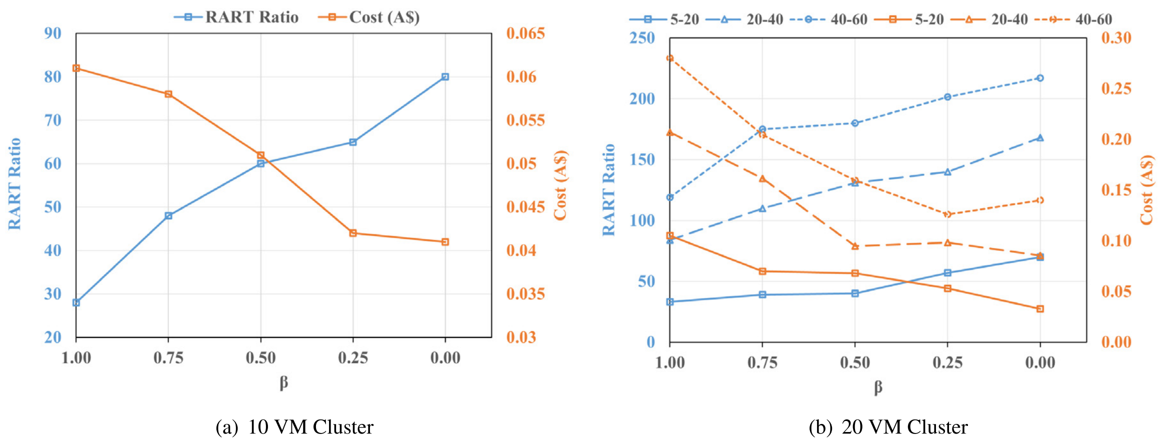
20 VM Cluster: The resource cost efficiency of the 20 VM cluster under varying load levels is illustrated in Fig. 7(c). In the first scenario with request rates ranging from 5–20, DQN ($\beta = 0$) model demonstrates a 34% reduction in VM usage costs, outperforming the best among the baseline algorithms, BPFF. In contrast to minimal improvements to application response time by the DQN agents at lower traffic levels as discussed earlier, the high cost savings is due to increased opportunity to keep high cost VMs from running since the cluster has plenty of other resources to accommodate the incoming requests. STCA-H, LZ-based and DTCA algorithms incur slightly higher VM costs compared to BPFF. At lower traffic levels, these baselines are not able achieve optimum resource efficiency without the combined knowledge of application workload characteristics and cluster resource levels. DQN ($\beta = 1$) agent shows the highest resource cost since it uses its workload and system awareness on spreading function instances on VMs with the highest free resource capacities (high cost VMs). The average number of VMs used in a scheduling episode under this scenario (Fig. 7(d)) mostly reflect a behavioral pattern comparable with VM costs, although there are deviations since the used VM count will not move directly in line with the objective of cost reductions in a heterogeneous cluster.

At 20–40 req/s, DQN ($\beta = 0$) still achieves the best performance with a 25% reduction in VM costs compared to the next best performing baseline algorithms of BPFF, STCA-H and LZ-based. An interesting observation is that as the load levels grows, even the DQN ($\beta = 0.25$) and DQN ($\beta = 0.5$) agents achieve noticeably high cost benefits, compared to baselines. This is because, as these DQN agents try to optimize dual objectives, the achieved response time improvements too contribute to lowering the infrastructure costs, as the applications require lesser time for their executions. RR and STCA-L algorithms result in high node costs due their inherent quality of spreading function instances among VMs without an elaborate understanding on workload and system interactions. KC scheduling policy is focused only on avoiding VM resource pressure and thus performs poorly in terms of resource efficiency.

At the highest level of request traffic under the 3rd scenario, surprisingly the DQN ($\beta = 0.25$) model outperforms its counterpart DQN ($\beta = 0$) agent which is solely focused on cost improvements. As discussed previously, this is further evidence that under high pressure on node resources, taking both objectives into consideration leads to training a policy which is better at optimizing cost more effectively in the long run. The relatively poor performance of BPFF and STCA-H algorithms which are generally good at packing function instances to save costs, also demonstrate the underlying indirect effect of application performance on cost performance in an overloaded cluster. Further, compared to baselines such as BPFF and STCA-H, the DQN ($\beta = 0$) and DQN ($\beta = 0.25$) agents show only a marginal difference in the average number of VMs in usage. This further establishes the fact that during high load levels, the achieved cost efficiencies are largely due to the intelligent placing of different application instances on suitably low cost host nodes, since simply packing them on to fewer VMs has only a limited ability to improve costs.

6.5.3. Evaluation of multiple reward maximization

Fig. 8(a) and (b) illustrate the movement of the optimized objectives with the change in the β parameter in the 2 cluster scenarios. The blue lines exhibit the effects on application response time while the orange lines present the effects on resource cost. On the 20 VM graph, the solid lines, dashed lines and the dotted lines represent the obtained results with regard to 5–20, 20–40 and 40–60 request load levels. As seen, the DQN agents



(a) 10 VM Cluster (b) 20 VM Cluster

Fig. 8. The effect of the β parameter in optimizing dual objectives in DRL model training.

are able to achieve stable results while optimizing one or more objectives as desired. Higher the β value, the trained agents are better at improving application response time, while lower β value indicates better ability to control VM usage costs. In each scenario, at $\beta = 0.5$, the agents display a balanced policy which is able to optimize both the objectives to a satisfactory level.

6.6. DRL model training and serving overhead

In this work all our DRL models are trained on a practical testbed. Unlike in a simulator where the time steps will generally be determined by an event based clock, in our practical set up, the time consumed is equivalent to the actual resource creation and execution times of the applications. Accordingly, the model training time is composed of these actual environmental set up and function run times, coupled with the overhead of using a neural network for deep learning, for each training episode until model convergence. The neural network overhead for model training is dependent on the modeled environment's state size, action space and the complexity of the agent's reward structure. Thus, as described under Section 6.4, we observe varying model training times with changing cluster sizes and the β parameter, which determines the reward structure. For the 10 VM cluster, the $\beta = 1$, $\beta = 0.75$, $\beta = 0.5$ and $\beta = 0.25$ scenarios all require approximately 60 h of training while the $\beta = 0$ scenario experiences faster convergence at half that time owing to having a simpler reward structure. Due to increased state exploration costs, the model requires 80 h of training on average to reach convergence for the 20 VM cluster.

In order to observe optimum scheduling results under more diverse function resource requirements, request arrival rates, expanded cluster sizes and changed optimization objectives, the model could be easily retrained by providing the required exploration data to the agent. Model scalability in this manner is largely demonstrated in our experiments which analyze its adaptability under cluster size and reward structure variations.

Model serving for the proposed DRL agent refers to mapping of the current environmental state to an action, which is derived based on the state–action values of the available actions in the trained model. Since model training takes place offline, we observe that this mapping consumes an insignificant time of about 33 ms on average, which is the scheduling overhead imposed. The model evaluation experiments show that this value is similar to the time spent on scheduling decisions of the other non-DRL baselines as well.

7. Conclusions and future work

The serverless computing model gives rise to flexibility in resource management for both the cloud provider and the end users. However, the multi-tenant nature of these computing environments could cause complex variations in function performance, when application demand levels are subject to rapid changes over time, due to resource constraints. At the same time, efficient usage of the underlying infrastructure has become increasingly important for the cloud providers with the advent of the "pay as you execute" billing modes. In this work we proposed a DRL based technique, which is trained and evaluated on a practical cloud setup, for efficiently understanding how the various system parameters of a VM cluster and the highly dynamic parameters of an incoming serverless workload interact with each other and affect application performance. We also strived to achieve a second objective of maintaining high resource cost efficiency, where the users are at liberty to set a desired level of significance to each of these often conflicting objectives. As evidenced by our experiments, we see that such granular approaches to understanding the system dynamics could immensely help both users and cloud providers to achieve their end goals.

As part of the future work, we will explore the possibility of incorporating a multi-agent DRL architecture to improve the efficiency of the training process and enhance the model adaptability to changing cluster conditions. This would allow us to investigate strategies to reduce the dependence of model training complexity on the scale of the cluster, and hence focus more on objective optimization.

CRediT authorship contribution statement

Anupama Mampage: Conceptualization, Methodology, Software, Validation, Resources, Writing - original draft, Writing - review & editing. **Shanika Karunasekera:** Conceptualization, Writing - review & editing, Supervision. **Rajkumar Buyya:** Conceptualization, Writing - review & editing, Supervision, Funding acquisition.

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] OpenFaaS, Home | OpenFaaS - Serverless functions made simple, 2022, https://www.openfaas.com/. (Accessed on 04/08/2022).
[2] T.K. Authors, Home - knative, 2022, https://knative.dev/docs/. (Accessed on 04/08/2022).
[3] F. Project, Fission, 2022, https://fission.io/. (Accessed on 04/08/2022).
[4] L. Wang, M. Li, Y. Zhang, T. Ristenpart, M. Swift, Peeking behind the curtains of serverless platforms, in: Proceedings of the USENIX Annual Technical Conference, ATC, 2018, pp. 133–146.
[5] 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.
[6] A. Mampage, S. Karunasekera, R. Buyya, Deadline-aware dynamic resource management in serverless computing environments, in: Proceedings of the IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), IEEE, 2021, pp. 483–492.
[7] M. HoseinyFarahabady, Y.C. Lee, A.Y. Zomaya, Z. Tari, A qos-aware resource allocation controller for function as a service (faas) platform, in: Proceedings of the International Conference on Service-Oriented Computing, Springer, 2017, pp. 241–255.
[8] Y.K. Kim, M.R. HoseinyFarahabady, Y.C. Lee, A.Y. Zomaya, Automated fine-grained cpu cap control in serverless computing platform, IEEE Trans. Parallel Distrib. Syst. 31 (10) (2020) 2289–2301.
[9] M. Stein, Adaptive Event Dispatching in Serverless Computing Infrastructures (Ph.D. thesis), Brunel University London, 2018.
[10] G. Aumala, E. Boza, L. Ortiz-Avilés, G. Totoy, C. Abad, Beyond load balancing: Package-aware scheduling for serverless platforms, in: Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID, IEEE, 2019, pp. 282–291.
[11] A. Das, A. Leaf, C.A. Varela, S. Patterson, Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications, in: Proceedings of the 13th IEEE International Conference on Cloud Computing, CLOUD, IEEE, 2020, pp. 609–618.
[12] 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.
[13] H. Yu, H. Wang, J. Li, S.-J. Park, Harvesting idle resources in serverless computing via reinforcement learning, 2021, arXiv preprint arXiv:2108.12717.
[14] 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.
[15] Kubeless, Kubeless, 2021, https://kubeless.io/. (Accessed on 01/13/2022).
[16] Kubernetes, Kubernetes, 2022, https://kubernetes.io/. (Accessed on 04/08/2022).
[17] 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.
[18] 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.
[19] M. Shahrad, R. Fonseca, Í. 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.
[20] K. Kaffes, N.J. Yadwadkar, C. Kozyrakis, Centralized core-granular scheduling for serverless functions, in: Proceedings of the ACM Symposium on Cloud Computing, 2019, pp. 158–164.
[21] 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.
[22] N. Mahmoudi, C. Lin, H. Khazaei, M. Litoiu, Optimizing serverless computing: introducing an adaptive function placement algorithm, in: Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, 2019, pp. 203–213.
[23] J.R. Gunasekaran, P. Thinakaran, N.C. Nachiappan, M.T. Kandemir, C.R. Das, Fifer: Tackling resource underutilization in the serverless era, in: Proceedings of the 21st International Middleware Conference, 2020, pp. 280–295.
[24] 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.
[25] P. Żuk, B. Przybylski, K. Rzadca, Call scheduling to reduce response time of a faas system, 2022, arXiv preprint arXiv:2207.13168.
[26] A. Fuerst, P. Sharma, Locality-aware load-balancing for serverless clusters, in: Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing, 2022, pp. 227–239.
[27] K. Kaffes, N.J. Yadwadkar, C. Kozyrakis, Hermod: principled and practical scheduling for serverless functions, in: Proceedings of the 13th Symposium on Cloud Computing, 2022, pp. 289–305.
[28] S. Ristov, P. Gritsch, FaaSt: Optimize makespan of serverless workflows in federated commercial FaaS, in: 2022 IEEE International Conference on Cluster Computing, CLUSTER, IEEE, 2022, pp. 183–194.
[29] H. Tian, S. Li, A. Wang, W. Wang, T. Wu, H. Yang, Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud, in: Proceedings of the 13th Symposium on Cloud Computing, 2022, pp. 78–93.
[30] V.M. Bhasi, J.R. Gunasekaran, A. Sharma, M.T. Kandemir, C. Das, Cypress: input size-sensitive container provisioning and request scheduling for serverless platforms, in: Proceedings of the 13th Symposium on Cloud Computing, 2022, pp. 257–272.
[31] A. Asghari, M.K. Sohrabi, F. Yaghmaee, A cloud resource management framework for multiple online scientific workflows using cooperative reinforcement learning agents, Comput. Netw. 179 (2020) 107340.
[32] A. Asghari, M.K. Sohrabi, F. Yaghmaee, Task scheduling, resource provisioning, and load balancing on scientific workflows using parallel SARSA reinforcement learning agents and genetic algorithm, J. Supercomput. 77 (3) (2021) 2800–2828.
[33] Y. Wang, H. Liu, W. Zheng, Y. Xia, Y. Li, P. Chen, K. Guo, H. Xie, Multi-objective workflow scheduling with deep-Q-network-based multi-agent reinforcement learning, IEEE Access 7 (2019) 39974–39982.
[34] Y. Qin, H. Wang, S. Yi, X. Li, L. Zhai, An energy-aware scheduling algorithm for budget-constrained scientific workflows based on multi-objective reinforcement learning, J. Supercomput. 76 (1) (2020) 455–480.
[35] Z. Peng, D. Cui, J. Zuo, Q. Li, B. Xu, W. Lin, Random task scheduling scheme based on reinforcement learning in cloud computing, Cluster Comput. 18 (4) (2015) 1595–1607.
[36] S. Agarwal, M.A. Rodriguez, R. Buyya, A reinforcement learning approach to reduce serverless function cold start frequency, in: Proceedings of the IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), IEEE, 2021, pp. 797–803.
[37] 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.
[38] Docker, Runtime options with memory, CPUs, and GPUs | Docker documentation, 2021, https://docs.docker.com/config/containers/resource_constraints/. (Accessed on 10/16/2020).
[39] P.-H. Chiang, H.-K. Yang, Z.-W. Hong, C.-Y. Lee, Mixture of step returns in bootstrapped DQN, 2020, arXiv preprint arXiv:2007.08229.
[40] MRC, Melbourne research cloud documentation, 2022, https://docs.cloud.unimelb.edu.au/. (Accessed on 07/22/2022).
[41] ARDC, ARDC nectar research cloud - ARDC, 2022, https://ardc.edu.au/services/nectar-research-cloud/. (Accessed on 07/22/2022).
[42] A.S. Foundation, Apache CouchDB, 2022, https://couchdb.apache.org/. (Accessed on 07/22/2022).
[43] Prometheus, Prometheus - Monitoring system & time series database, 2022, https://prometheus.io/. (Accessed on 09/08/2022).
[44] A.S. Foundation, Apache JMeter - Apache JMeter™, 2021, https://jmeter.apache.org/. (Accessed on 11/08/2021).
[45] Keras, Keras: the Python deep learning API, 2021, https://keras.io/. (Accessed on 01/20/2022).
[46] TensorFlow, TensorFlow, 2021, https://www.tensorflow.org/. (Accessed on 01/20/2022).
[47] AWS, AWS pricing calculator, 2022, https://calculator.aws/#/addService/EC2. (Accessed on 07/25/2022).
[48] A. Kuriata, R.G. Illikkal, Predictable performance for QoS-sensitive, scalable, multi-tenant function-as-a-service deployments, in: Proceedings of the International Conference on Agile Software Development, Springer, 2020, pp. 133–140.
[49] R.C. Chiang, Contention-aware container placement strategy for docker swarm with machine learning based clustering algorithms, Cluster Comput. (2020) 1–11.

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 with the School of Computing and Information Systems, 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 625 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=149, g-index=322, 116,700+ citations).