Autonomous Resource Management for Serverless Computing

Anupama Mampage

Submitted in total fulfilment of the requirements of the degree of

Doctor of Philosophy

School of Computing and Information Systems
THE UNIVERSITY OF MELBOURNE

November 2023

ORCID: 0000-0002-8155-3584

Autonomous Resource Management for Serverless Computing

Anupama Mampage
Principal Supervisor: Prof. Rajkumar Buyya
Co-Supervisor: Prof. Shanika Karunasekera

Abstract

Serverless computing is gaining momentum as the latest cloud deployment model for applications, with many major global companies shifting towards a complete adoption of this new computing paradigm. As the name implies, the burden of server management is non-existent to the end user under this model, with the cloud vendor taking full responsibility of infrastructure management. The users simply deploy the logic of their application in the form of code segments called 'functions', with a rough estimate on the resource requirements and the rest is taken care of by the serverless platform. This greatly reduces the time-to-market and upfront costs for client products with no expertise required in initial server configurations and subsequent operations. The rapid auto-scalability feature allows customers to scale their businesses fast, without the need for any prior infrastructure requirement planning. The pay-per-use billing model enables a fair ground for applications with spontaneous fluctuations in traffic loads.

However, the 'serverless' nature to the end user unequivocally leaves the entire set of end to end server management responsibilities with the cloud vendor. In contrast to conventional Infrastructure-as-a-Service (IaaS) cloud model, where the provider only handles the physical infrastructure maintenance, now the complete virtual machine maintenance is also part of the provider service offering. Moreover, unlike the users managing their allocated resources for the execution of their own applications, the cloud provider has to undertake the same set of tasks with far lesser knowledge available to them. Not only do they have to successfully manage the infrastructure for an application belonging to a separate party, but they need to accommodate the needs of thousands of user applications on the same shared platform, considering the impact of their co-resident behaviors as well. Thus this is a tremendous feat for cloud vendors if accomplished to the satisfaction of all the parties involved. Further, while auto-scaling and granular billing features are vastly favorable to end users, the cloud vendors are at a grave disadvan-

tage if measures are not taken to maintain high efficiency in their underlying resources. Hence, having proper techniques for resource management in place which are capable of meeting all of these challenges is of utmost importance. The existing commercial and open-source serverless platforms mostly follow primitive resource management policies at the moment, which have a vast potential to be optimized. Further, although there is a huge interest in the research community in this area of study, most of the existing works are limited in their generalizability to be adaptable to practical serverless computing environments, considering the multi-tenant and rapidly changing nature of these systems. Moreover, a vast majority of these works are negligent on the service provider perspective of their offered solutions, which is a key factor determining the probable adoption of the same in vendor platforms.

This thesis investigates novel techniques for the smooth running of all resource handling operations including resource provisioning, resource scheduling and scaling, which are dynamic and intelligent enough to handle the complexities of this computing environment. Proposed approaches strive to gain a thorough understanding on the behavior of the serverless computing infrastructure, along with the different application workloads, in developing strategies beneficial for both end users and cloud vendors. In essence, this thesis advances the state-of-the-art by making the following contributions:

1. A comprehensive taxonomy and literature review on the aspect of resource management in serverless computing environments, along with a discussion on identified research gaps, laying the ground work for future research work.

2. A dynamic resource management and a function request placement technique for meeting user specific application requirements and maintaining high resource efficiency.

3. A Deep Reinforcement Learning (DRL) based workload and system aware technique for scheduling applications in resource-constrained, multi-tenant serverless computing environments, along with flexibility in achieving a desirable level of application performance and resource cost optimization.

4. A framework for horizontally and vertically scaling allocated resources to func-

tion instances based on a multi-agent DRL model for elevating application performance while maintaining provider resource cost efficiency.

5. A deployment and scheduling approach for applications in a hybrid serverless and serverful environment based on DRL, with the aim of reducing application latency and incurred user cost.

6. A detailed discussion outlining challenges and the potential for future work for the efficient use of serverless computing environments.

Declaration

This is to certify that

1. the thesis comprises of only my original work towards the PhD,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Anupama Mampage, November 2023

Preface

Main Contributions

This thesis research has been carried out in the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne. The main contributions of the thesis are discussed in chapters 2-6 and are based on the following publications:

- **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)*, Volume 54, Issue 11s, Article 222, 36 pages, September 2022.

- **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "Deadline-aware dynamic resource management in serverless computing environments", *Proceedings of the 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, Pages: 483-492, Melbourne, Australia, May 10-13, 2021.

- **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "Deep reinforcement learning for application scheduling in resource-constrained, multi-tenant serverless computing environments", *Future Generation Computer Systems (FGCS)*, Volume 143, Pages 277-292, June 2023.

- **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "A Deep Reinforcement Learning based Algorithm for Time and Cost Optimized Scaling of Serverless Applications", *Future Generation Computer Systems (FGCS)* [Under Re-

view, April 2024].

- **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "Deep Reinforcement Learning for Scheduling Applications in a Serverless and Serverful Hybrid Environment", *IEEE Transactions on Services Computing (TSC)* [Under Review, November 2023].

Supplementary Contributions

During the Ph.D. candidature, I have also completed the following work (this thesis does not claim it as its contributions):

- **Anupama Mampage** and Rajkumar Buyya, "CloudSimSC: A Toolkit for Modeling and Simulation of Serverless Computing Environments", *Proceedings of the 25th IEEE International Conference High Performance Computing and Communications (HPCC), Melbourne, Australia, Dec 13-15, 2023*.

Acknowledgements

Ph.D. is a long and challenging journey, which would have been neither successful, nor enjoyable, if not for the help and guidance of many, every step of the way. Upon reaching almost the very end of this journey, I would like to take this opportunity to express my heartfelt gratitude to all the wonderful people who truly made it possible.

First and foremost, I would like to thank my principal supervisor, Professor Rajkumar Buyya for giving me the opportunity to pursue my PhD at the University of Melbourne under his guidance. His continuous guidance and encouragement on my research work as well as his positive attitude on all aspects of life have helped me tremendously throughout this journey. I would also like to extend sincere gratitude to my co-supervisor Professor Shanika Karunasekera for her invaluable mentorship during all these years. Her kindness and patience as a person as well as the constructive feedback on all my works is highly appreciated. I also take this opportunity to thank my advisory committee chair, Professor Lars Kulik for his constant guidance during progress meetings for staying on track, and support towards the successful completion of this study.

I would also like to thank all the past and current members of the CLOUDS Laboratory at the University of Melbourne, whose companionship and support at times of need have been invaluable. In particular, I thank Dr. Muhammed Tawfiqul Islam, Dr. Mohammad Goudarzi, Dr. Shashikant Ilager, Dr. Redowan Mahmud, Dr. Samodha Pallewatta, Dr. Maria Rodriguez, Dr. Muhammad Hilman, Dr. TianZhang He, Amanda Jayanetti, Zhiheng Zhong, Jie Zhao, Ming Chen, Tharindu Bandara, Siddharth Agarwal, Thanh-Hoa Nguyen, Yulun Huang, Zhiyu Wang, Kalyani Pendyala, Duneesha Fernando, Qifan Deng, and TianYu Qi for their support.

I am immensely grateful to the University of Melbourne for providing me with the scholarship and resources to pursue my doctoral studies and I acknowledge the Australian Federal Government and the Australian Research Council (ARC) for enabling the funding and supporting my PhD research.

I would also like to thank the past and present admin staff of the School of Computing and Information Systems for their continued support in resolving queries throughout my candidature.

I would like to extend my heartfelt gratitude to my parents Wasantha Mampage

and Ramya Attanayake, and my parents-in-law Susil Samarasinghe and Sunethra Chandrakanthi for their unconditional love and support.

Last, but most importantly, I am extremely grateful to my husband Sayuru Samarasinghe, for being the closest to my heart, with endless love, without which this arduous but amazing journey would not have been possible.

Anupama Mampage
Melbourne, Australia
November 2023

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Serverless computing is emerging as a novel and compelling paradigm for the deployment of cloud applications. This could partially be attributed to the recent shift of enterprise application architectures from monolithic structures to more fine-grained individual units i.e. microservices [1]. Although the first usage of the term "serverless" seems to have appeared around 2012, the concept gained more popularity in 2015, following AWS's serverless platform launch in 2014. Today it is gaining rapid popularity with a Compound Annual Growth Rate (CAGR) of 21% and a market size of 9.3 billion in 2022 which is estimated to reach 28.9 billion by 2028 [2].

The concept of cloud computing has been around since the early 2000's and along with it came the three cloud execution models, Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Although the pay-as-you-go model in acquiring cloud computing resources allows customers to pay for only the resources leased from the cloud providers, often times the users experience a significant gap between the resources acquired and paid for and the actual resource utilization (CPU, memory etc.) [3]. Further, under these execution models, the users are burdened with the operational matters with regard to the provisioned cloud resources for their applications.

Under the IaaS model, the developer has control over the application code, data and provisioning of cloud resources. The PaaS model allows users to write customized code but they no longer possess control over the execution environment. The serverless computing model is a similar paradigm where, the developer has control over the code they deploy, but provides only abstract details of application resource requirements to the platform. The serverless provider is equipped to manage application resource require-

ments automatically, including instance selection, resource scheduling, fault tolerance, monitoring and resource scaling to suit demand surges. In addition, this computing model offers many unique features such as rapid resource auto-scaling, strong isolation, fine-grained billing options and access to a massive service eco-system. While the computing model that uses functions as the deployment unit is called Function-as-a-Service (FaaS), serverless architectures also provide Backend-as-a-Service (BaaS) – client applications, such as database and authentication services [1]. As such, due to its many advantages, serverless computing has currently been explored for use in many application scenarios including big data analytics [4], [5], [6], machine learning [7], [8], Internet of Things [9], [10], and large scale mathematical computations [11], to name a few. Although initially adopted as an execution model for cloud environments, today serverless computing model is being increasingly explored for usage in fog computing environments as well, with a mix of cloud and edge resources [10], [12].

To date, major cloud providers including Amazon, Google, Microsoft and IBM have launched commercial platforms with serverless capabilities, which are being used by many companies in their production environments. (Ex: Netflix uses AWS Lambda serverless platform for video file processing). In addition to the commercial platforms, there are many open-source serverless frameworks such as IBM's OpenWhisk, Fission, Kubeless and OpenFaas [13].

Although serverless computing offers convenience to users in terms of reduced complexity in infrastructure maintenance, easy scalability, and faster set up, many challenges still exist, that may hinder achieving its intended performance objectives. An inherent challenge in the serverless execution model is the lack of control the users have over the resource management for their applications. On the other hand, the cloud vendors carry the heavy burden of overseeing the successful execution of applications of a multitude of users, with minimal prior understanding of their individual requirements. Moreover, maintaining the distinguishing core features of this new model such as its fine-grained auto-scaling properties in such a dynamic multi-tenant environment is quite a challenge. With the novelty of the serverless concept, solutions addressing these challenges are still at an evolving stage.

With the complete offload of the resource allocation and management decisions to

the serverless provider from the developer, adoption of efficient resource management techniques is imperative to achieving both the end user and provider objectives. Thus, the main focus of this thesis is to study techniques for adaptive and autonomic management of resources for applications deployed in serverless platforms, to the satisfaction of all involved parties. This includes investigation into all related matters of resource allocation, scheduling and resource scaling for serverless applications. We first conduct a comprehensive survey reviewing the current status, identified problems and proposed solutions along with their evident shortcomings, with regard to all aspects of serverless resource management. We then proceed to propose efficient techniques for resource provisioning, scheduling and scaling for serverless applications for the optimum use of this computing environment. The superiority of the proposed solutions are proven by evaluating them under both simulation and practical settings.

1.1 Background

In order to better understand the scope and breadth of the research problem addressed in this thesis, this section presents a brief background on the concept of serverless computing, its platform architecture, and the key characteristics of a serverless computing environment. Further, we provide a short introduction to the main application domains for serverless use cases.

1.1.1 Serverless Computing

Serverless computing is an application service model in which, the provider manages the server and handles all the responsibilities related to the execution of the application during its lifetime, with minimum involvement of the user.

Starting with bare-metal servers maintained on-premise with redundancy in order to enable high availability, the way server infrastructure is managed for use by consumers has evolved largely over the years. The biggest transformation in this regard was with the advent of the concept of cloud computing around early 2000. Along with it came the three cloud service models, IaaS, PaaS and SaaS (Figure 1.1). Under the IaaS model

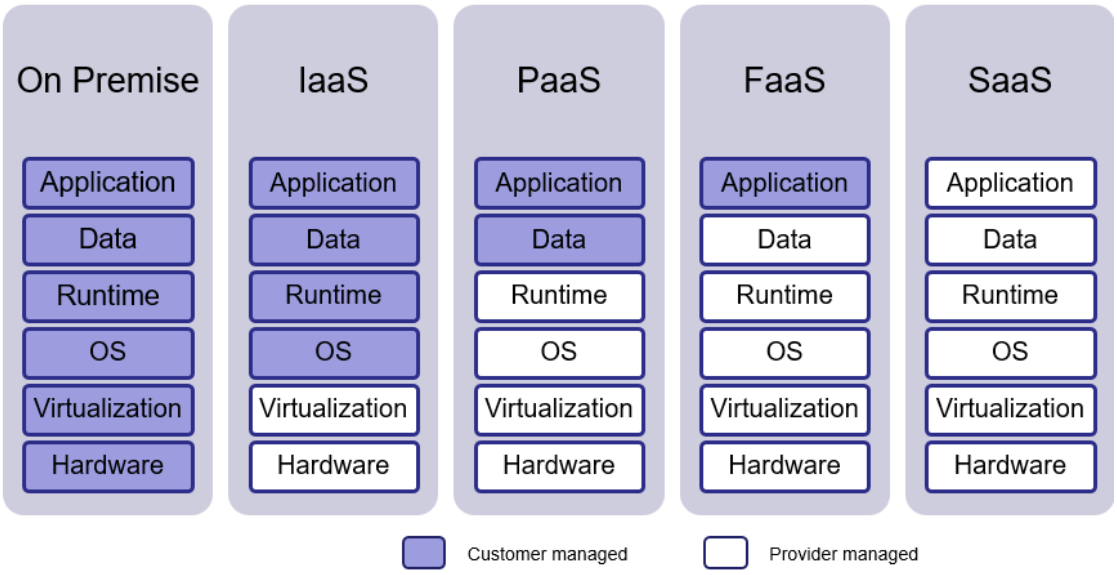


Figure 1.1: Evolution of Cloud Service Models.

(e.g., AWS EC2, Azure Virtual Machines (VMs), Google Cloud Platform), the provider manages hardware resources at their data centers. The developer rents out required virtualized cloud resources from a vendor and thus possesses the responsibilities of resource provisioning, runtime configurations and the management of application code and data. While this pay-as-you-go model in acquiring cloud computing resources allows customers to pay for only the resources leased from the cloud providers, studies show a significant gap between the resources acquired and paid for by cloud users and the actual resource utilization (CPU, memory etc.) [3]. Thus an inherent challenge with the IaaS model is the resultant under utilization of resources in general, when resources are acquired to match the peak demand of a system, and the resultant over utilization and performance degradation when resources are acquired to cater to average demand levels. The PaaS model (e.g., AWS Elastic Beanstalk, Azure app services, Google app engine) provides a platform for users to develop, run and manage customized applications while the execution environment is managed by the cloud provider. The serverless computing model is a similar paradigm where, while the developers have control over the code they deploy, they need to follow certain standards to suit the provider platforms. In addition, this model offers far more granularity with regard to application scaling

and billing schemes which differentiate it from other execution models and create new opportunities. For example, under the serverless model, an application has the option to scale to zero, with no instance of the application consuming any resources when there is no traffic. This is in contrast to a PaaS model, where at least one instance of the application will always be up and running, consuming some amount of resources, irrespective of the traffic levels.

Serverless Computing Architecture

Serverless architecture is an event-driven architecture where users would initially deploy code with the application logic, in the form of stateless functions. A serverless platform defines a set of event sources which could trigger the invocation of these predeployed functions as per the user requirement. A user defines rules, binding deployed functions with corresponding event sources. The supported event sources depend on the platform and these could be HTTP requests from a user interface, a change to a database or an object storage, a notification from an Internet of Things (IoT) device etc. Figure 1.2 illustrates the basic high-level execution flow of the serverless architecture.

Upon the occurrence of an event, a request(s) is sent to the API gateway in order to invoke the relevant function(s) as per the defined set of rules. The scheduler along with the load balancing logic, then determines which worker node is best suited to accommodate the function execution and dispatches the execution request to the relevant worker. A suitable isolated environment with the required resource configurations (e.g., a container) is created on the worker to accommodate the request, if a ready resource is not available. Function execution commences once the required runtime and the associated application code from the application repository, are loaded on to the created environment. Upon completion of execution, the response is sent to the user and the environment created is usually destroyed, releasing the allocated resources. Intermediary data and state management is done via external storage services. The function scaling decisions to meet the requirements of subsequent function execution requests, are taken considering the data from monitoring metrics and the implemented function scaling logic.

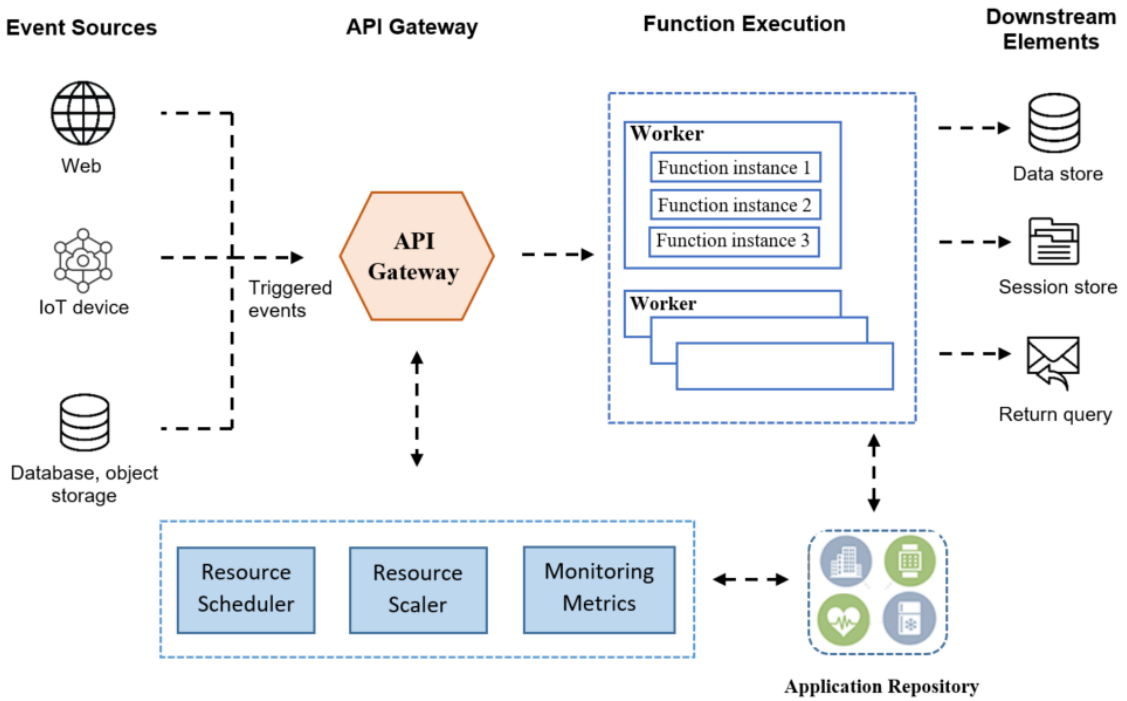


Figure 1.2: Serverless Architecture.

1.1.2 Key Characteristics of Serverless Platforms

As per [1], below are some unique key features of existing serverless platforms.

- Auto-scaling – The platform is expected to be able to scale resources automatically and instantly as per the demand. Serverless platforms are equipped with container technologies which have minimal start-up delays, thus enabling the provision of thousands of instances within a few seconds. Similarly, when there is no traffic to an application, the function instances scale to zero maintaining minimum idle resources.

- Billing – The usage is metered and users are only charged for the resources used when serverless functions are in execution. This means that when a function scales to zero and no node is running the user's code, there is no cost to the user.

- Performance and limits – A variety of limits are set on the run time resource requirements of a serverless code, including the number of concurrent requests,

maximum memory and CPU resources available to a function invocation and an upper bound in execution time before a function instance times out.

- Programming languages – Most platforms support function code written in multiple programming languages which include Javascript, Java, Python, Go and Swift.
- Security and accounting – Serverless platforms are multi-tenant and thus providers need to be considerate of isolating the function execution of different users. Linux kernel features like namespaces and cgroups offered by container technologies provide some level of resource isolation for individual function executions.

1.1.3 Application Domains

In general, bursty and compute-intensive workloads which are stateless and ephemeral could benefit more from a serverless architecture. From a cost perspective, the auto-scaling feature of the platforms proves useful, when traffic arrives in bursts (inconsistent traffic levels), since the system can also scale to zero when there is no traffic [1]. During recent times, serverless computing has increasingly being explored for use in many applications domains such as web and mobile, big data, internet of things, machine learning model training and large scale mathematical computations. The nature of the serverless workload will vary depending on the different requirements of these domains and their inherent characteristics. The focus of the resource management techniques needs to be adapted accordingly. As discussed in the following sections, the core design features of the existing platforms make the serverless model easily adaptable for some domains while for others, an extra effort is needed to reap the full benefits of this novel model.

Web Services

Serverless model is said to have been initially developed for lightweight use cases like serving APIs or small backend services [14]. Even today, studies on serverless use cases reveal that web services is the most common application domain utilizing this new computing paradigm due to ease of adoption [15].

Big Data Analytics

Due to high data volumes and computational requirements, big data processing is traditionally undertaken using clusters of machines with jobs consisting of multiple tasks being executed across a set of machines. Advancement of technology in the field has addressed these performance and scalability needs through distributed computing frameworks specialized for big data analytics [16], [17]. The costs and the knowledge required for configuring, deploying and maintaining these systems is still a challenge. The lower startup times, automated resource management, function level auto-scaling and granular billing features present an interesting opportunity in the serverless model for big data processing [14]. The potential of serverless for big data analytics is being recognized by the major cloud providers as well. AWS provides guidance on reaping benefits of Lambda for streaming data analytics [18]. IBM introduces IBM-PyWren, a data analytics platform using IBM Cloud Functions [19]. A number of research efforts too are seen trying to adopt the serverless model for obtaining favorable results for big data applications [5], [20], [21]. However, many fundamental challenges still exist that impede the performance of distributed data-driven applications on current serverless platforms.

Internet of Things

FaaS model could be beneficial for adoption in IoT applications in edge/fog computing networks owing to a number of factors. Deploying applications as a number of lightweight functions go in line with resource limitations at the edge devices. Statelessness of serverless functions add portability for parts of applications to be moved across the edge/cloud computing network with lesser complications. As a result, today, serverless computing has been exploited in many IoT domains including home automation and other custom-built IoT solutions. AWS offers AWS IoT Greengrass [22] which is included as a service in the serverless ecosystem as well and this allows processing data at the edge. Azure IoT edge [23], which could be used in conjunction with their FaaS service Azure Functions, moves cloud analytics to the edge devices. A number of researches focus on introducing frameworks and scheduling techniques for serverless applications deployed in edge-cloud computing networks [24], [10], [25], [26], [9], [12].

Machine Learning

Machine learning (ML) applications typically consist of three phases: model design, model training and model inference (model serving). VM clusters were traditionally used for the diverse tasks in ML model training. The distinct stages of a ML workflow pipeline consist of varying computational requirements. As such the traditional approaches face several challenges such as the need for developers to provision, configure and manage these resources and the over and under provisioning of these resources [27]. Serverless computing seems to be a promising approach for resource provisioning challenges for ML users, in terms of its simplified deployment opportunities, fine-grained resource provisioning and billing models and the ability to auto-scale both computation and storage resources. Many research works exist in this area that propose serverless frameworks capable of accommodating these applications [8], [7].

Mathematical Computation

Large scale mathematical computations are traditionally deployed on supercomputers or high-performance computing clusters connected by high-speed, low-latency networks [27]. Considering this, serverless seems a poor fit for such applications. However, the ability to unburden non-computer scientists of having to manage infrastructure and scalability to support varying needs of resource parallelism during a computation, highlights benefits of a FaaS model over managing a cluster with a fixed size. Shankar et al. [28] present Numpywren, a serverless system for linear algebra computations. Werner et al. [4] present a prototype for matrix multiplication using FaaS. These experiments show that serverless computing could be a good fit for large scale linear algebra (e.g., matrix multiplication, singular value decomposition) when computation time dominates communication delays. But the high latency of external storage causes limitations for smaller problem sizes.

1.2 Challenges in Serverless Resource Management

Serverless computing is increasingly encouraged to be adopted by developers due to the ease of program deployment, configuration and access to the automated process of resource provisioning, scheduling, monitoring and ensuring fault tolerance. However, the more convenienced the user is in such environments, the more challenges the provider will face in automating the said processes with minimal interruption to performance and with certain QoS guarantees to the user as well. In order to be able to effectively manage the resource allocation and life time management of resources for the applications, the provider needs to base its techniques on the knowledge derived on the state of the system under different scenarios, and the requirements.

The initial task of resource management for serverless applications lies with the allocation of the right amount of resources for an application. Next is deciding on a suitable worker node (virtual machine) to direct the request for execution. Once a host node is selected, the options are to either select an existing environment, or create a new secure and isolated environment (e.g.: container) to commence function execution. The subsequent decisions for scaling resources as required for the current application as well as to accommodate future requests are further operations to be handled.

Below we discuss the challenges associated with fulfilling these chain of responsibilities from within this provider-centric cloud execution model:

- **Blackbox architecture**: One major challenge to overcome under this novel computing model is its promoted lack of transparency of the platform operations to end users. Although this helps to maintain minimal user involvement in the whole process of application execution, it also elevates the level of responsibility for cloud vendors. For example, at the point of application deployment to a serverless platform, the user reveals only minimal application requirements and requests an approximate amount of resources. As such, guaranteeing any level of Quality of Service (QoS) to users requires the provider to determine the appropriate level of resource provisioning which is a massive challenge.

- **Shared platform**: Unlike leased virtual infrastructure under the IaaS model, which are dedicated to the user (if needed), the underlying infrastructure of a server-

less platform are to be shared by millions of users at a time. The co-location of multiple applications with varying resource requirements could lead to resource contentions which in turn could cause performance issues and denial of service. Further, the complicated interference effects of various applications if not isolated properly, could result in poor service reliability. Thus multi-tenancy is a crucial element to be factored in to any resource scheduling or scaling techniques introduced for these platforms.

- **Auto-scaling**: A highly regarded feature of serverless platforms is its ability to attain just-in-time scaling for applications at function level. This means that the platform needs to be capable of adhoc provisioning of resources to suit the demand from multiple users in the scale required. At an extreme end, function resources are to scale to zero when there is no traffic and scale back up when needed. The creation of additional instances real time, adds latencies for request executions (cold-start delay), while maintaining idle resource pools to mitigate its effect leads to resource wastage. The commercial serverless platforms existing today employ very simple techniques for managing the resource scaling problem, which do not lead to optimal outcomes in terms of addressing all these challenges.

- **Fine-grained billing model**: As per the billing model used in commercial serverless environments, the user is charged only for the resource-time actually consumed by incoming traffic for an application with a milli-second granularity. However, in order to accommodate dynamic load levels, the underlying provider infrastructure would need to stay active for longer periods. This is in contrast to rented infrastructure under the IaaS model, which are billed for the whole duration regardless of whether they are actually used or not. Given this situation, optimizing resource utilization is in the best interest of the provider at all times. Nevertheless, packing requests on to a resource for increased utilization at one point sacrifices the QoS guarantees to the user and thus arises a problem of conflicting objectives which needs careful investigation.

- **Execution compatibility**: Serverless computing is being increasingly explored for adaptability in various application domains due its many favorable aspects. As

such, designers of novel applications tend to follow architectures more synonymous with the serverless deployment model, for e.g.: micro-services architecture. However even then, the nature of an application workload could determine the actual user ability to reap the best benefits of the serverless execution model at times. The cold start delays may not be tolerable by a latency critical workload and the high per request cost could bring up the total cost for a regular predictable workload beyond expectations. A conventional cloud execution style such as IaaS, where the readiness of resources is guaranteed, with a much lower per request cost could be the preferred solution in such an instance. On the other hand, bursty traffic with lesser regard for response times, could perform well under the serverless model. As such its worthwhile to draw attention in to filtering the best application scenarios for harvesting the best out of this novel computing paradigm.

1.3 Research Questions and Objectives

The shift in the role of managing cloud resources from the user to the vendor could be identified as the biggest challenge associated with a serverless computing environment. Further, the complications of a shared computing space together with extreme auto-scaling capability expectations have given rise to many unprecedented circumstances with regard to managing the underlying infrastructure. The objective of this thesis is to study and propose adaptive strategies for undertaking all serverless resource management related processes to the satisfaction of both end users and cloud vendors. To meet these objectives, we formulate and solve the following research questions:

- Q1. *How to dynamically manage function resource allocations and request placements to attain high resource efficiency and meet user requirements?* With the lack of user involvement in serverless application deployment and resource allocation, the cloud vendor is exposed to only minimal information regarding application specific behavioral patterns and its requirements. Thus the initial resource allocations done to a function may not always be ideal to fulfill its intended objectives. For example, in a scenario where each function execution has a tightly coupled deadline/response time requirement, the platform needs to ensure maintenance of the QoS at

a sufficient level. This requires monitoring the application performance in runtime and employing dynamic resource provisioning techniques so that host resources are properly distributed among applications. Further, it is important to ensure that the underlying cloud resources are managed efficiently, in order to compensate for the novel pay-per-execute serverless billing model. The initial placement of function instances also need to be in line with these objectives.

- Q2. *How to leverage workload and system characteristics in function scheduling, towards to minimizing resource contention among co-located functions while maintaining high resource efficiency?* Serverless systems are occupied by a variety of user applications at a time. These applications would have varying sensitivities towards different resource elements such as cpu and memory, that affect their performance. As such, this multi-tenant nature of serverless platforms raises a concern regarding performance limitations caused by resource contention from co-located applications. Further, the cost efficiency of cloud resources for the provider is an equally important but conflicting factor of concern in these systems. Thus its important that a platform develops awareness on the workload characteristics and requirements along with the state and behavior of the system in each situation, in automating the process of function scheduling on a shared resource. Balancing the dual objectives of function performance and provider resource efficiency is an added challenge requiring attention in this regard.

- Q3. *How to manage scaling of system resources to minimize cold start latency and achieve satisfactory resource efficiency?* Auto-scaling is a much appreciated feature of the serverless computing paradigm, which promotes resource efficiency. However, a primary shortcoming of enabling auto-scaling is the application latency caused by the cold start of resources, which tends to be significant when applications are short-lived. One way to mitigate this challenge is to proactively scale application resources in time to cater to invocation requests without further delays. Nevertheless, if not done with a comprehensive understanding on the multiple application workloads and the resource cost implications, this could potentially lead to large pools of idling function instances. While the user is not charged for these extra re-

sources, their cost needs to be fully born by the cloud provider which renders the solution unattractive. Thus any auto-scaling solution needs to be able to strike a balance between targeting reduced cold starts while also incorporating techniques for retaining good resource efficiency. Intelligent solutions with capability to capture full system state and derive solutions involving both horizontal and vertical scaling have the potential to meet these expectations.

- Q4. *How to schedule applications in a serverless and serverful hybrid environment in order to achieve highest performance and cost-efficiency?* The embrace of any new computing model requires an analysis on the positive and negative aspects of its performance under varying conditions. As such, the optimal use of the serverless computing paradigm is largely dependent on the platform user's ability to filter the application workload scenarios that could leverage the best out of this deployment model. While certain features such as auto-scaling, are beneficial for most use cases, the resulting cold start delays may offset its advantage for time critical applications. Also, while the pay-per-use billing model favors users with unpredictable workloads, the high per request charge reduces its desirability for an application with regular load patterns. Thus, attention could be drawn to a potential hybrid execution model for a given workload, to explore the full potential of this environment.

1.4 Thesis Contributions

This thesis makes the following contributions to address the research problems mentioned above:

1. Presents a taxonomy covering the major aspects of serverless resource management and their influencing factors, along with a detailed analysis of existing related works in literature using this taxonomy

2. Investigates policies for dynamic management of function resources and initial placement of function requests, aimed at satisfying user specific application requirements and minimizing resource wastage (addresses the Q1).

- A deadline-sensitive placement algorithm for function requests, which effectively enhances VM resource efficiency.
- A fine-grained approach to dynamically manage provisioned resources to functions in the run time.
- Extension to the existing CloudSim [29] simulation environment to support serverless function executions, for testing and evaluating various resource management policies.

3. Proposes a Deep Reinforcement Learning (DRL) based, workload and system aware application scheduling algorithm for resource constrained and multi-tenant serverless computing environments, with flexibility in optimizing application response time and provider cost (addresses the Q2).

   - A RL model of the problem of function instance scheduling in a resource constrained, multi-tenant serverless computing environment.
   - A workload and system aware scheduling framework for serverless functions based on a multi-step Deep Q Learning (DQN) model.
   - Flexibility to establish a trade-off between the two conflicting goals of application response time latency and provider cost efficiency, based on user requirement.
   - A practical testbed environment with the open-source serverless platform Kubeless [30] deployed on a Kubernetes [16] cluster composed of heterogeneous VMs, integrated with Tensorflow (TF)[31] and Keras [32] libraries for training and evaluating the proposed DRL agent.

4. Proposes a framework for response time and provider cost optimized scaling of serverless applications based on a multi-agent DRL model (addresses the Q3).

   - A RL based model of the function auto-scaling problem in a multi-tenant serverless computing environment.
   - A novel multi-agent function scaling algorithm based on the policy gradient algorithm Asynchronous Advantage Actor Critic (A3C), modeled with a

multi-discrete action space required in making horizontal and vertical scaling decisions.

- A configurable reward model to attain a balance in optimizing application performance and resource efficiency.
- An event based simulator environment for serverless function executions with TF agents in the backend, for training and evaluating the proposed DRL model.

5. Puts forward a DRL based approach for deploying and scheduling applications in a serverless and a serverful hybrid environment for minimizing user cost and elevating function performance (addresses the Q4).

    - Problem model for scheduling an application request on a serverless and serverful hybrid cluster environment, based on RL.
    - An actor-critic architecture enhanced with the proximal policy optimization (PPO) technique, and a hierarchical actor network capable of decision making at two levels, namely on the mode/environment of deployment and the node for scheduling within the selected cluster environment.
    - A multi objective reward model for optimizing application performance and user cost for a given workload.
    - DRL agent modeled to capture application workload as well as the serverless and serverful cluster resource details and behavioral patterns in order to gain a comprehensive understanding on its action environment.

1.5 Thesis Organization

The structure of this thesis is shown in Figure 1.3. The remaining part of this thesis is organized as follows:

- Chapter 2 presents a taxonomy and literature review on the aspects of resource management in serverless computing environments. This chapter is derived from:

- **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions", *ACM Computing Surveys*, Volume 54, Issue 11s, Article 222, 36 pages, September 2022.

- Chapter 3 presents a dynamic resource management and a function placement technique to satisfy user request deadlines and achieve high resource efficiency. This chapter is derived from:

  - **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "Deadline-aware dynamic resource management in serverless computing environments", *Proceedings of the 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, Pages: 483-492, Melbourne, Australia, May 10-13, 2021.

- Chapter 4 presents a DRL based workload and system aware function instance scheduling technique for resource constrained, multi-tenant serverless computing environments. This chapter is derived from:

  - **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "Deep reinforcement learning for application scheduling in resource-constrained, multi-tenant serverless computing environments", *Future Generation Computer Systems (FGCS)*, Volume 143, Pages 277-292, June 2023.

- Chapter 5 proposes a multi-agent DRL framework for horizontal and vertical scaling of function resources in a multi-tenant serverless computing environment. This chapter is derived from :

  - **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "A Deep Reinforcement Learning based Algorithm for Time and Cost Optimized Scaling of Serverless Applications", *Future Generation Computer Systems (FGCS)* [Under Review, April 2024].

- Chapter 6 presents a serverless and serverful hybrid scheduling framework for applications, based on a hierarchical actor-critic based DRL architecture. This chapter is derived from:

- **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "Deep Reinforcement Learning for Scheduling Applications in a Serverless and Serverful Hybrid Environment", *IEEE Transactions on Services Computing (TSC)* [Under Review, November 2023].

- Chapter 7 concludes the thesis by summarizing the findings and highlighting the potential for future research work.

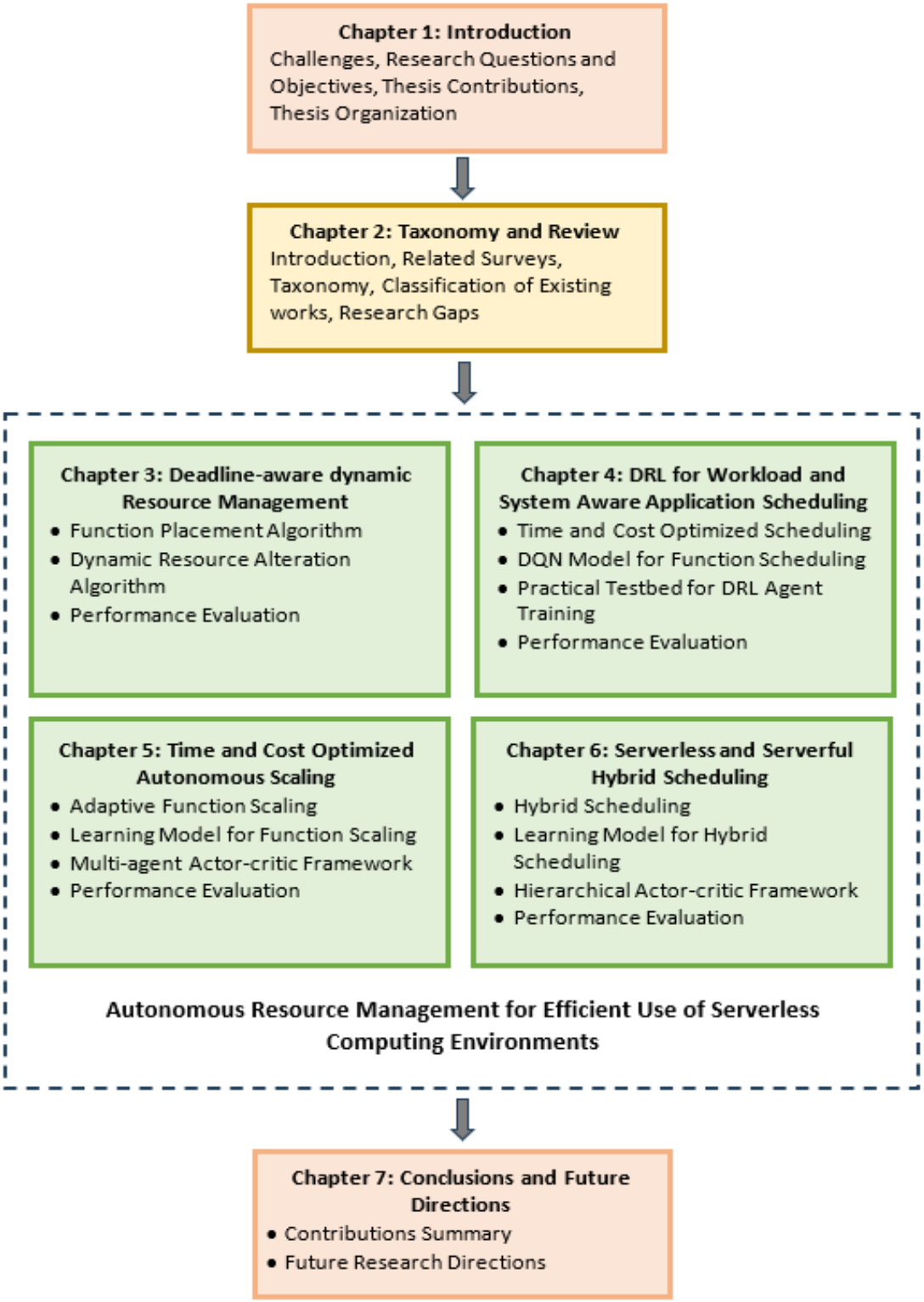


Figure 1.3: Thesis Structure.

Chapter 2

A Taxonomy on Resource Management in Serverless Computing Environments

This chapter investigates the major aspects covering the broader concept of resource management in serverless computing environments and proposes a taxonomy of elements that influence these aspects, encompassing characteristics of system design, workload attributes, and stakeholder expectations. Here we take a holistic view on serverless computing environments deployed across edge, fog, and cloud computing networks. Further, we analyse existing research works discussing aspects of serverless resource management, and the commercially available serverless platforms, using this taxonomy. Finally, we identify the gaps in literature and discuss them in detail for further improving the capabilities of this computing model.

2.1 Introduction

The core differentiator of the serverless model from other computing models is the complete shift of server management responsibilities to the vendor, thus rendering the model 'serverless', from the perspective of the developer. A major, if not the primary aspect of server management, lies in the process of proper management of server resources for the execution of applications. Resource management in a serverless environment refers to the overall aspect of managing the resource requirements of an application workload

This chapter is derived from:

- **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions",*ACM Computing Surveys*, Volume 54, Issue 11s, Article 222, 36 pages, September 2022.

and the available system resources efficiently, with minimal involvement of the user. Due to the autonomic nature of the expected resource management process in these environments, special focus is required on each step of this process for better performance of the applications and the system. We identify three major aspects of resource management, which need to be dealt with in a manner suitable for this new serverless computing model.

- **Workload characterization and performance prediction**: Developers prefer minimal work in using a serverless deployment model. Having to specify a resource configuration and other characteristics when deploying an application could be cumbersome. Hence it is ideal for a serverless platform to be able to infer application as well as workload characteristics using modelling and bench-marking techniques, which could then be used for performance prediction. An effective scheme for developing such an understanding, leads to better resource scheduling and scaling decisions as well, which help in satisfying user QoS requirements.

- **Resource scheduling**: Mapping workloads to suitable host nodes based on their resource requirements, while efficiently utilizing the available resources is an important challenge to both the developer and the cloud providers or system owners. Scheduling also involves determining the order of execution of applications when the resource demand exceeds the available resource capacity. While the developer would expect certain QoS guarantees, it is essential for the provider to manage resources efficiently, which is a primary goal of resource management.

- **Resource scaling**: Under the serverless model, environment creation and resource allocations to applications happen in real time, as and when workloads arrive. This ensures greater flexibility in resource allocations and higher resource efficiency. In order to maintain required application performance while maintaining scaling at such a granular level requires smart and dynamic resource scaling techniques.

Next we identify the challenges associated with resource management strategies which are specifically significant in a serverless environment. We analyze the stated challenges from the perspective of end users and the service providers.

- Cold start delay: Due to the auto-scaling nature of these environments, resource creation needs to happen on the go and setting up new resources for a function execution results in a considerable start up time. Applications often face performance degradations due to this initial delay, which becomes specially significant for functions with very short execution times. On the other hand, maintaining idle resource pools to alleviate this issue often results in wasted resources on the provider side.

- Co-located application interference: Applications deployed on serverless platforms run in multi-tenant environments, inside specially created isolated environments such as containers. When several applications run on the same host node and compete for the same set of resources, it is difficult to fully avoid resource interference effects on the applications. In extreme cases, if resource interference is not handled properly, dominant applications could consume all the resources. This would lead to very poor performance for other user applications. Developing sandboxing techniques which are light-weight enough to avoid large setup times and secure enough to provide the required level of resource isolation is an associated challenge for cloud vendors.

- Resource efficiency: In contrast to a general cloud computing billing model, serverless systems usually charge only for the resources allocated/consumed during the application execution, with a millisecond accuracy. The provider on the other hand may be maintaining the underlying infrastructure for longer periods. This creates the need for special attention to have strategies for high resource efficiency on the host nodes. On the user side, in order to avoid poor application performance, there is a tendency to overbook resources for function executions. Frequent under utilization of these resources could lead to poor value for money for the user and lack of confidence on these services in the long run.

- Diverse workload management: With serverless systems, there is minimal user involvement in the resource management process. Thus, these systems need to develop an understanding on the application and workload characteristics on their own, in order to deliver a favorable outcome. The diverse nature of applications

that are being deployed on serverless platforms makes this a challenging task. The lack of an understanding on the application requirements and characteristics could result in delays in resource setup time, heightened resource interference effects, etc. that lead to customer dissatisfaction.

- Runtime limitations: To alleviate the risk of loosing flexibility of the infrastructure, serverless providers impose various runtime limitations on applications. These include limitations on the maximum allocated CPU, memory, disk, I/O resources as well as maximum allowed execution time for a function instance. Platforms also configure an upper limit on function concurrency levels which limits the number of parallel executions of the same function by a user. Although successful in preserving flexibility of the system, these limitations often deem the serverless platforms unsuitable for long-running applications with highly compute-intensive workloads.

- Lack of QoS guarantees: The blackbox nature of the serverless systems means that the operational aspects visible to end users is minimal. Although this is generally convenient to most users, for certain high-precision, latency-sensitive applications, these environments could be unusable without precise performance guarantees. For the provider, guaranteeing performance to each and every user in a shared environment is a very complex task requiring a lot of attention.

- State management: The fundamental building blocks of a serverless architecture naturally leans more towards short-lived and stateless task executions. Although currently some service providers offer solutions for orchestrating stateful workflows, generally developers are expected to work with external storage services for intermediary state management. Having to frequently retrieve data to and from these storage devices with added delays in the network poses many challenges for the execution of latency critical data-intensive applications. On the other hand, since flexibility in resource management is a distinguishing quality for a serverless environment, addressing the issue of state management to the satisfaction of all users is a highly challenging task for a provider.

The three aspects of resource management identified above need to be addressed

considering the aforementioned associated challenges. Various approaches have been experimented by researchers to overcome these challenges and devise better workload characterization, resource scheduling and scaling techniques. In this chapter, we identify a classification of the factors which influence these decisions, by reviewing the existing related literature. In the classification, we discuss challenges inherent to this new computing model, as well as the general concerns to resource management in the context of serverless computing.

Successfully managing resources in any system is determined by the underlying system design features as well as the understanding on the characteristics of the incoming workloads. As such, our classification comprises of a reference to the key design aspects related to resource management in a serverless system, characteristics of the workloads submitted to these systems, and finally, the primary goals of resource management in these environments. Further, we summarize existing research works related to workload modelling and prediction, resource scheduling and scaling techniques in the serverless domain, using our proposed taxonomy. Moreover, we provide a discussion on the features of existing key commercial and open-source serverless platforms, in line with this classification. We also propose ideas for future work in developing enhanced techniques. Serverless system designers would benefit from our classification by gaining an understanding on the key focus areas of a system design under this computing model and also the existing approaches that have already been evaluated. Researchers studying resource management techniques would be able to refer to existing approaches which could subsequently form the basis for the design of novel and rigorous techniques in the future. This classification would also assist application developers in the design of their serverless applications, choosing the right infrastructure and in budgeting their deployments.

The rest of this chapter is organized as follows. An overview on the existing surveys and studies on resource management in serverless computing environments is provided in section 2.2. Section 2.3 presents the proposed taxonomy of resource management. Section 2.4 summarizes existing works on serverless resource management based on the taxonomy. Section 2.5 provides a discussion on currently available serverless platforms. Finally, section 2.6 discusses the identified gaps in literature, and section 2.7 concludes

this chapter.

2.2 Related Surveys

A literature survey can provide a foundation to examine and understand an evolving research area, in the context of existing work. Preliminary surveys on serverless computing identify opportunities of this new computing model, inherent challenges with the serverless concept and ideas for overcoming the said drawbacks [33], [27] [34], [35], [36], [37]. Hellerstein et al. pinpoint specific challenges of the serverless model with regard to applicability for data-driven distributed computing [38]. Garcia et al. analyse and discuss trade-offs of today's serverless platforms required for effective processing of big data analytics applications [6]. Many studies focus on exploring performance of existing commercial and open-source serverless platforms [39], [40], [13], [41], [42]. Eismann et al. present a characterization of serverless use cases [15]. A comprehensive study on the characterization of applications which have successfully adapted to the serverless domain is done in [43]. None of these works are focused specifically on the overall aspect of resource management under the serverless computing model and the emerging body of related literature.

2.3 The Taxonomy

The top level of our taxonomy is comprised of: the key elements of resource management, deployment environment, workload management and QoS goal. Figure 2.1 illustrates the proposed taxonomy. We discuss each category in detail in the following sections.

2.3.1 Elements of Resource Management

In this section we briefly discuss the techniques explored in literature so far, under the three primary elements of resource management that we have identified. All the factors

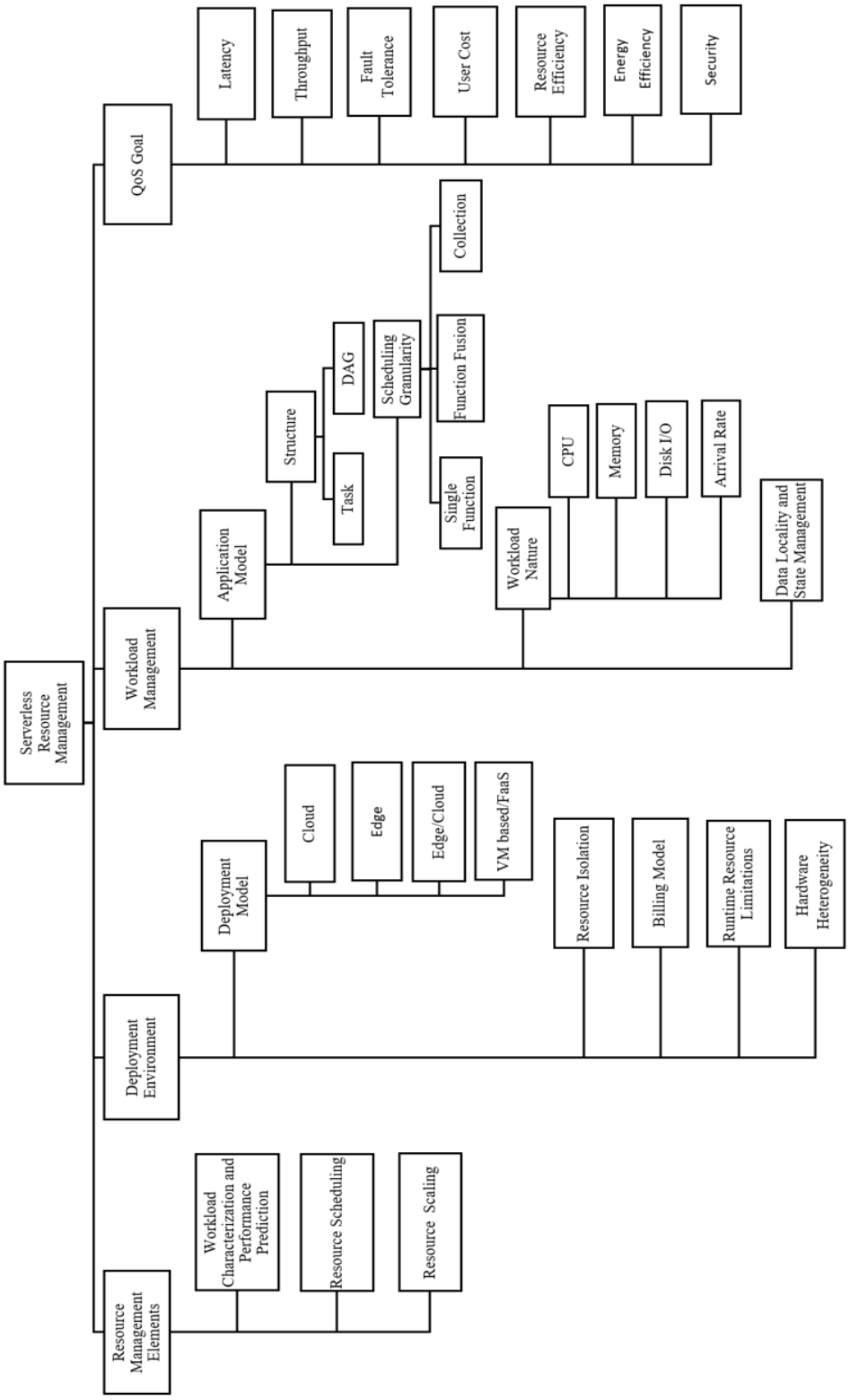


Figure 2.1: The Taxonomy of Resource Management in Serverless Computing Environments.

discussed in detail in the rest of the sections in the taxonomy drive the development of techniques for these key areas.

Workload Characterization and Performance Prediction

Workload characterization refers to using empirical and theoretical approaches to understand and model, the varying resource requirements of different applications, workload arrival patterns and run time behavioral patterns. The knowledge gained in this manner is subsequently used to derive latency and cost models for the system and the applications, which is known as performance prediction. The whole process of developing performance models using the understanding on workload characteristics could be referred to as workload modelling. In the existing literature, mathematical modelling techniques are used in abundance in developing workload characterization techniques and prediction models in serverless environments.

Singvi et al. use a method of exponentially weighted moving average over the measured request arrival rate to get a new estimate of the workload arrival rates [44]. Gunasekaran et al. use a moving window Linear Regression (LR) model to predict the average request rate in their VM/FaaS hybrid model, in order to provision the required VMs in advance [45]. In [46], the authors characterize the entire production workload of Azure Functions. They identify each function in terms of its trigger type, execution time, resource usage and invocation frequency. This information is subsequently used to develop a hybrid histogram and time-series model to reduce cold start invocations. In addition to these research efforts, a number of works have also focused on composing benchmark suites for serverless workloads. Kim et al. present a suite of function workloads which consume CPU, memory, disk I/O and network resources in varying levels [47]. Their collection of applications consists of micro-benchmarks as well as data-oriented realistic applications which would assist in realizing a multitude of application scenarios for research works in the filed. Grambow et al. design an application-centric benchmarking framework for serverless use cases [48]. It is built in a flexible manner to accommodate a variety of applications and also to suit federated cloud and edge scenarios.

A mixture density network based model is used to predict function response time in [49]. Based on the resultant model, a Monte-carlo simulation is used to do cost prediction for serverless workflow executions. This approach takes into consideration, the effects of input parameters to each function, on the execution times. Similarly, application latencies parameterized by the function input data sizes and framework overheads, are modeled using Regularized Ridge Regression in their work by Das et al. [50]. They use training data obtained by running a substantial number of jobs in AWS Lambda, and the OpenFaas platform deployed in the private cloud. Application specific performance models are developed in [26] which are able to predict application latency and costs for various container configurations under an edge/cloud environment. They consider network transfer times, container start up times, function execution times and storage latencies in building their models and use various techniques of regression over the training data sets. For estimating application latency in the cloud, they identify Gradient Boosted Regression Trees to be the most robust. These performance and cost models enable application developers to plan and budget their deployments in the most cost effective manner. Lin et al. propose an analytical model to predict response time and cost for workflow execution in serverless platforms [51]. They consider variations in response time and cost with the allocated memory in building these models. Performance and cost predictions under different resource configurations in serverless settings are explored in [52] and [53]. Mahmoudi et al. also propose an analytical model to help developers to extract performance metrics for their applications before the actual deployment [54]. In particular, their model enables the calculation of the cold start probability, average response time and the required average number of function instances, under stable conditions. Linear Regression is used in [55] for function execution time prediction which is then used in a Least Slack First (LSF) algorithm to select a request for execution from the queue. Zhang et al. use regression techniques to determine the total latency of executing a batch of serverless tasks over the edge and cloud runtime environments, in order to determine the runtime with the least latency [56]. In their work, median sliding window time series modelling technique is used to predict runtime deployment time while Bayesian Ridge regression technique is used for processing time estimation.

Application performance is also largely affected by the characteristics and behavior of the underlying serverless platform. Platform characterization and evaluation is thus an important factor in application performance prediction process. Yu et al. present an open-source benchmark suite for exploring a number of commercial and open-soruce serverless platforms in terms of their function startup latency, performance isolation, communication efficiency, CPU contention etc [57]. Benchmarking frameworks have also been developed accommodating experiments on multiple existing serverless platforms, for ease of comparing their runtime behaviors and performance levels [58], [59]. Mahmoudi et al. present a simulation framework for serverless application developers to develop, test and optimize the performance and cost of their applications [60].

Resource Scheduling

Application scheduling in a serverless environment addresses the challenges of decision making with regard to resource provisioning, resource allocation and scheduling of function invocation requests. A comprehensive resource scheduling scheme would make use of performance prediction tools as identified above, for determining the optimal level of resource allocations and the scheduling policies to meet consumer and provider expectations.

Constraint programming-based approaches (e.g., Integer Linear Programming (ILP), Mixed Integer Linear Programming (MILP) try to minimize or maximize an objective by satisfying the set of constraints for the function execution and the restrictions of available resources. Scheduling serverless functions over the underlying infrastructure of a serverless platform based on user Service Level Agreement (SLA) is a NP-hard problem. Therefore algorithms and approaches for obtaining optimal scheduling decisions may not be feasible for these platforms considering the magnitude of the problem caused by the scale of resources. In contrast, using heuristic approaches is faster and they are scalable to large clusters. These approaches are able to provide acceptable performance and near-optimal solutions. Schedulers in commercial and open-source serverless platforms also seem to be using simple load balancing mechanisms such as round robin and bin-packing approaches [39], [61]. A considerable set of works exist in literature which

study the applicability of various heuristic based approaches for function scheduling in serverless environments [62], [63], [64], [44]. In their work, Pinto et al. explore the applicability of Bayesian Upper Confidence Bound (UCB) algorithms for deciding the optimum location for the execution of functions in an edge/cloud network [25]. Elgamal et al. focus on the problem of finding the most cost effective way of fusing multiple functions and placing them, either on the cloud or the edge [24]. The problem is formulated into a Constrained Shortest Path (CSP) problem and they find the Lagrange Relaxation-based Aggregated Cost (LARAC) algorithm to be the most efficient for solving the problem. Palma et al. present a novel idea of a declarative language which the developers can use to state a scheduling policy and performance goals for their functions [65]. Later, the scheduler uses this policy to choose the best fitting worker for the execution. Although at very initial stages, learning based approaches such as machine learning, and Deep Reinforcement Learning (DRL) methods are also being increasingly explored for resource allocation and scheduling decision making in these environments [66], [67].

Resource Scaling

The ability to scale resources automatically to meet time varying application demand levels is a unique feature of paramount importance, under the serverless deployment model. In order to achieve high resource efficiency, the underlying resources are expected to scale-out as required, when there is a rise of demand for an applications and scale-in with diminishing demand. As such, an application would scale to zero with no resource consumption, at times when there is no demand for application execution. This ensures that the user is only billed for the exact resource consumption during application execution. In turn, this relieves them of incurring costs for over provisioned idling resources during demand drops, which is often a critical issue in traditional serverful deployments. A good approach to scale resources also needs to ensure that the QoS requirements of the user as well as the provider are sufficiently met.

Resource scaling or elasticity of resources in cloud, edge and fog environments, usually refers to two dimensions as horizontal and vertical scaling. Horizontal elasticity of

resources allows to scale-out or scale-in the number of application instances, while vertical elasticity allows to scale-up/down the amount of computing and other resources assigned to each application instance [68].

• *Horizontal scaling*: The existing commercial serverless platforms host containerized function instances in readily available VMs. Hence the horizontal scaling of serverless applications is affected by the availability of VMs with required resources. Experimentation done on AWS Lambda show that there is no significant change to cold start delay, when a new function instance is launched on a new VM previously not used for executions, and an existing VM [39]. This indicates that the service providers usually maintain a pool of ready VMs for function executions and thus VM start up time is not usually a determining factor for function scheduling approaches. But recent interest in using the serverless model for compute intensive workloads such as predictive analysis applications using pre-trained deep learning models, present situations where dynamic scaling of VM resources is required to manage the dynamic workloads with varying resource requirements [69].

Serverless platforms often utilize containers as the sandboxing mechanism for the isolation of applications from each other. Each function instance is usually run on a separate container with the required resource configurations. Thus, prior to application execution, the required resource setup generally includes launching a new container, setting up the runtime environment and application specific initialization. The time taken for all these steps is collectively knows as the cold start latency. Generally the cold start latency of containers, which are lightweight resource units, is in the order of milliseconds. But studies have shown that in serverless executions, this delay is largely dependent on each function's runtime dependencies and at times could grow to be even a few seconds [39], [63]. In order to attain the intended benefits of serverless auto-scaling abilities, it is a necessity that the function cold start latencies are managed appropriately.

To alleviate the frequency of cold starts, serverless platforms often try to either reuse warm containers or create pre-warmed containers. The reuse of warm containers avoids any setup and initialization delays while pre-warmed containers avoid container launching delays. AWS, Azure and Google Cloud Functions maintain idle function instances for a particular time period, before they are recycled, in order to increase chances of con-

tainer reuse [39]. Apache OpenWhisk maintains pre-warmed containers and also uses load balancing strategies to direct similar function executions to the same set of workers as much as possible, thus enhancing chances of container reuse [61]. In existing research work, many models are presented to predict the arrival rates of incoming function requests and the demand for particular function executions and thereby proactively setup and maintain a pool of warm containers across VMs [70], [44], [64], [71], [72], [73], [74]. Subsequently, load balancing algorithms are devised to benefit from these existing resource pools. In contrast to these works, Mohan et al. identify the processes of network creation and connection to be the major bottlenecks in container startup and propose a method for maintaining a pool of pre-created network elements which could be attached to a new function container whenever needed [75]. This is done by using the concept of pause containers, which are network-ready empty containers which could be attached to other containers. Similarly in [76], Silva et al. try to reduce the container startup time by implementing a checkpoint mechanism for restoring snapshots from recently created function processes. Stein et al. propose a non-cooperative resource allocation heuristic for serverless environments which aims to predict the number of function instances required to be kept in order to maintain request waiting time below a threshold level [70]. Somma et al. use a Q-Learning based approach to determine the ideal number of function containers to scale-up/down at each instance, to maintain high resource efficiency and low application latency [77]. Gias et al. compare the idle time of a function instance in a FaaS platform to the Time To Live (TTL) value of a TTL caching system [78]. They present a model to decide on the most suitable idle time that each function instance is to be maintained in the system so as to meet the function response time requirements of the user. At times the reuse of warm containers may even cause additional latencies for example, when already fetched data in a function instance is out-of-date and required database connections have already reverted by the time a new request arrives at the container. Mechanisms to freshen up the warm instances prior to use is of importance in such instances [79].

The concurrency level, which determines the maximum number of requests that the system could process in parallel for each different function, is also an important factor in function scaling. Existing commercial platforms have set fixed limits on concurrency

levels for a particular function [39]. Schuler et al. present a Q-learning based Reinforcement Learning (RL) approach to determine the best level of request concurrency, to achieve better performance in terms of system throughput and mean function latency [66].

• *Vertical scaling*: The core concept behind the serverless paradigm is to shift the complexities of application resource management from the developer to the service provider. Thus, the provider has the responsibility to autonomously manage application resource allocations as required, in contrast to allocating resources as per a detailed resource request under a serverful model [27]. For instance, AWS Lambda requires the user to only provide the amount of memory to be allocated per function instance and CPU power is stated to be allocated linearly in proportion to the requested memory [18]. CPU is known to be a source of contention in serverless environments [63], affecting application latencies. Hence, resource allocations to applications need to be monitored during runtime, to avoid potential SLA violations to the user. Further, the providers need to carefully manage the CPU and memory resources allocated to applications in order to achieve resource efficiency and avoid over/under provisioning of resources.

Adjusting CPU allocations to function instances in the runtime is a new research area in serverles computing. Suresh et al. study the impact of dynamically adjusting CPU-shares to containerized function instances in the runtime [63]. CPU-shares indicates the relative weight given to a container in terms of the proportion of CPU time it is given access to when CPU resources are limited [80]. As per their model, containers are launched on VMs when spare memory capacity is available to accommodate the container. Thus multiple containers co-located on a VM would share the same processor core and thus the CPU time available to a container varies over time. This could hinder satisfying user latency requirements to certain applications. Experiments show that fine-tuned regulation of the CPU-shares allocations to containers dynamically could result in better QoS satisfaction. In a previous work [62], we proposed a technique for dynamic CPU resource management to containers running serverless functions by applying the concept of cpu-quota and cpu-period enabled in Linux Kernel's Completely Fair Scheduler (CFS) [81]. The cpu-quota value sets the number of microseconds per cpu-period that the function instance's access to CPU resources is limited to, before it is throttled

[80]. Thus this acts as an effective ceiling and a hard limit for CPU resources allocated to a function instance. This technique helps in allocating a guaranteed CPU time for a function execution and also fine-grained management of underlying resources. Yu et al. propose a DRL based technique to dynamically harvest CPU and memory resources from idle functions and allocate them to under-provisioned functions [82]. As requests arrive, the agent evaluates the cluster state and each function's past cpu and memory consumption, request arrival rates etc., and decide on a new resource configuration for the existing containers.

2.3.2 Deployment Environment

The design of a serverless platform needs to address the inherent unique features offered by this computing model and also the associated key challenges. We characterize these factors in terms of the platform deployment model, resource isolation techniques, the incorporated pricing models, the nature of imposed runtime resource limitations and hardware heterogeneity. Identifying these influencing factors help in developing effective algorithms, system models and other techniques for the efficient management of resources in serverless environments.

Deployment Model

Serverless platforms could be deployed on public cloud, private cloud or on edge resources. The serverless model could also be used in conjunction with a serverful model, i.e.: a VM based deployment. The deployment model is concerned with the nature of infrastructure combination from each resource environment, the associated resource capabilities, pricing models and subsequent resource management decisions suited for each environment for better QoS. This section is focused on cloud and edge only deployments as well as hybrid deployment models with resources used interchangeably from different environments for better performance and efficiency.

• *Cloud*: This is the most common as well as the basic deployment model targeted by serverless computing when it first emerged. A cloud environment could be an on premise, user managed, private environment or a vendor managed public environment

with seemingly unlimited resources. A private cloud environment gives better flexibility to the user to device scheduling techniques to achieve desired performance guarantees. In addition, privacy is less of a concern. While these environments usually enable reaping cost benefits in the long run for regular workload scenarios, the downside is the inability to meet sudden demand surges if additional hardware is not maintained. With a private/public hybrid cloud model, whenever the load goes beyond the capacity of local infrastructure, some of the functions could be dispatched and processed in the public cloud at a cost. In a hybrid model, one could use a commercial serverless provider's service together with an open source serverless framework deployed on a private network. The serverless scheduling challenge then is to decide when and which functions are to be off-loaded to the public cloud for better QoS guarantees and cost benefits. These decisions need to consider the data transfer times across networks, resource setup times, along with the load levels and required QoS guarantees. Das et al. [50] propose a hybrid cloud scheduling framework for multi-function serverless applications with AWS Lambda as the public cloud and OpenFaaS [83] deployed in the private cloud. The portable nature of the serverless workloads also makes it a viable candidate for adopting multi-cloud architectures to harness the benefits of different providers [84]. Liu et al. present a JointCloud [85] platform for serverless computing, which is capable of coordinating resources of multiple cloud vendors for request executions [86]. Service from high performant clouds is requested for latency-sensitive applications while cheaper clouds are used for jobs with lesser requirements. A primitive idea for a federated ecosystem is proposed in [87] for a decentralized serverless architecture for balancing load traffic at the edge. Further studies on the design of multi-cloud architectures in a serverless setting are seen in [88], [89].

• *Edge*: Edge computing leverages computing power and storage facilities close to the consumer in order to reduce application latencies and bandwidth usage. As discussed previously under serverless use cases, serverless computing seems to be a good fit for the deployment of applications across resource constrained edge computing networks due to the ephemeral nature and portability of serverless functions. Gand et al. propose an architecture for clustered container applications for the edge, based on serverless computing [90]. Baresi et al. propose a serverless edge computing architecture

for mobile applications with low latency and high throughput requirements [91]. They use mobile devices and mobile edge computing servers as their main computational elements and evaluate the feasibility of their approach via a mobile augmented reality use case. Containers which are often used as the isolation mechanism for serverless functions, are at times not viable for edge environments with limited resource capabilities due to their inherent setup and runtime overheads. Hall et al. propose a novel run time and an isolation mechanism for serverless functions called WebAssembly which reduces resource startup times and the resource provisioning requirements [92].

• *Edge/Cloud*: A hybrid deployment model leveraging resource capabilities of both the edge and the cloud is often times the most viable solution under many practical scenarios. But many challenges exist, for realizing the true benefit of such a setting. For an application deployed in a serverless edge/cloud infrastructure, which functions are to be deployed on the fog or cloud resources and which node containing the deployed function is best suited for accommodating a new request is a scheduling decision. The resource allocation and scheduling decisions for serverless applications in an edge/-cloud computing network will depend on how compute intensive the function is, the size of data involved and data transfer costs over different network paths, the cost and the setup time of resources at each location, and the expected QoS levels [24], [25], [10].

• *VM based/ FaaS*: Traditionally VM based deployments were used to dynamically scale resources as per the demand. However challenges exist such as higher costs due to over-provisioning or SLA violations due to under-provisioning. Under the serverless model, functions are auto-scaled within containers which have a low startup latency. Further, user is billed per function invocation at a very granular level, thus avoiding resource over-provisioning costs. However, it is observed that deploying an entire application as serverless functions would not be cost effective at times. Lambda functions are expensive (cost increases linearly) when there is a fixed load (constant request arrival rate for a function) and a higher average request rate. Thus it is seen that functions are more cost efficient with demand variations and lower average request rates while VM based deployments serve better with higher arrival rates. Exploring hybrid models of VM and FaaS deployments is an interesting dimension [45]. Li et al. present a serverless architecture for dynamically switching the deployment of a microservice between FaaS

and IaaS infrastructure [93]. The decision is taken in a contention-aware manner, based on the predicted performance of the service on a FaaS platform.

Resource Isolation

Serverless systems are multi-tenant systems and thus techniques for isolating applications from each other is important for reasons of both application performance and security. Container technologies are used as the sandboxing mechanism for function executions by many commercial serverless platforms. Studies show that co-located function instances show effects of resource contention with regard to CPU and network bandwidth, raising concerns on the effectiveness of the isolation mechanisms [39]. By design, components isolated by operating system (OS) level virtualization techniques (e.g., containers), share hardware and the host's OS kernel and thus are open to security vulnerabilities. AWS uses MicroVMs which are hardware-isolated lightweight virtual machines with their own mini-kernel [94]. They offer security and workload isolation from hardware-virtualization as with VMs, and the resource efficiency and smaller startup times of containers. Existing work in this area propose customized sandboxing techniques for function execution. Arguments are laid suggesting that stronger isolation mechanisms such as containers are needed only among different applications and weaker mechanisms such as having separate processes is sufficient for isolation among functions of the same application. Akkus et al. propose a new sandboxing method where different applications run inside separate containers, but different functions of the same application run inside the same container in parallel processes [95]. Concurrent calls to the same function are handled inside the same container by spawning new processes by incrementing the memory allocation to the container. Since the memory footprint of a process is smaller than that of a container, this results in higher resource efficiency. Further, forking a new process inside a container only incurs a short startup latency. Oakes et al. present a container system optimized for serverless workloads [96]. Bottlenecks of container cold starts are identified to be caused by limitations in Linux isolation primitives and loading dependent packages.

Billing Model

Billing model refers to how costs are calculated for function executions in serverless environments. Since serverless offers a very fine-grained billing model, it is important to understand the fine details of billing criteria for different resources. This helps to make cost efficient resource allocation and scheduling decisions for both the provider and end user.

Serverless platforms offer a pay-as-you-execute billing model usually based on the CPU, memory and storage resource costs associated with each function execution. Additional costs may also be involved based on the nature of the application scenario and additional services consumed (e.g., state transitions costs of AWS Step Functions, Amazon Simple Queue Service (SQS) costs etc.). Although the basic billing model on the major commercial serverless platforms is the same, differences exist in finer details. Above its monthly free tier, AWS Lambda charges function executions per GB-s (rounded up to the nearest 1ms) of memory allocated and the number of execution requests [18]. The billing scheme of Azure functions is similar to AWS except that billing is done per GB-s of average memory consumed during an execution instead of the memory allocated. Google functions charge users for both the GB-s of memory provisioned and GHz-s of CPU provisioned.

Decisions on which functions of an application are suitable to be fused (combined) and which are best placed in the cloud or on the edge, are affected by the billing model [24]. Also, under a public/private hybrid cloud serverless model, function placement decisions in the private and the public cloud are based on the serverless billing model in the public cloud [50]. Gunasekaran et al. present a framework to use serverless functions and VM based executions interchangeably for better cost efficiency [45]. Instead of the static pricing schemes of the existing platforms, schedulers could also utilize dynamic pricing schemes which enable users requiring higher QoS levels (e.g., faster response time for applications with high delay sensitivity) to pay and acquire better services [97].

Runtime Resource Limitations

Serverless platforms impose various limitations on the allowed resource configurations for applications deployed on them. This is primarily aimed at improving flexibility in managing the available resources among multiple users without locking in a set of resources with a single user or application. These restrictions also prevent serverless systems in the public cloud from being subject to denial-of-service (DoS) attacks by malicious function requests flooding the resources attempting to overload the system and preventing legitimate requests from being fulfilled. On commercial serverless platforms, these limitations are primarily in the allowed memory, local disk space and cpu resource configurations, maximum allowed time for a function execution and the maximum number of parallel executions for a function without compromising on latency. AWS Lambda allows users to select the amount of memory available to a function during execution, and allocates CPU power linearly in proportion to the configured memory. Concurrency is also allowed to be configured for each function with different options available which could be explored based on the workload [18]. AWS Lambda provides a non-persistent local disk of a standard size for all the functions [39] and imposes a maximum function timeout duration. Azure functions on the other hand introduces different hosting plans for function apps such as the consumption, premium and dedicated plan, based on which the resource and time out limitations will be determined [98]. Google Cloud Functions too, limits the maximum allowable memory for a function and imposes a cap on function duration [99]. Imposing limitations as discussed here, have detrimental effects for some application domains such as long running, compute intensive, data-driven applications [38]. For example, for a data analytics application, an automatic timeout would mean loosing any data cached on the running instances, and the addition of a significant latency for retrieval of the same from a slow external storage. Limitations on the concurrent function executions could also affect adaptability in a data processing scenario requiring a massive number of parallel computations.

Hardware Heterogeneity

With the continuous expansion of the use of the serverless paradigm in different domains, there is also an escalation of varying demands in order to fulfill SLAs for the applications. Computing platforms with heterogeneity in the hardware layer are being increasingly sought after, especially for workloads requiring High Performance Computing (HPC) [100], [101]. Specifically, compute-intensive workloads such as deep learning, complex data analytics, blockchain, genetics, require support from accelerated hardware components, which even the latest generation of CPUs are unable to provide. Offerings from IaaS providers today have evolved to address this emerging need, by providing instances with access to accelerators such as Graphical Processing Units (GPUs), Tensor Processing Unit (TPUs), Field Programmable Gate Arrays (FPGAs) and even access to quantum computers. Inferentia and Titanium chips from AWS [102], [103] to train ML models, Amazon Elastic Inference service [104] to attach GPUs to VM instances, GPU-enabled nodes on Azure Kubernetes Service (AKS) [105], FPGA support in Azure [106], Google's cloud and edge TPU [107], [108] are a few examples. In addition, quantum-based processing elements have rapidly emerged in recent times and cloud service providers have started to offer such quantum processing capabilities as part of their cloud services as well [109]. However, none of the serverless platforms from these providers offer access to hardware accelerators at the moment [110]. As a result, research focus has been increasingly directed towards expanding the FaaS service offerings by adding access to hardware accelerators. Naranjo et al. introduce a GPU enabled serverless framework which links virtual GPUs with the OpenFaas serverless framework via the rCUDA [111] remote GPU virtualization service [112]. Ringlein et al. propose a system architecture involving disaggregated FPGAs within a FaaS offering [113]. A distributed FPGA sharing system which realizes multi-tenancy for FPGAs in a cloud serverless environment is presented in the works of Bacis et al. [114]. For certain application scenarios, serving all the queries using expensive hardware accelerators may not be economically viable (e.g., ML inference queries). For such scenarios, hybrid approaches to opportunistically serve the incoming requests using a mix of GPU-enabled instances and traditional CPU-only instances, could be explored [115], [116]. Research efforts are also directed towards highlighting the adaptability of quantum computing in

the design of serverless systems [117], [118].

2.3.3 Workload Management

Since the serverless provider is responsible for the autonomic management of resources for applications, it is imperative that the platform develops an understanding of the nature, requirements and behavior of the incoming workloads. Three aspects in which awareness of the incoming workload is important for making efficient resource management decisions, are discussed below.

Application Model

Application model is the nature of the scheduling unit of an application that is deployed on a serverless platform. An application could have a certain structure and also QoS requirements that are independently specified for each task or for the application unit as a whole.

• *Structure*: An application could be defined by a single task or a set of tasks compiled in the form of a Directed Acyclic Graph (DAG). A task would be denoted by a single function. Thus in a DAG, each node is a function representing a fine-grained task and each edge represents a dependency among two functions [119], [44], [120], [50]. A developer would specify the QoS requirements for a DAG based application either as a whole [44] or for each individual task. A serverless application could also be monolithic, composed of a single function, representing a single task [121], [122], [61], [67], [62], [123].

• *Scheduling Granularity*: Scheduling granularity refers to the way in which scheduling algorithms handle the execution of an application submitted by a user. When an application takes the form of a DAG, it could either be scheduled as a single unit or each individual function could be scheduled separately. In addition, the scheduler could decide to queue requests and schedule them in batches [124]. The scheduling granularity could be specified as a requirement of the developer or decided by a serverless platform, aiming for better resource efficiency.

AWS schedules each function independently, irrespective of whether it is part of a

workflow or a single function application. Accordingly, a user request calling the first function in a workflow application and a request calling one function from another during a workflow execution, are treated in a similar manner. Both requests go through the same scheduling policies [18]. Azure Cloud Functions uses the concept of "function app", as the unit of scheduling and management of a serverless application. A function app is comprised of one or more functions with dependencies [98]. Studies suggest that orchestrating functions of the same application on an individual basis at a global system level could incur extra latency. Akkus et. al suggest a serverless architecture utilizing a hierarchical message queuing mechanism with a local message bus on each host which handles local interactions among functions of the same application and their orchestration [95]. Developers could also find the optimal way to fuse or combine a number of functions in a DAG for scheduling purposes, so as to reduce execution cost and latency associated with state transitions and network delays. This would be applicable to serverless deployments in the edge/fog environments as well due to enhanced delays over the network caused by data transfers across functions and associated costs [24]. When requests are less latency sensitive, the scheduler could decide to queue and schedule requests in batches or as a collection, for better resource efficiency.

Workload Nature

Serverless platforms are multi-tenant infrastructures and thus applications of multiple users compete for resources in a common shared environment. This could cause a lot of contention on the platform and lead to poor application performance if the resources are not properly managed as per the requirements of different applications.

By nature any generic application could be either CPU, memory or I/O intensive. In some commercial serverless platforms, function placement on a VM is treated as a bin packing problem to maximize memory utilization [39]. While this helps maximize resource utilization, it could also lead to CPU contentions if strict resource isolation strategies among applications are not adopted. Having an understanding on the rate at which requests arrive for applications too is an important factor.

The nature of the workload could be adopted in resource management decisions ei-

ther when scheduling functions on VMs initially or in the subsequent management of limited resources. Mahmoudi et al. use a machine learning based approach to develop a predictive performance model which tries to predict the normalized performance of any workload when assigned to a specific VM[67]. Each time a new function is being deployed on a platform, a profiling step identifies the memory and CPU utilization levels of the function as well as the dependence on I/O features of the platform. Then predictive models are used to identify the VM that gives the best system performance for the function at that time slot. A good understanding of the nature of the applications is also important when the execution happens in an edge/cloud environment. Compute intensive functions are better served at the cloud which has seemingly unlimited resources even with a higher latency due to network delays. Lightweight functions may be better served at the edge with a lower response time. The placement decision of functions on the edge/cloud devices requires a thorough understanding of the application behavior in each of the environments [24], [25].

If a serverless platform does not impose hard upper limits on the level of resources that a particular function instance could utilize, the co-location of many such functions having the same resource needs, could lead to deterioration of performance over time. Under such conditions, based on the state of the system and the resource needs of each application, dynamic management of applied resource limits to containers is an approach worth exploring more [62], [63]. Further, we could use Linux Kernerl's cgroups to control and isolate resource usage of a collection of processes based on the requirements [81], [120]. These techniques could be adopted effectively with a better understanding on the needs of different workloads. Further, in addition to the resource requirements, modelling arrival rate of requests for different functions help in taking proactive resource scaling decisions [44].

Data Locality and State Management

Data locality generally refers to the movement of computation to the nodes which contain the data required for the task execution. For data intensive applications this could improve the makespan drastically due to reduced network delays. Serverless platforms

are known to decouple data management from function execution. Thus serverless is said to be rather a data-shipping architecture (ships data to code) instead of shipping code to data [38]. This is a defining property of the serverless paradigm since the dis-aggregation of these two aspects allows serverless functions to scale in an independent manner [6]. For example, ensuring locality among serverless functions may mean executing functions which share data, in the same node or VM instance. While this would improve performance with fast shared memory, this could also reduce the flexibility of the provider to schedule functions and scale capacity [6]. Therefore scheduling and resource management techniques under the serverless model traditionally have not incorporated data-locality awareness in making their decisions. However, a significant repercussion of this dis-aggregated model is the challenge of managing the intermediary state for stateful applications. Generally applications are required to be executed as stateless functions in commercial serverless platforms, and they share state through external storage services (e.g., Amazon S3) [27], [38]. As such, for latency and bandwidth sensitive data-intensive applications such as machine learning, delays in network and storage layers make this a less than an ideal environment. Today, this realization has motivated research into the development of serverless frameworks that support stateful executions. Azure Durable Functions allow to write and orchestrate stateful workflows [125]. AWS Step Functions [18] and IBM composer [126] support the creation and coordination of state transitions for complex workflows. Shillaker et al. introduce a new abstraction for function isolation, which allows memory to be shared between functions in the same address space [127]. They propose a two-tier architecture, where the local tier allows sharing of state among functions in the same host and the global tier supports state sharing across machines. A data transfer and state management approach is presented in [128] for a stateful function chain in an edge network. Jia et al. present a new serverless runtime which exposes an API to a shared log service which enables serverless functions to manage their state efficiently [129]. This mechanism of shared logs is able to achieve state management with high durability, fault tolerance and consistency.

In addition to the above, many studies focus on achieving data locality via load balancing methods to direct new function requests to workers with pro-actively spawned containers or containers that could be re-used [44], [64]. Research works have also fo-

cused on attaining data locality among functions in terms of the runtime package dependencies of functions, by routing functions requiring similar packages to the same node for execution [122]. Lee et al. propose a greedy load balancing algorithm which tries to maximize locality and improve the cache-hit ratio while also minimizing load imbalance among nodes [130]. The new sandboxing method proposed by Akkus et al. [95] where functions of the same application share the same container, is able to improve latency by increased data locality, since the libraries shared by all the functions are needed to be loaded from memory only once.

2.3.4 QoS Goal

The resource allocation, scheduling and scaling techniques are aimed at satisfying certain requirements upon the execution of applications. These could be constraints that applications need to satisfy or an optimization goal that determines the performance of the resource management techniques.

Latency

Latency refers to the delay between a user request submission and the serverless provider's response. The request queuing time, resource setup time and the function execution time collectively result in the response time for a serverless application. The ability to maintain low latency for function executions is a key concern in serverless environments, specially since a majority of individual functions have execution times less than a second, or of a few seconds [44]. One main cause for high application latency in serverless environments is the cold start delay in resource setup, which tends to become significant compared to application execution times [62]. Scheduling algorithms along with different container pool management techniques, customized sandboxing methods, and low-overhead runtimes to reduce resource setup times are explored in literature to reduce cold start delay [131], [122], [44], [64], [78], [95], [96]. Latency caused by CPU contention is also studied and dynamic resource control methods are proposed for serverless applications [62], [63], [120], [121].

Throughput

Throughput refers to the number of function requests processed by a serverless system in unit time. This is a good metric to demonstrate efficiency of the overall system and the ability to support high request arrival rates [67]. Throughput and mean latency of a system usually show a negative correlation. Thus techniques to reduce latency also improve throughput of a system. The allowed concurrency level for function instances in serverless platforms is also a determining factor for system throughput [66].

Fault Tolerance

Fault tolerance in cloud computing refers to the ability of a system to function uninterrupted, at the sight of failure in one or more of its components. Serverless platforms generally achieve fault tolerance by implementing retry based approaches. For example, a platform would automatically resend the request in case of failure of a function execution. Sreekanti et al. show that under certain circumstances, a retry based approach to fault tolerance may always not give accurate results, specially where state management is involved via external storage services [132]. They emphasize that the accuracy may be affected in case a parallel execution of a function, views results of a partially executed failed attempt of the same execution. A low-overhead fault tolerance scheme is proposed to guarantee read atomic isolation, which prevents parallel executions of functions from viewing partial outputs from any failed attempts. Zhang et al. introduce a runtime system and a library, to help developers to write stateful and fault tolerant serverless workflows [133]. Their approach encompasses a log-based request re-execution approach to achieve fault tolerance. The developed framework could be adapted to work with existing cloud providers in a federated environment.

User Cost

Function execution cost to the user is as per the billing model of serverless platforms as described in the previous section. Since billing is mostly correlated with the function execution times, efforts to reduce execution latency result in optimized cost to the

user. But for compute intensive applications, response time and cost could be inversely related since the allocated CPU time would affect the response time and thus the execution cost. Experiments are done to observe the response time of functions for different resource allocations and associated costs. Such models could help a user decide on a suitable resource allocation scheme when there is a budget constraint, and also the cost to achieve a certain performance level [51], [97]. The scheduling granularity in terms of function fusion in serverless workflows could have an impact on the overall costs as well due to associated state transition costs and effects on function response times [24].

Resource Efficiency

Resource efficiency refers to maintaining a high utilization level for the underlying active resources of the provider at all times. The fine-grained serverless billing model implies that the user is charged only for the resource-time actually consumed by the application during its execution. Regardless of this, the provider has to maintain the overall infrastructure and as such, consolidating as many serverless applications as possible into a single host is in provider's best interest in order to yield better profits. On the other hand, packing too many requests on a single resource subsequently leads to poor performance. This reflects the typical conflicting objectives of the providers and consumers: minimization of cost and maximization of performance [120]. Thus finding an optimal resource consolidation strategy to the satisfaction of both parties is important. HoseinyFarahabady et al. propose a QoS aware resource allocation controller for serverless environments which tries to minimize QoS violations to users while maintaining a healthy CPU utilization level of 60%-80% in each host in order to reach an ideal balance between system performance and energy consumption [121]. Somma et al propose a function container auto-scaling technique to optimize throughput and resource efficiency [77]. Resource efficiency is also important in serverless platforms deployed on edge resources as well due to limited capacities at the edge.

Energy Efficiency

Recently, attention has also been given to the aspect of energy efficiency of running serverless systems, which is also connected to resource efficiency. Early surveys identify serverless computing to be promoting green computing due to the on-demand creation and release of resources used for function execution. Moreover, the model of billing per execution time incentivizes programmers to improve resource usage and execution time of their code [33]. On the other hand, breaking down an application into a set of functions and the practice of setting up resources on-demand is deemed to cause additional latency and an execution overhead, which affect function performance. Kansi et al. define a measure for energy efficiency, based on the mean time between two calls to the same function, mean time to setup resources and the mean time to execute the function [134]. They show that the resource saving in serverless systems is generally highest when function calls are irregular and have large time gaps in between, compared to the resource setup and function execution times. Deriving from this idea, Poth et al. present a further developed model, capturing the overheads of the components which manage the serverless system during function invocation and execution, which is aimed at helping design decisions of serverless applications and systems [135]. Gunasekaran et al. present a resource management framework to efficiently manage function chains on a serverless platform by improving container utilization and cluster wide energy consumption [55].

Security

In addition to the security challenges that are common to any computing environment, serverless systems are susceptible to certain threats that are unique to these environments. Shafiei et al. identify application-level and function-level authentication to be one such specific challenge [**?**]. Application-level authentication refers to a mechanism which determines which users are allowed to access a certain application, while function-level authentication refers to the allowed invocations of one function from another. They also highlight the possible chance of replay attacks, where an intruder would capture a function execution request and execute it repeatedly, constraining the

system resources and blocking legitimate users from accessing the service. In general, runtime limitations are in place, in terms of limited execution times and maximum allowed CPU and memory allocations for a function, in order to minimize the effect of such attacks. Function isolation mechanism too plays an important role in enabling data privacy among tenants on the same host. Container namespaces are commonly used by current serverless systems for providing isolation among functions [136].

2.4 Classification of Resource Management Techniques Using Taxonomy

In Table 2.1, we review existing key works on serverless resource management, that identify most with the proposed taxonomy. In essence, we present here, the works which explore novel techniques for one or more of the primary aspects of resource management that we have identified.

Work	Deployment Environment			Workload Management				QoS Goal	Resource Management Technique		
	Deployment Model	Application Isolation	Pricing Model Awareness	Application Model		Workload Nature Awareness	Data Locality Awareness		Workload Modelling	Resource Scheduling	Resource Scaling
				Structure	Scheduling Granularity						
[121]	Cloud	Kernel thread	-	Task	Single Function	✓	-	Latency, Resource Efficiency	-	Feedback control system, Heuristic	-
[24]	Edge/Cloud	Container	✓	DAG	Function Fusion	✓	✓	Latency, User Cost	-	LARAC	-
[25]	Edge/Cloud	Container	-	Task	Single Function	✓	-	Latency	-	Greedy, UCB1, Bayesian UCB	-
[70]	Cloud	Container	-	Task	Single Function	✓	-	Latency, Resource Efficiency	-	Non-cooperative game theoretic heuristic	Heuristic
[122]	Cloud	Optimized container	-	Task	Single Function	-	✓	Latency	-	Greedy	-

Work	Deployment Environment			Workload Management				QoS Goal	Resource Management Technique		
	Deployment Model	Application Isolation	Pricing Model Awareness	Application Model		Workload Nature Awareness	Data Locality Awareness		Workload Modelling	Resource Scheduling	Resource Scaling
				Structure	Scheduling Granularity						
[69]	Cloud	Container	✓	Task	Single Function	✓	✓	Latency, User Cost	Mathematical Modelling, Heuristic	Machine Learning	Machine Learning
[45]	VM/FaaS based	Container	✓	Task	Single Function	-	-	Latency, User Cost	Mathematical Modelling	Greedy	Heuristic
[123]	Cloud	Container	-	Task	Single Function	-	-	Latency, User Cost	-	Heuristic	-
[64]	Cloud	Container	-	Task	Single Function	✓	✓	Latency, Throughput	-	Greedy	-
[67]	Cloud	Container	-	Task	Single Function	✓	-	Latency, Throughput	ML Modelling	Heuristic	Heuristic
[52]	Cloud	-	✓	DAG	Single Function	✓	-	Latency, User Cost	Mathematical Modelling	-	-
[10]	Edge/Cloud	-	✓	Task	Single Function	-	✓	Resource Efficiency	-	Heuristic	-

Work	Deployment Environment			Workload Management				QoS Goal	Resource Management Technique		
	Deployment Model	Application Isolation	Pricing Model Awareness	Application Model		Workload Nature Awareness	Data Locality Awareness		Workload Modelling	Resource Scheduling	Resource Scaling
				Structure	Scheduling Granularity						
[53]	Cloud	Container/ MicroVM	✓	Task	Single Function	✓	-	Latency, User Cost	ML Modelling	-	-
[26]	Edge/Cloud	Container	✓	Task	Single Function	✓	-	Latency, User Cost	Mathematical Modelling	Heuristic	-
[50]	Cloud	Container	✓	DAG	Single Function	✓	-	Latency, User Cost	Mathematical Modelling	Greedy	-
[49]	Cloud	-	✓	DAG	Single Function	✓	-	Latency, User Cost	ML Modelling, Mathematical Modelling	-	-
[55]	Cloud	Container	-	DAG	Single Function	✓	✓	Latency, Resource Efficiency, Throughput	ML Modelling, Mathematical Modelling	Heuristic	Heuristic
[97]	Cloud	-	✓	DAG	Single Function	-	-	Latency	-	MILP	-

Work	Deployment Environment			Workload Management				QoS Goal	Resource Management Technique		
	Deployment Model	Application Isolation	Pricing Model Awareness	Application Model		Workload Nature Awareness	Data Locality Awareness		Workload Modelling	Resource Scheduling	Resource Scaling
				Structure	Scheduling Granularity						
[120]	Cloud	Process	-	Task/ DAG	Single Function	-	-	Latency, Resource Efficiency	Feedback control system	Feedback control system, Heuristic	-
[51]	Cloud	-	✓	DAG	Single Function	✓	-	Latency, User Cost	Mathematical Modelling	-	-
[93]	VM/FaaS based	Container	-	Task	Single Function	✓	-	Latency, Resource Efficiency	Mathematical Modelling	Heuristic	Heuristic
[54]	Cloud	-	✓	Task	Single Function	✓	-	Latency, User Cost	Mathematical Modelling	Heuristic	Heuristic
[135]	Cloud	Container	-	Task	Single Function	✓	-	Energy Efficiency	Mathematical Modelling	-	-
[72]	Cloud	Container	-	Task	Single Function	-	✓	Latency	-	Heuristic	Heuristic

Work	Deployment Environment			Workload Management				QoS Goal	Resource Management Technique		
	Deployment Model	Application Isolation	Pricing Model Awareness	Application Model		Workload Nature Awareness	Data Locality Awareness		Workload Modelling	Resource Scheduling	Resource Scaling
				Structure	Scheduling Granularity						
[77]	Cloud	Container	-	Task	Single Function	-	-	Latency, Resource Efficiency, Throughput	-	Heuristic	RL
[63]	Cloud	Container	-	Task	Single Function	✓	-	Latency, Resource Efficiency	-	Greedy	Heuristic
[137]	Cloud	Container	-	Task/ DAG	Function Fusion, Single Function	-	-	Throughput, Resource Efficiency	-	Heuristic	-
[56]	Edge/Cloud	Container	-	Task	Collection	✓	-	Latency	Mathematical Modelling	Heuristic	-
[138]	Cloud	Container	-	DAG	Single Function	✓	-	Latency, Resource Efficiency	Mathematical Modelling	Heuristic	Heuristic
[130]	Cloud	Container	-	Task	Single Function	-	✓	Latency	-	Greedy	Heuristic

Work	Deployment Environment			Workload Management				QoS Goal	Resource Management Technique		
	Deployment Model	Application Isolation	Pricing Model Awareness	Application Model		Workload Nature Awareness	Data Locality Awareness		Workload Modelling	Resource Scheduling	Resource Scaling
				Structure	Scheduling Granularity						
[62]	Cloud	Container	-	Task	Single Function	✓	-	Resource Efficiency, Latency	-	Greedy	Heuristic
[139]	Edge/Cloud	Container	✓	Task	Single Function	✓	✓	Latency, User Cost	-	Greedy	-
[66]	Cloud	Container	-	Task	Single Function	-	-	Latency, Throughput	-	Heuristic	RL
[44]	Cloud	Container	-	DAG	Function Fusion, Single Function	✓	✓	Latency	Mathematical Modelling	Heuristic	Heuristic
[82]	Cloud	Container	-	Task	Single Function	-	-	Latency	-	DRL	Heuristic

Table 2.1: Classification of Resource Management Techniques.

2.5 Industrial Serverless Computing Platforms and Frameworks

Currently, all the major cloud service providers have launched commercial serverless platforms, namely Amazon Web Services (AWS) Lambda, Google Cloud Functions, Azure Functions and IBM Cloud Functions. While they provide additional complementary services as well for serverless application executions, they require the function code to be composed in certain ways resulting in vendor lock-in in the long term. To overcome these limitations, several open source serverless frameworks have emerged over the years. These frameworks could be deployed on private cloud resources as well as on devices at the edge/fog, thus bringing the serverless computing capabilities on-premise with better flexibility. Figure 2.2 presents the existing commercial serverless platforms and some of the open source frameworks.

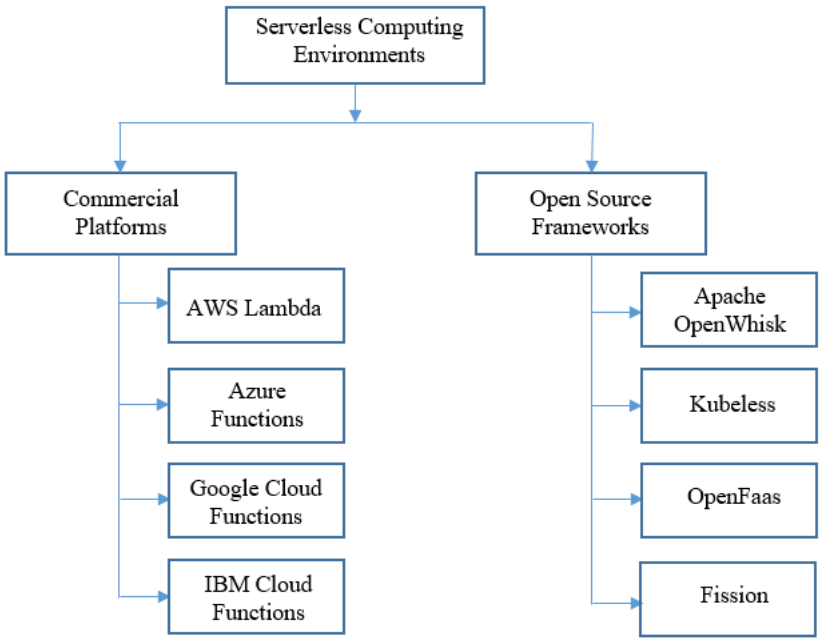


Figure 2.2: Existing Serverless Platforms and Frameworks.

AWS Lambda

Function as a service (FaaS) offering from AWS is done via the AWS Lambda serverless platform launched in 2014. AWS Lambda currently holds the leading position in the

market for serverless computing with a wide range of services available. Lambda supports code written in Node.js, Python, Java, C# and Golang languages [40]. The platform identifies changes to data in an Amazon Simple Storage Service (Amazon S3) bucket or an Amazon DynamoDB table, HTTP requests using Amazon API Gateway and API calls made using AWS SDKs, as event triggers. At the time of writing this article, AWS offers a free-tier of 1 million requests and 400,000 GB-Seconds (GB-s) of computing time per month. Beyond this limit, function executions are charged per GB-s of memory allocated and the number of execution requests. Lambda allows the developer to specify a maximum memory limit that the function will have access to, at the deployment stage, and allocates CPU power proportionate to the allowed memory limit [18]. AWS enables applications to be executed as composed of a single function or multiple functions. AWS Step Functions [18] facilitates developers to define workflows with a sequence of Lambda functions and create a state machine that orchestrates the set of functions in the application. AWS uses a new virtualization technology called Firecracker, to execute function requests [94]. These are lightweight micro-virtual machines (microVMs). AWS Lambda treats instance placmeent in VMs as a bin packing problem to maximize VM memory utilization. AWS has set a concurrency limit for scaling a single function and experiments show that idle instance recycling is done based on a pre-defined inactive period [39].

Azure Functions

Microsoft launched its serverless services in 2016 as Azure functions. Azure supports a number of runtime languages such as C#, Node.js, PHP, Bash, Power Shell. The billing scheme is similar to that of AWS except that billing is done per GB-s of average memory consumed during an execution instead of the memory allocated. Azure uses the concept of function app as the unit of deployment and management of a serverless application. A function app is comprised of one or more functions which are deployed, managed and scaled together. All the functions in a function app share the same pricing plan and runtime configurations [98]. Azure Functions seems to try not to co-locate concurrent instances of the same function on the same VM, which indicates a spread placement

approach [39]. Azure currently supports the execution of stateful functions as well in a serverless computing environment with Azure Durable Functions, which is an extension of Azure Functions [125].

Google Cloud Functions (GCF)

Google released its serverless solution in 2017. GCF supports code written in Node.js, Python, Go, Java, .NET, Ruby and PHP. Google has a free tier of 2 million requests with 400,000 GB-s of computing time per month, as of now. Their pricing scheme is different to AWS Lambda and Azure Functions pricing mechanism since the user is charged for both GB-s of memory provisioned and GHz-s of CPU provisioned [140]. GCF supports a set of primary triggers and additional triggers and the user application is allowed to be integrated with any Google service, supporting cloud Pub/Sub or HTTP callbacks.

Apache OpenWhisk

OpenWhisk is an open source serverless platform developed by IBM and later incorporated as an Apache incubator project. It represents the underlying technology used in IBM Cloud Functions. OpenWhisk combines technologies such as Nginx, Kafka, Docker and CouchDB in forming its serverless platform. OpenWhisk supports many deployment options and could be deployed both locally and within a cloud environment. The OpenWhisk programming model is based on the three primary concepts, actions, triggers and rules. Actions are functions that execute deployed code. Triggers are a set of events created from different sources. Rules bind actions with triggers. OpenWhisk supports a number of runtimes including, .Net, Go, Java, JavaScript, PHP and Python [141]. In selecting a VM instance for a function execution, OpenWhisk follows a hash-based first-fit heuristic where the function name's hash value is used to identify a preferred host in order to improve reusing warm containers [61]. Multiple functions, which may even be implemented in different languages, could be composed together to create a function pipeline called a sequence. A sequence could be considered a single action in terms of its creation and invocation. In executing a function sequence, the output from one action becomes the input to the next action in the sequence [141]. In addition, func-

tion sequences could be extended to create custom workflows with conditional branching, with support for error handling, retries and loops by means of the OpenWhisk Composer [126]. OpenWhisk makes use of pre-warmed containers and warm containers (container re-use) to manage unwanted container startup delays.

Kubeless

Kubeless is a Kubernetes-native open-source serverless framework developed by Bitnami [142]. Kubeless creates functions as a custom Kubernetes resource using a Custom Resource Definition (CRD). An in-cluster controller is used to watch these custom resources and execute functions on-demand by dynamically launching runtimes as required. Kubeless supports runtimes of Golang, Python, NodeJS, Ruby, PHP, .NET and Ballerina and allows to use HTTP or event triggers to invoke functions. The Kubeless framework uses core Kubernetes functionalities such as Deployments, ConfigMaps and Services as it is. It also leverages the native Kubernetes components for function scheduling, auto-scaling and monitoring as well.

OpenFaas

OpenFaas started as an independent project by Alex Ellis in 2016 [83]. Initially, it was developed in collaboration with VMWare and now it involves a large community of users and developers. OpenFaas framework is built on Docker and Kubernetes, and could be deployed on a private or public cloud environment, and even on a resource constrained edge device such as a Raspberry Pi, due its lightweight nature. OpenFaas provides an API gateway for invoking functions, which can be accessed via its REST API, Command Line Interface (CLI) or the User Interface (UI). The gateway acts as an external route to the functions, collects cloud native metrics through Prometheus [143], and also takes function scaling decisions with the help of Prometheus and an AlertManager component. AlertManager works by reading the requests per second metric from Prometheus and alerting the gateway to scale functions based on the min/max replica count set at function deployment. Alternatively users could use the built-in Horizontal Pod Autoscaler (HPA) of Kubernetes. OpenFaas offers runtimes of Node.js, Python,

and Go for function deployment. OpenFaas also supports the orchestration of multi-function workflow applications with synchronous and asynchronous function chains and parallel branching.

2.6 Research Gaps

This detailed review on resource management under the emerging serverless computing model on edge, fog and cloud resources highlights open problems with great potential for exploration in future work. Here we discuss these areas in detail along the broader categories we have identified in this article, laying the groundwork for both research and development work in the coming years.

2.6.1 System Design Characteristics

Serverless computing model has been explored under different deployment environments with hybrid infrastructures with geographical as well as performance distribution. These include a mix of edge, fog and cloud infrastructures along both private and public domains. However, one notable challenge for developers across all these domains, in adapting to the serverless model is the concern over the vendor lock-in effect due to the unique function signatures, naming conventions and other compatibilities required by each provider. An application designed to suit one vendor platform may not necessarily be readily deployable on another. This imposes an unnecessary burden on potential users.

Another prominent challenge for a majority of users in embracing this new paradigm is its lack of efficiency in accommodating certain application workloads, even if the application itself is designed to suit a serverless architecture. For example, a latency sensitive application workload with a high regular traffic pattern could benefit less from a serverless deployment. While it may not be able to tolerate the cold start delays of a serverless system, the high per request charge under the associated billing model may render it less viable cost-wise as well. In contrast, a burst of traffic received by an application with little or no sensitivity to response time would find the features of a serverless system to be ideal.

In the current commercial serverless platforms users are not allowed to specify the runtime hardware environment for their function executions. In terms of processing capabilities, they only offer access to CPU processing for the deployed applications. With increased exploitation of this cloud model for different application domains, there has

been a rise in demand for access to specialized hardware, which still remains unfulfilled.

The serverless model is built on the premise that the provider is at liberty to control the underlying infrastructure as they wish. Although not currently explored extensively, this opens up interesting opportunities for them to seek maximum resource efficiency by leveraging idling resources from other running services as well as less attractive machines not conducive for leasing out to IaaS customers [27].

To date, the billing models in existing serverless platforms offer only fixed pricing schemes to all consumers. Customers having different requirements may be willing to pay more or less for enhanced or standard levels of service. For example, a generalized level of service may not be sufficient for time-sensitive critical applications requiring a reliable performance level, such as in the medical domain.

2.6.2 Workload Management and QoS Goals

Serverless platforms still face many challenges when being explored for some applications with complex data flow dependencies [138]. Lack of an efficient function to function communication mechanism complicates the process of orchestrating complex workflow structures. Most of the real-world applications would generally have complex workflow structures and orchestrating such applications is still an open challenge.

Current commercial serverless platforms offer little to no QoS guarantees (latency, reliability etc.) for applications of its multitude of users. Allowing too many specific demands from users could also lead to poor resource efficiency for the provider and thus this requires thorough investigation. For example, the maintenance of large resource pools for mitigating application latency deterioration would have very low attraction to a provider if it leads to a large resource wastage.

Further, many resource scheduling models proposed for the serverless model focus on meeting the latency and budget constraints of consumers and rarely on the efficient resource usage at the provider. With the provider centric resource management model introduced by this new pradigm, studying techniques favorable for serverless vendors in maximizing their gains is critical for the progress of this concept.

With the increased use of this model under various infrastructure domains specially

in largely distributed and resource constrained edge environments, the aspect of energy efficiency seems to have a high significance. But this area of research still remains mostly unexplored.

2.6.3 Resource Management Techniques

Increased level of abstraction in specifying resource usage by developers leaves the serverless provider with the need to infer resource requirements and code dependency requirements in making appropriate resource allocation and scheduling decisions. Serverless platforms cater to needs of diverse applications with vastly differing characteristics and requirements. Understanding and profiling of workloads as per their resource needs and arrival patterns is a massive challenge. Nevertheless, this largely assists in resource scheduling and scaling decisions that could benefit both the consumers and providers. Although existing research directed towards modelling application performance and cost help in taking static scheduling decisions such as deciding on initial resource configurations and placement decisions, most of these approaches do not address dynamic factors that would affect performance in the runtime. Applications may not perform as required due to co-located application interference, machine performance variations and degradation due to over-utilization. Moreover, dynamic variations in workload patterns need to be taken into account when taking these high impact decisions.

Although initial efforts are made in extending serverless model across the edge and fog computing networks, much further investigation is required in this area for taking effective resource management decisions. Given the involved delays across networks, transporting data and code to the edge for individual function executions as per the cloud serverless model would be ineffective specially for latency sensitive applications. Limitations caused by heterogeneity of edge and fog devices, on handling compute intensive function executions need to be considered more in scheduling and scaling decisions. Data locality concerns inherent with the serverless model may be even more relevant for geographically distributed edge/fog computing networks.

Cold starts arising from the unique auto-scaling capability is a growing research

area. Numerous solutions ranging from proactive resource scaling to reduce coldstart frequencies, to designing optimized sandbox solutions are proposed. The need to cater to a variety of concurrent applications of a multitude of users in designing these solutions, is yet quite an unresolved challenge.

2.7 Summary

In this chapter we presented a comprehensive review on the aspect of resource management, referring to unique characteristics of this new serverless computing model. We proposed a taxonomy covering the broader concept of resource management in edge, fog and cloud infrastructures, along the categories of system design decisions, approaches for incoming workload identification and management, and the QoS goals of involved parties. We also discussed three key aspects of resource management techniques and analysed the existing works using the proposed taxonomy. This taxonomy presents a clear view for serverless system designers on the essential features for consideration for a fully-fledged effective system. Further this provides a learning platform for researchers studying resource allocation, scheduling and scaling techniques in the serverless domain, based on which their work could demonstrate progress in the field. Finally, we provided a gap analysis referring to identified and addressed challenges, emphasizing on the vast potential for further research work.

This thesis investigates and addresses some of these research gaps and presents the potential for new research directions in the last chapter.

Chapter 3

Deadline-aware Dynamic Resource Management in Serverless Computing

The total shift in operational responsibility from the user, challenges the cloud provider to maintain acceptable performance while having minimal knowledge of the application requirements. In addition, the serverless billing model favors users by charging only for resource time during function executions, while the provider maintains underlying infrastructure for longer periods. This chapter focuses on both the provider and user's perspectives and proposes a function placement policy and a dynamic resource management policy for applications deployed in serverless computing environments. The policies aim to meet the user's specific performance requirements, while minimizing the resource consumption cost for the service provider. We implement and evaluate our approach through simulation using ContainerCloudSim toolkit. The proposed function placement policy when compared with baseline scheduling techniques can reduce resource consumption by up to three times. The dynamic resource allocation policy when evaluated with a fixed resource allocation policy and a proportional CPU-shares policy shows improvements of up to 25% in meeting the required function deadlines.

3.1 Introduction

The many attractions of the serverless computing paradigm include rapid auto-scaling, strong isolation for applications, a fine-grained billing mechanism and more importantly, the access to a service ecosystem, which automatically handles instance selection,

This chapter is derived from:

- **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "Deadline-aware dynamic resource management in serverless computing environments", *Proceedings of the 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, Pages: 483-492, Melbourne, Australia, May 10-13, 2021.

resource management, fault tolerance, monitoring, and security [27].

The core concept behind the serverless execution model is to shift the complexities of application resource management from the developer to the cloud provider. The serverless model requires the provider to autonomously manage resource allocations to functions in real time, in contrast to a service placement scenario under a serverful model (e.g. Infrastructure as a Service), where the user configures the environment with required resources prior to application execution [27]. Serverless platforms have minimal knowledge on the resource requirements of different functions, at the time of initial resource allocation. For instance, AWS Lambda allows the user to specify the amount of memory available to the function during execution and allocates the CPU power linearly in proportion to the configured memory [18]. Google Cloud Functions seems to adopt the same strategy for resource allocations [144]. Studies show that CPU is often a cause for contention in serverless environments, specially when compute intensive applications are involved [63], leading to high application latencies. Thus, any arbitrary resource allocation policy could lead to subsequent resource contentions for the applications during runtime, leading to Service Level Agreement (SLA) violations to the user. Thus arises the need for dynamic resource management techniques.

As per the serverless deployment model, the user is charged only for the resource-time actually consumed by the application during its execution. Regardless of this fact, the cloud provider maintains the underlying Virtual Machine (VM) resources during its entire active life time. The resource-time covered by a VM during its life time comprises of its set up time and its entire active time when one or more functions are using the VM resources either fully or partially. As the serverless model is being increasingly experimented for longer running tasks such as massively parallel task executions as in [11], [145], such partial resource usages is more prevalent. Hence its imperative, that the cloud provider maximizes the utilization of the set of active VMs at any given time, thus reducing the cost of maintaining too many underutilized VMs. On the other hand, when an application has a deadline target on execution as part of the SLA, the placement decision of a function instance on an available VM needs to consider optimizing the resource usage of VMs, without compromising on the stated execution time limitations.

Many existing serverless platforms follow different strategies to manage their under-

lying infrastructure. Experimentation done on AWS Lambda platform indicate that the function placement decision is currently treated as a bin packing problem to maximize VM memory utilization [39]. Azure Functions seems to try not to co-locate concurrent instances of the same function on the same VM, which indicates a spread placement approach [39]. IBM OpenWhisk uses a hash-based first-fit heuristic which aggregates application executions by function type, aimed at improving instance re-use and cache hit rate [61]. Docker Swarm employs a spread placement algorithm which tries to evenly spread tasks across the nodes in a cluster [146].

While some of the above approaches try to maximize resource utilization in function executions, they do not consider application specific details and hence could result in SLA violations, while not achieving optimal resource usages. In the literature, many works exist, which address the problem of reducing function response time to users and optimizing resource cost for the end user [70], [122], [147], [50]. However, the importance of dynamic management of allocated resources to function instances in the runtime, considering user requirements, and also the problem space of efficient resource management on the provider side have not been studied extensively.

Thus, a major challenge for the service provider under this model is to choose a suitable compute node for the function request placement and allocate sufficient resources to the containerized function instance, such that the desired user requirements are met, and the cost of resources is maintained at an optimum level.

In this work, we present a deadline-sensitive heuristic algorithm for selecting a VM for function execution, which tries to manage the VM resources efficiently in order to minimize the provider cost of maintaining the cloud infrastructure. Further, we present an approach to dynamically monitor and manage the allocated CPU resources to function instances in the runtime, targeted at meeting the user deadlines, irrespective of the initial assignment of resources. We implement our proposed policies in ContainerCloudSim [148] simulation environment and conduct experiments using real-world and synthetic traces. The experimental results show that our policies increase the efficiency of VM resources and also perform better in terms of meeting the function deadlines, as compared to baseline techniques.

The key **contributions** of our work are as follows:

1. An efficient placement algorithm for function requests, which aims to enhance VM resource efficiency.

2. A fine-grained approach to dynamically manage resource allocations to functions.

3. Implementation of our proposed policies in a simulation environment and conducting extensive experiments using both real-world and synthetic workloads.

4. Evaluation of the efficiency of our proposed solution in comparison with baseline load balancing algorithms and two resource allocation policies, namely, a fixed resource allocation policy and a cpu-shares policy in terms of VM resource usage and meeting the function deadlines.

The rest of the chapter is organized as follows. Section 3.2 highlights related research. In section 3.3, we show the system model and formulate the scheduling problem. Section 3.4, presents our proposed approach. In section 3.5, we discuss the experimental set-up and present the performance evaluation of our proposed method. Section 3.6 concludes the chapter and highlights future research directions.

3.2 Related Work

Serverless computing as a cloud application deployment model, is still at an early stage of being widely adopted and explored in different application domains. As such, research work referring to efficient resource management in serverless platforms are still growing and also refer to many diverse aspects. Here we focus on key research work related to application resource management in serverless platforms.

HoseinyFarahabady et al. present a QoS-aware resource allocation controller for serverless platforms [121]. The scheduler aims to dynamically scale resources by predicting the future rate of incoming events using a closed-loop model predictive mechanism. Although the controller tries to maintain a healthy CPU utilization level at each host, specific focus is not given to using application level details to minimize provider cost. Overall this work addresses the challenge of the initial placement of functions, but the handling of sub-optimum resource allocations is not discussed.

A package-aware scheduler is proposed by Abad et al. [122] for serverless functions. The objective is to reduce the cold start latency by assigning functions requiring similar

packages to the same node. The efficiency of this model could be affected by functions having multiple, large package dependencies. Further, this model ignores workload characteristics other than the package requirements and does not focus on optimizing cloud resource usage.

Stein et al. [70] present a non-cooperative resource allocation heuristic which tries to predict the number of function instances required to keep the request waiting time below a chosen threshold. This work considers container re-use and pre-warming of containers to reduce response times. Meeting any function specific user requirements has not been focused on, in the proposed approach.

Mahmoudi et al. [67] present a function placement algorithm which uses a machine learning based approach for selecting the VM for a new function invocation, so as to reduce operational cost to the user. They explore a predictive performance model which tries to predict the normalized performance for any workload when deployed to a specific VM. Their approach takes in to account the nature of the workload such as CPU, disk or memory intensiveness in making its decision. The model requires a profiling step each time a new function is deployed in the platform.

A latency-aware function scheduler is presented by Suresh et al. [63]. Their model is focused on dynamically adjusting cpu-shares [149] of containers based on the latency degradation to each application type as a whole. The greedy algorithm presented for VM scaling results in reduced number of VMs being used compared to spread placement approaches, but there is no specific focus on reducing partial VM usages. They achieve reduced latency degradation to applications via adjusting cpu-shares of containers. Since cpu-shares is a relative allocation of CPU to each container, the application performance largely depends on the co-located functions in a VM.

A core-granular scheduler for serverless environments is introduced by Kaffes et al. [123]. In this model, the scheduler assigns functions directly to individual cores aiming to eliminate overloading of cores and reducing co-located function interference. Consideration for any workload specific requirements is not observed in the proposed approach.

Singhvi et al. implement a low-latency serverless platform for DAG based applications [147]. The design entails a set of node clusters with semi-global schedulers, which

Table 3.1: Summary of Literature Study.

Work	Application Model		Deadline Awareness	Efficient VM usage	Dynamic Re-sizing	Dynamic Re-scheduling
	Task(Single Function)	DAG(Multiple functions)				
HoseinyFarahabady et al. (2017)	✓		✓	✓		
Abad et al. (2018)	✓					
Stein et al. (2018)	✓					
Mahmoudi et al.(2019)	✓					
Suresh et al. (2019)	✓		✓	✓	✓	
Kaffes et al. (2019)	✓					
Singhvi et al. (2019)		✓	✓			
Das et al. (2020)		✓	✓			
Kim et al. (2020)	✓	✓		✓	✓	
Our proposed work	✓		✓	✓	✓	✓

follow deadline-aware function scheduling. Although a deadline is considered for initial function placement, the subsequent management of allocated resources to functions is not discussed.

Das et al. [50] propose a hybrid cloud scheduling framework for multi-function serverless applications. They suggest a greedy algorithm to decide the order and placement of each function in either the private or the public cloud. The objective is to minimize the cost of public cloud use for the consumer and to complete the execution of a batch of jobs within a specified deadline.

Kim et al. [120] propose a technique for CPU resource management for serverless worker processes based on the throttled time and the number of unprocessed functions in the queue of each worker. They try to reduce the function response time and increase the CPU utilization of the worker processes. They do not consider application level details or requirements in their load balancing policy and thus may not be responsive to specific user execution time limitations and achieving optimal resource usage levels.

A summary of the reviewed related works is presented in Table 3.1, comparing them in terms of the focus on efficient VM resource usage, application deadline awareness, dynamic resource re-sizing and re-scheduling, and the application model. Although a few works discuss increasing the VM resource utilization levels in general, they are not specifically focused on reducing provider cost by making use of application level details and requirements. Further, while many works present strategies to optimally place the functions initially, they rarely discuss the performance during its full life cycle.

In our work we present a comprehensive function placement algorithm which aims

to manage VM resources efficiently and thereby reduce the underutilization of VMs by considering the function deadlines, in making the placement decision. We also propose an algorithm for the dynamic management of resource allocations to functions instances, in order to meet the application deadlines.

3.3 System Model and Problem Formulation

We present the serverless system model used in our work and formulate the problem of scheduling functions in VMs.

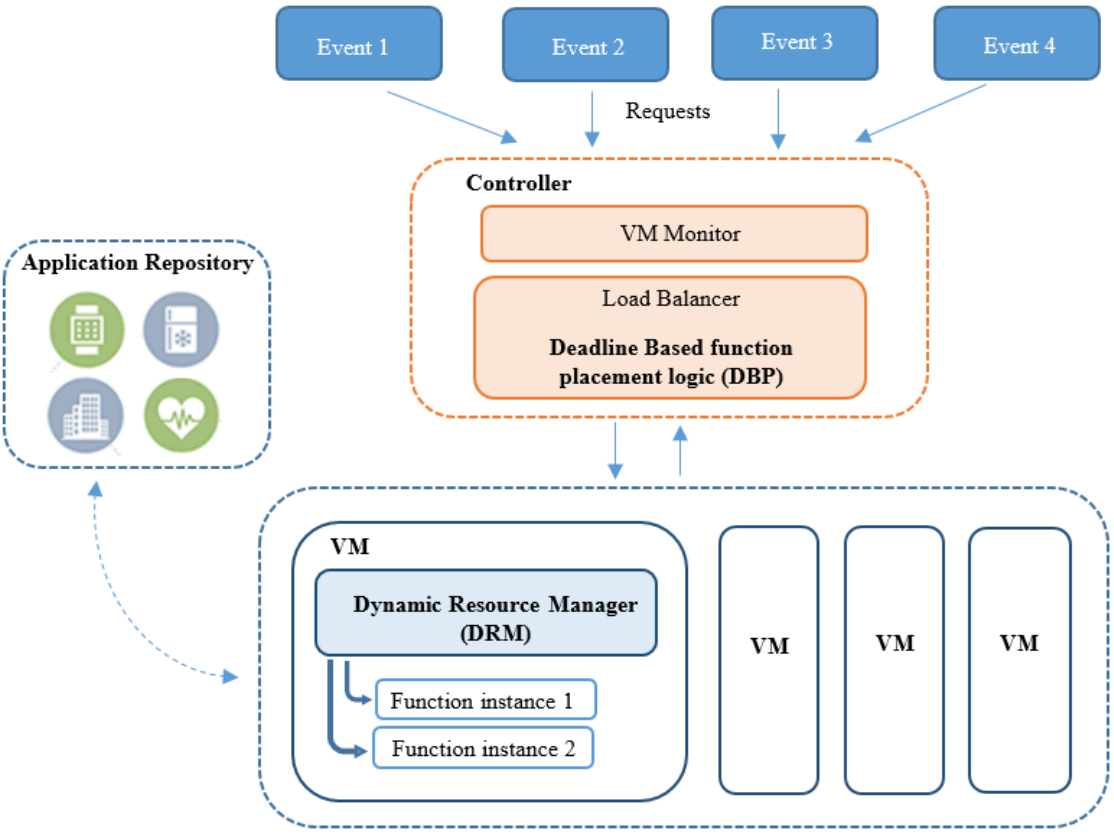


Figure 3.1: System Model.

3.3.1 System Model

We follow a similar approach to Apache OpenWhisk serverless platform [150]and also many commercial serverless platforms, while designing the system model for evaluating our proposed approach to address the challenges mentioned above. Figure 3.1 illustrates the high-level system components involved in our model.

Function invocation requests created from user initiated events are received at the system controller. The controller contains the function placement logic which handles the load balancing responsibility in choosing a compute node for function execution. The VM monitor module constantly updates and retains meta data on the expected remaining VM runtimes and the functions in execution in each VM. This information is used by the load balancer in its decision making. Once a suitable node is selected, the requests are dispatched to the destination nodes.

A compute node represents an active VM available for task execution. An active VM would contain multiple concurrent functions in execution inside containers. A container with the required resource configurations is spawned for a new request execution. The VM loads the required runtime and the associated application code from the application repository, on to the launched container. As per our proposed approach, the Dynamic Resource Manager (DRM) module which is deployed on each VM, is responsible for monitoring the functions in runtime and handling dynamic resource re-provisioning to containers as applications approach their deadlines. It causes a dynamic update to a container's CPU resource limits until the task completes its execution. This module also enables eviction and re-scheduling of low priority task executions to avoid performance degradation in VMs due to resource contentions. Functions which were evicted are re-scheduled on a different node, by sending a new function invocation request to the controller where the function placement logic is called again to find a suitable compute node.

3.3.2 Problem Formulation

Based on the system model, we now formulate the function scheduling problem to minimize the provider cost of VM resources usage, and to meet execution deadlines of the

functions. For the scope of this work we consider an application to be composed of a single function and hence the terms "function" and "application" are used interchangeably. Table 3.2 summarizes the important notations and descriptions presented in this chapter.

Given an instance of a serverless platform, let $V = \{v_1, v_2, ..., v_N\}$ be the set of VMs or compute nodes available for function execution, where N is the total number of VMs and v_j, $1 \leq j \leq N$ is the j^{th} VM. Each VM has available resource capacities defined by a two-dimensional vector: CPU and memory, represented as v_j^c and v_j^m respectively. Hence we have, $v_j =< v_j^c, v_j^m >$. The total CPU capacity in a VM is determined as the product of the number of cores and the processing power of each core, denoted in Million Instructions Per Second (MIPS). The available free CPU and memory resources in VM, v_j at time t is denoted by, $v_j^{ac}(t)$ and $v_j^{am}(t)$ respectively.

Let $R = \{r_1, r_2, ..., r_M\}$ be the sequence of function invocation requests received at the scheduler where M is the total number of requests and r_i, $1 \leq i \leq M$ is the i^{th} request. Each request carries five attributes, i.e., $r_i =< r_i^{type}, r_i^{priority}, r_i^{ta}, r_i^d, r_i^m >$ where r_i^{type} represents the application ID of the function to be invoked, $r_i^{priority}$ denotes the user requested priority level for the request i, and r_i^{ta}, r_i^d and r_i^m are the time of arrival, specified deadline and the memory requirement of request i respectively. We assume the initial CPU allocation r_i^c, to the containerized function instance, to be done in proportion to the requested memory, adopting AWS Lambda's initial resource allocation policy [18] (i.e. if r_i^m is the requested memory and v_j^m and v_j^c are the total memory and CPU capacities of the VM, the allocated CPU capacity is $(r_i^m / v_j^m) * v_j^c$). Since we dynamically update CPU allocations to the function instances in the runtime, we use the notation $r_i^{uc}(t)$ to denote the updated CPU allocation to request r_i subsequently in time t.

Now the challenge at hand is to decide the mapping of a request execution to an available VM where the application would start its execution inside a container with access to assigned resources, and to manage resource allocations to its containerized instance throughout the life time.

$$Schedule = \{r_i \rightarrow v_j\} \tag{3.1}$$

Function scheduling and resource allocation would be subject to the following con-

Table 3.2: Definition of Symbols.

Symbol	Definition
v	A VM or compute node available for function execution
N	The total number of available VMs
δ	The index set of all the available VMs, $\delta = \{1, 2, 3, ..., N\}$
M	The total number of function invocation requests
r	Function invocation request
r_i^{type}	Type of the function to be invoked by i^{th}request, r_i
$r_i^{priority}$	Requested priority level for the execution of i^{th}request, r_i
r_i^{ta}	Time of arrival of i^{th}request, r_i
r_i^{d}	Deadline for i^{th}request, r_i
r_i^{m}	Memory requirement for i^{th}request, r_i
r_i^{c}	Initial CPU allocation for i^{th}request, r_i
$r_i^{uc}(t)$	The CPU allocation for i^{th}request, r_i at time t
r_i^{w}	The waiting time for scheduling i^{th}request, r_i
r_i^{p}	Total processing time of i^{th}request, r_i
v_j^{m}	Total memory capacity of j^{th} VM, v_j
v_j^{c}	Total CPU capacity of j^{th} VM, v_j
$v_j^{am}(t)$	Available free memory in j^{th} VM, v_j at time t
$v_j^{ac}(t)$	Available free CPU capacity in j^{th} VM, v_j at time t

straints.

VM resource capacity constraints: A VM is chosen for function execution only if the requested memory and the initial CPU allocation requirements for a function execution do not exceed the available free memory and CPU capacities of the VM at time t, i.e.,

$$r_i^m \leq v_j^{am}(t) \tag{3.2}$$

$$r_i^c \leq v_j^{ac}(t) \tag{3.3}$$

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

$$v_j^{am}(t) = v_j^m - \sum_{i=1}^{M} u_{ij}(t) r_i^m(t) \tag{3.4}$$

$$v_j^{ac}(t) = v_j^c - \sum_{i=1}^{M} u_{ij}(t) r_i^{uc}(t) \tag{3.5}$$

where we define a binary variable u_{ij} to indicate whether request i is currently placed in v_j or not, i.e., $\forall j \in \delta$, we have;

$$u_{ij}(t) = \begin{cases} 1, & \text{if request } i \text{ is being executed in } v_j \text{ at time } t \\ 0, & \text{otherwise} \end{cases} \tag{3.6}$$

The time t in the above expressions: (3.2), (3.3), (3.4), (3.5) and (3.6) refers to the request arrival time i.e., r_i^{ta}.

Overall, the primary focus of this study is to minimize the provider expenses of running serverless applications by efficiently utilizing resources in VMs, while also minimizing the violation of user requirements of function execution.

In our work, we assume all the compute nodes to be homogeneous, and thereby the combined uptimes of all the VMs is representative of the provider's opportunity cost of utilizing the same resources for revenue generation from other services. Thus, we formulate the optimization problem for resource-efficient function scheduling as follows:

$$\begin{aligned} Minimize : T &= \sum_{j=1}^{N} t_j \\ s.t. &: (2), (3) \end{aligned} \tag{3.7}$$

where t_j is the sum of the active periods of j^{th} VM over the course of the experiment and T is the summation of the active periods of all the VMs used in the experiment. A VM is considered to be active when at least one container is running in it. We assume that VMs are available on-demand without additional start-up delays. Primarily, our proposed function placement logic contributes towards realizing this objective.

We also formulate the objective of minimizing user deadline violations as follows:

$$\begin{aligned} Minimize : Z &= \sum_{i=1}^{M} x_i \\ s.t. &: (2), (3) \end{aligned} \tag{3.8}$$

where we define a binary variable x_i to indicate whether request r_i violates its deadline or not, i.e., $\forall i \in M$, we have;

$$x_i = \begin{cases} 1, & \text{if } r_i^w + r_i^p > r_i^d \\ 0, & \text{otherwise} \end{cases} \tag{3.9}$$

where r_i^w is the waiting time for function scheduling, r_i^p is the processing time, r_i^d is the user specified deadline and Z is the total number of deadline violations. The deadline for a function indicates the expected maximum time to finish execution from the request arrival time. Our approach of dynamic resource management mainly contributes towards meeting the objective of deadline satisfaction.

3.4 Proposed Algorithms

We propose a heuristic algorithm for the resource efficient placement of functions, and a dynamic resource alteration algorithm to solve resource contentions in VMs in the runtime and to meet user specified deadline constraints.

3.4.1 Function Placement Algorithm

The proposed heuristic function placement algorithm (Algorithm 1) follows a deadline sensitive function aggregation policy. The algorithm aims to align the runtimes of functions executing in a particular VM, such that the underutilization of VM resources is minimized, allowing the instance to be released after experiencing high utilization during its active life time.

We maintain a list of the existing active VMs ($VMList$) in the ascending order of the remaining time each of them are expected to run, depending on the functions already in execution and their stated deadlines. The remaining time to deadline $r_i^{\Delta t}(t)$, for request

Algorithm 1 Deadline Based Function Placement (DBP)

Input: The function invocation request r_i,
$r_i =< r_i^{type}, r_i^{priority}, r_i^{ta}, r_i^{d}, r_i^{m} >$
Input: The list of active VMs sorted in the ascending order of the expected remaining runtimes, $VMList$
Output: VM selected for function execution, v_s

1: **procedure** PROCESSVMSELECTION(r_i)
2: $r_i^{\Delta t}(t) \leftarrow$ Time to deadline for r_i
3: $CPU_{max} \leftarrow 0$
4: **for** each VM v_j in $VMList$ **do**
5: **if** r_i is a request for re-scheduling **then**
6: **if** $v_j = r_i.GetOldVm$ **then**
7: **continue**
8: $v_j^{\Delta t}(t) \leftarrow v_j.GetRemainingRunTime$
9: $v_j^{util}(t) \leftarrow v_j.GetCPUUtilization$
10: **if** Placement of r_i in v_j satisfies resource capacity constraint and $v_j^{util}(t) < v^{util_T}$ **then**
11: $v_{temp} \leftarrow v_j$
12: **if** $v_j^{\Delta t}(t) \geq r_i^{\Delta t}(t)$ **then**
13: **if** $r_i^{priority} = Low$ **then**
14: $v_s \leftarrow v_j$
15: **break**
16: **else**
17: **if** $v_j^{ac}(t) > CPU_{max}$ **then**
18: $CPU_{max} \leftarrow v_j^{ac}$
19: $v_s \leftarrow v_j$
20: **if** $v_s = null$ **then**
21: **if** $v_{temp}! = null$ **then**
22: $v_s \leftarrow v_{temp}$
23: **return** v_s
24: **else**
25: Add a new VM, $vmNew$ to the active pool
26: $v_s \leftarrow vmNew$
27: **return** v_s
28: **else**
29: **return** v_s

r_i at time t is expressed as follows:

$$r_i^{\Delta t}(t) = r_i^{ta} + r_i^d - t \tag{3.10}$$

where r_i^{ta} is the arrival time and r_i^d is the deadline, of request r_i respectively. We assume, that the longest of the remaining times to deadlines of the current functions running in a VM to be an approximation of its expected remaining runtime. The $VMList$ is updated whenever a new request is allocated to a VM or a function completes its execution.

When a new function invocation request (r_i) arrives at the controller, its time to deadline is calculated (line 2). Next the algorithm starts its iteration over the sorted list of active VMs. While we aim to maintain high VM utilization levels with the provider, for the set of active VMs at all times, we also try to avoid potential performance degradation caused by CPU contentions arising with resource overloading. Therefore, apart from the availability of sufficient free resources, a VM, v_j is considered for function placement only if its current CPU utilization, $v_j^{util}(t)$ is below a defined CPU utilization threshold v^{util_T} (line 10). If these requirements are met, this VM is chosen for execution if the expected remaining time of the VM is greater than the time to deadline of the request and the request bears a low level of priority (line 12-15). In case r_i is a high priority request, we choose the VM with the highest free CPU resources out of the VMs having higher remaining runtime than the time to deadline for r_i (line 17-19). We assume each request to be accompanied with either a high or low priority level and the high priority requests to have a tighter deadline than requests with low priority. Assigning a high priority request to a relatively less congested VM gives a better opportunity to dynamically monitor and increment the allocated CPU resources to the request in the runtime, if needed. It could be that none of the active VMs with sufficient free resources have the required remaining active time. In that case we choose the VM with the highest remaining active time. This ensures that the increase to the expected runtime of the chosen VM by the new function execution will be minimum (line 21-23). In case r_i is a request for re-scheduling, we avoid assigning the request to the same node it was previously being executed in (line 5-7). If none of the active VMs have sufficient free capacity for the function execution, a new free VM is added to the pool (line 25-27).

Algorithm 2 Dynamic Resource Alteration (DRA)

Input: The function invocation request r_i
Input: Current VM, v_j and current container, c_{ij} of r_i
Input: The list of requests in execution in v_j sorted by time to deadline in descending order, r_{list}

```
1:  procedure ProcessResourceAlteration(r_i)
2:      v_j^util(t) ← v_j.GetCPUUtilization
3:      if v_j^util(t) ≥ v^util_T or c_ij has reached its max CPU allocation then
4:          return Failure
5:      else
6:          c_ij.UpdateResources (CPU)
7:          v_j^util ← v_j.GetUpdatedUtilization
8:          if v_j^util ≥ v_j^util_T then
9:              for each request r in r_list do
10:                 if r satisfies re-scheduling criteria (3.12) then
11:                     c_r ← r.container
12:                     Re-schedule request r
13:                     Destroy c_r
14:                     v_j^util ← v_j.GetUpdatedUtilization
15:                     if v_j^util < v^util_T then
16:                         break
```

Once a VM is selected, the request r_i is forwarded to the selected compute node v_s for execution, where a new container with the requested resources is created and the function execution is initiated. Assuming the total number of VMs to be n, the worst case time complexity of Algorithm 1 for selecting a worker node, is $O(n)$.

3.4.2 Dynamic Resource Alteration (DRA) Algorithm

The proposed *Dynamic Resource Alteration* (DRA) algorithm (Algorithm 2) aims to alter the resource allocations to function instances approaching the deadline during runtime. The algorithm also efficiently manages resource contentions in the VMs by evicting suitable recently started tasks from constrained nodes to nodes with sufficient free resources.

The algorithm is executed by the VMs each time a task reaches a certain percentage of the task's time to deadline from the arrival time, denoted by d_{check}. We decide this checkpoint based on the level of priority requested by each request on arrival (for example,

a high priority request would have a better opportunity of meeting the deadline from a lower d_{check} value). At this point, if the function is still in execution, the initially allocated upper limit of CPU processing power to the container is incremented, provided that the underlying VM's CPU utilization level is below v^{util_T} and the container has not reached its maximum CPU allocation (line 3-6). We assume the maximum CPU power allocated to a container to be equal to that of one full vCPU core. Here we use the concept of cpu-quota and cpu-period enabled in Linux Kernel's Completely Fair Scheduler (CFS) [81], in setting and updating the CPU upper limits of the container. The cpu-quota value sets the number of microseconds per cpu-period that the container's access to CPU resources is limited to, before it is throttled [149]. Thus this acts as an effective ceiling and a hard limit for CPU resources allocated to a container. This is in contrast to the concept of cpu-shares used in [63], which adjusts the relative weight of CPU resources accessible to a container when co-located with other containers [149]. Container orchestration technologies allow updating the container resource configurations in the runtime [151]. During the evaluation of this approach, we conduct experiments while varying the d_{check} value and the cpu-quota increment values with the request priority levels.

After each resource update, the VM is checked for resource overloading (line 8). In the presence of resource overloading and performance interference as a result, the algorithm proceeds to efficiently evict some of the most recently scheduled tasks, scheduling them on another suitable node with sufficient free resources (line 9-12). Task eviction is undertaken only if the re-scheduling cost satisfies criteria (12) below. The re-scheduling cost r_i^{cost} at time t for the i^{th} request is defined by the time spent from arrival of r_i and the waiting time for function re-scheduling r_i^w, i.e.,

$$r_i^{cost}(t) = t - r_i^{ta} + r_i^w \tag{3.11}$$

$$r_i^{cost}(t) \leq r_i^d \times r^{evict_T} \tag{3.12}$$

where r_i^d is the task deadline and r^{evict_T} is a defined threshold for function eviction. This eviction policy ensures that the task execution loss is minimized and the evicted task has sufficient re-scheduling time until its deadline. Once a task is chosen for eviction, its running container is destroyed, freeing up resources in the constrained node. Re-

scheduling of a task follows the function placement algorithm (Algorithm 1), by sending a function placement request to the controller. A node continuously monitors each function and employs the DRA algorithm, each time the time to deadline approaches the defined checkpoint. The process continues until the function finishes its execution or its container occupies a full vCPU core, which is assumed to be the maximum allocated CPU capacity for a function instance.

If v_j, the current VM of r_i, has r number of requests already in execution, under the worst case scenario, Algorithm 2 has a linear time complexity of $O(r)$.

3.5 Performance Evaluation

To evaluate the performance of our algorithms, we simulate a serverless computing environment using ContainerCloudSim [148] simulator. It is a simulation toolkit developed for modeling containerized cloud infrastructures. ContainerCloudSim is built on top of CloudSim [29] simulator which is widely used in evaluating resource management and scheduling techniques in cloud environments. We extended the simulator by implementing the Dynamic Resource Manager and VM Monitor modules as described in section 3.3, to include our scheduling and dynamic resource management policies.

3.5.1 Baselines

We compare our function placement policy with the following baseline scheduling policies:

Round Robin (RR): This method tries to equally balance the load among the VMs by sending successive function requests to different VMs in a cyclic manner

Random Placement (RP): Function requests are randomly distributed among the VMs

Bin packing First-Fit (BPFF): Each request is directed to the first VM which satisfies the resource requirements of the request out of the active VMs, similar to AWS Lambda's function placement policy [39], packing the requests within a fewer VMs as possible.

Further, we compare our Dynamic Resource Alteration (DRA) technique with the following techniques:

Fixed Resource Allocation (FRA): The cpu-quota allocation to each container is done in proportion to the requested container memory and this CPU upper limit is maintained throughout the application lifetime, similar to the policy used in AWS Lambda serverless platform [39],[18].

OpenWhisk Resource Allocation (OW): OpenWhisk [150] sets the cpu-shares for each container proportional to the requested memory for each function, as mentioned in [63]. Cpu-shares indicates the relative weight given to a container in terms of the proportion of CPU time it is given access to when CPU resources are limited [149].

3.5.2 Experimental Set-up

We simulate a serverless computing environment with a cluster of VMs, each with four vCPU cores and 3 GB of memory. We follow the CPU configuration of Intel E5-2666 (2.9 GHZ), identified as one of the machine configurations seen in AWS Lambda infrastructure [39]. Since ContainerCloudSim identifies processor capacity in terms of MIPS (Million Instructions Per Second), we refer to CISCO's industry benchmark [152] in converting GHz values to MIPS (2.9 GHz $\rightarrow$ 11600 MIPS). We design experiments using cluster sizes of 12, 25 and 40 VMs each for three load levels of 4x, 8x and 16x requests per second, respectively. We use fixed time durations of 500 ms and 20 ms respectively as the container set-up delay and function scheduling delay in all the simulations. Variations and the impact of container start-up delay is not considered a part of this study. The VM CPU utilization threshold (v^{util_T}) is kept at 85% referring [153] and the threshold for function eviction (r^{evict_T}) is maintained at 20% in our experiments.

Experimental Workloads: We employ a number of synthetic and real-world traces to evaluate our proposed algorithms. The synthetic workloads enable us to observe the behavior of the system, while maintaining a constant request arrival rate at a time, and varying the rates across workloads. Under both the real and synthetic workloads, a request received at the controller consists of the id of the application to be invoked, the level of priority requested for the application execution, the requested container memory size and a user specified deadline parameter associated with each priority level. The deadline is the incremental value derived by increasing the average execution time

of each application by a certain percentage. We use two levels of priority as high and low in our experiments, where the high priority requests are associated with a tighter deadline and the low priority requests with a more relaxed deadline.

We create a workload with real-world arrival patterns using trace snippets from Wikipedia [154] and Azure function traces [46]. We extract the set of single function applications from the Azure data set, and refer to the attributes of average container memory size and average execution times (we only consider applications with execution times exceeding 1 second in this study), coupled with the fluctuation of request arrival patterns from Wikipedia traces. We used the arrival patterns from Wikipedia traces since the Azure data set does not contain details of request arrival times. These traces drive the load for 140 application types with a peak load of 16x requests per second, and we run the experiments spanning for a period of one hour, using a cluster of 25 VMs.

Each synthetic workload consists of four synthetic traces which are created with requests arriving for four application types and run in parallel. Average application execution times and requested container memory sizes are generated randomly to be in the range of 1-50 seconds and between 128 - 512 MB in 64 MB increments, respectively. We conduct a series of experiments with multiple synthetic workloads created by varying the request deadline percentages for each priority level, and the arrival rates. In each workload, the inter arrival time of the function invocation requests is modeled using Poisson distribution as in [147], adjusting the Poisson mean to demonstrate different application load levels. A set of experiments are carried out for each workload, adjusting the different system parameters of the task deadline checkpoints (d_{check}) and cpu-quota increment values. At each experiment, we run the workload for a period of approximately 5 minutes in the simulation environment described above.

Performance Metrics: In all the experiments, we observe the performance metrics mentioned below.

1. Total VM uptime during the simulation time. Since we are considering a homogeneous resource environment in our experiments, VM uptime is used as a proxy for the function execution cost efficiency to the provider - we measure the intermittent VM uptimes and add them for all the VMs in the cluster.

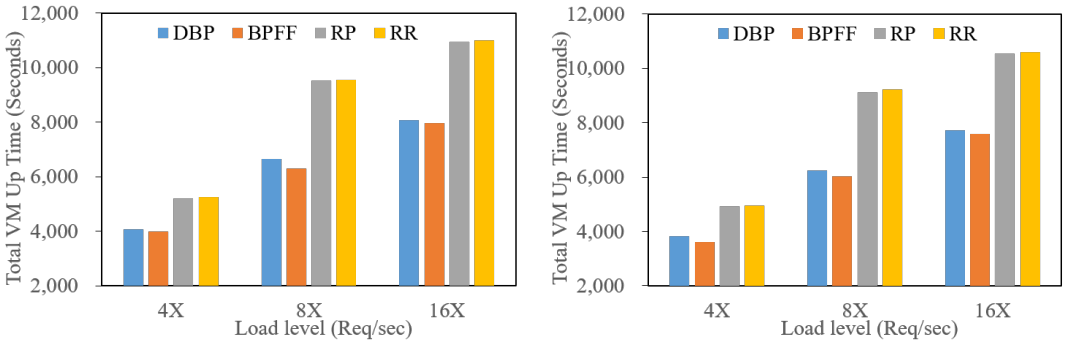


Figure 3.2: VM uptime comparison for the different load balancing algorithms when requests have tighter deadlines at both priority levels (a) d_{check} at 65% and 85%, cpu-quota increment at 20% and 40% (b) d_{check} at 55% and 75%, cpu-quota increment at 40% and 60%.

2. Percentage of requests meeting the deadline - The number of requests finishing on or before the specified deadline as a percentage of the total number of requests.

3.5.3 Results and Analysis

We carry out performance evaluation in two steps for each of the experimental scenarios.

1. We run the experiments with our DBP algorithm (Algorithm 1) for the initial placement of functions and the DRA algorithm (Algorithm 2) for dynamically managing CPU resource allocations. The results are compared with the baseline schedulers: RR, RP and BPFF for load balancing, also coupled with the DRA algorithm for dynamic resource management.

2. The performance of our DBP algorithm for load balancing accompanied with the fine-grained DRA policy is evaluated with a Fixed Resource Allocation (FRA) policy and a cpu-shares policy similar to that adopted by OpenWhisk (OW).

We now discuss the results, primarily in terms of the efficiency in consuming the cloud resources and the level of satisfying the SLA requirements (meeting the function deadlines in this case) of the user.

Evaluation of resource efficiency: The efficient use of cloud resources is evaluated in terms of the total uptime of all the VMs used in each of the scenarios with different load

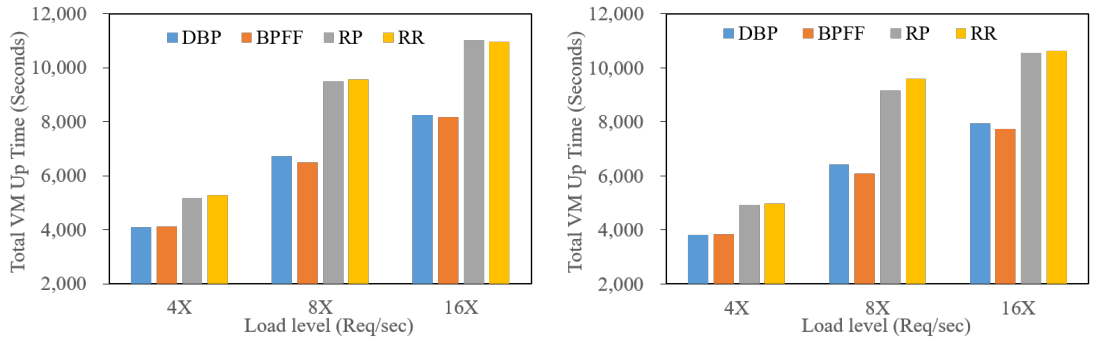


Figure 3.3: VM uptime comparison for the different load balancing algorithms when requests have relaxed deadlines at both priority levels (a) d_{check} at 65% and 85%, cpu-quota increment at 20% and 40% (b) d_{check} at 55% and 75%, cpu-quota increment at 40% and 60%.

levels of the incoming requests. The total time a cloud provider dedicates its resources for serverless function scheduling could be directly related to the cost incurred by the provider as means of the revenue lost during the same period by rendering other services using those resources. Figure 3.2 and Figure 3.3 present results of the resource efficiency study done using the synthetic workloads. Here we compare the VM uptimes using the DBP algorithm with the baseline schedulers under different scenarios, with dynamic provisioning of CPU resources and re-scheduling. Figure 3.2 depicts results under different load conditions when the incoming function execution requests have tighter deadlines (an increment of 5% and 15% over the average execution time for high and low priority requests respectively), while Figure 3.3 shows results for the same workload with relatively relaxed deadlines (an increment of 10% and 20% over the average execution time for high and low priority requests respectively). The results show that the DBP method is able to achieve a high resource efficiency similar to the BPFF heuristic, while the RP and RR schedulers show significantly higher resource usage levels and hence, lesser efficiency. We do experiments varying the time point of CPU re-provisioning (d_{check}) for the high and low priority requests, from 65% and 85% of remaining time to deadline (Figure 3.2(a)) to 55% and 75% (Figure 3.2(b)). We also incorporate two levels of CPU re-provisioning, changing the incremental cpu-quota/cpu-

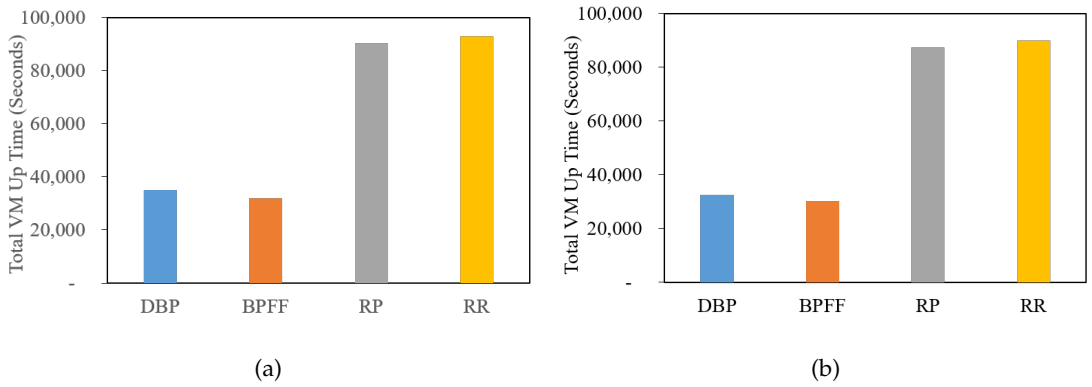


Figure 3.4: VM uptime comparison for the different load balancing algorithms using real-world traces (a) d_{check} at 65% and 85%, cpu-quota increment at 20% and 40% (b) d_{check} at 55% and 75%, cpu-quota increment at 40% and 60%.

period value for each of the priority levels from 20% and 40% to 40% and 60% of CPU time of a vCPU core, for the same two scenarios. Results show that as we vary these system parameters to identify resource contentions faster and resort to better resource alterations (earlier checks for CPU re-provisioning and higher CPU quota increments), the overall VM uptimes decrease slightly, yielding better results. It is noticeable that the longer a varying load level prevails, the higher the distinction of resource efficiency between a random or a spread placement method, as compared to an efficient bin-packing method. This is further emphasized by the experimental results from real-world traces, shown in Figure 3.4, where the VM uptimes recorded when using RR and RP algorithms are approximately 3 times that from DBP and BPFF algorithms.

Evaluation of deadline requirements: Evaluation of application/user SLA requirements is done by taking the percentage of functions meeting the set deadline. The results discussed here are from the same set of experiments described in the above section, for both the synthetic and real-world traces. As shown in Figure 3.5 and Figure 3.6, in all the scenarios with dynamic resource provisioning and re-scheduling, it is evident that the RP and RR load balancing algorithms are able to show higher levels of meeting deadlines. This is because they have a better opportunity of dynamically provisioning CPU resources to functions approaching their deadlines as required, since the initial function placement tends to happen in VMs with more free resources. Despite

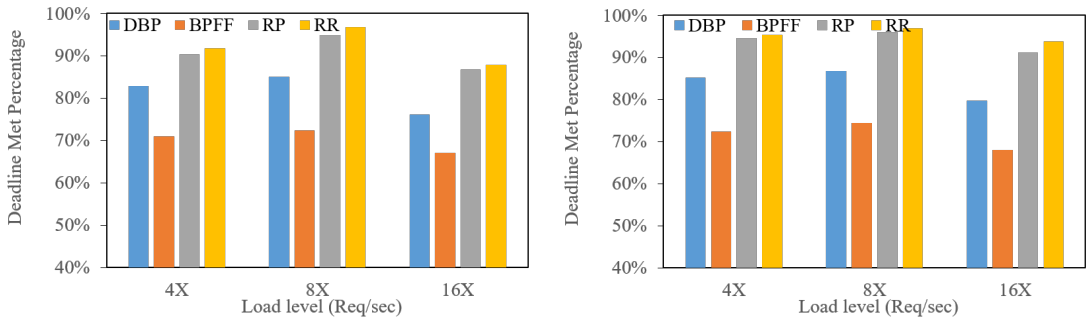


Figure 3.5: Comparison of the percentage of requests meeting the deadline for the different load balancing algorithms when requests have tighter deadlines at both priority levels (a) d_{check} at 65% and 85%, cpu-quota increment at 20% and 40% (b) d_{check} at 55% and 75%, cpu-quota increment at 40% and 60%.

having relatively lower deadline met percentages, the DBP method is able to maintain a low level of deadline violations while also reducing the provider cost by the efficient use of cloud resources as discussed in the previous section. This is because the DBP algorithm considers the deadline priority level of the request in choosing a VM with either higher or lower free CPU quota levels. In general, the BPFF heuristic shows poor performance with higher deadline violations since it always tries to pack the function executions to a minimum number of VMs and hence show lesser flexibility in the ability to face resource contentions in the runtime. When compared with BPFF, DBP performs better, when function deadlines are tighter as well. As the load level increases to 16 requests/second, a slight increase in deadline violations is seen in all the scenarios. The results also show that dynamic resource provisioning and re-scheduling is better able to improve SLA violations when the VM CPU contentions are addressed early and with higher CPU quota increments (Figure 3.5(a) Vs. 3.5(b) and Figure 3.6(a) Vs. 3.6(b)). Figure 3.7 shows results from the workload created from real-world traces. The ability of DBP algorithm to maintain a higher level of deadline satisfaction compared to the BPFF algorithm, when the load level varies, is clearly observed here.

Figure 3.8 shows the performance of our load balancing approach of DBP with dynamic resource alteration (DRA), compared with a fixed CPU allocation policy (FRA) and a proportional cpu-shares policy (OW), both using BPFF as the load balancing

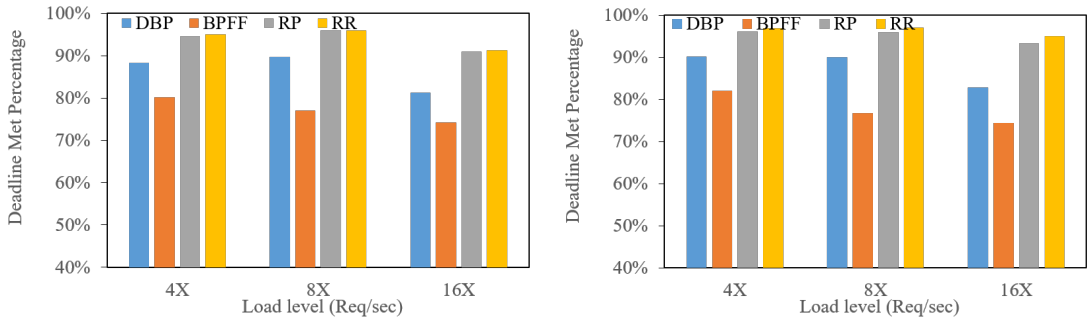


Figure 3.6: Comparison of the percentage of requests meeting the deadline for the different load balancing algorithms when requests have relaxed deadlines at both priority levels (a) d_{check} at 65% and 85%, cpu-quota increment at 20% and 40% (b) d_{check} at 55% and 75%, cpu-quota increment at 40% and 60%.

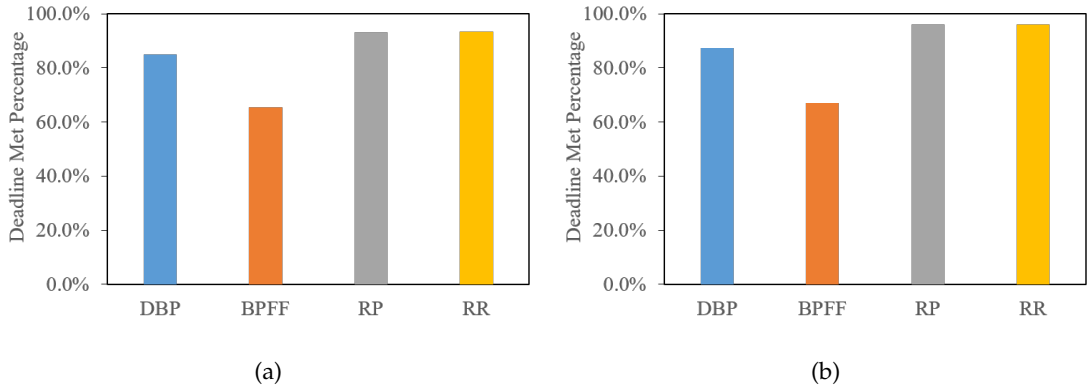


Figure 3.7: Comparison of the percentage of requests meeting the deadline for the different load balancing algorithms using real-world traces (a) d_{check} at 65% and 85%, cpu-quota increment at 20% and 40% (b) d_{check} at 55% and 75%, cpu-quota increment at 40% and 60%.

method. It is seen that sub-optimal initial resource allocations and CPU performance variations in the runtime result in higher deadline violations under the fixed CPU allocation method. Under the cpu-shares policy, the cpu-shares determine the relative weight of CPU power available to each function instance in the presence of CPU contentions [149]. Hence the functions co-located in a VM would largely affect each other's performance and the presence of a function instance with a larger cpu-share could cause a function instance with a relatively smaller share to perform poorly. In comparison,

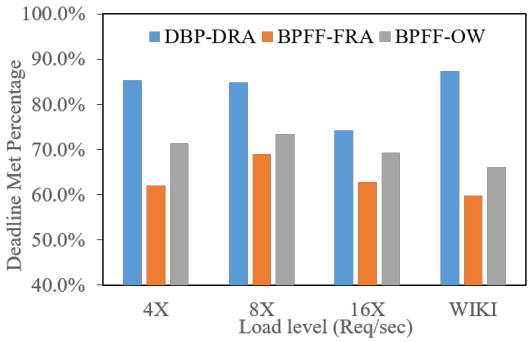


Figure 3.8: Comparison of the percentage of requests meeting the deadline under different resource management methods.

our policy of SLA-aware dynamic handling of the CPU time available to each function, is able to result in better performance under all the load levels.

Performance and cost trade-off: The above experimental results demonstrate that the users are able to meet their required targets of application performance and infrastructure cost by applying the proposed techniques suitably. While the best overall results in both cost and performance is achieved by utilizing the dual techniques of deadline based function placement (DBP) and dynamic resource alteration (DRA), these can be flexibly adopted depending on the most critical requirement. For example, coupling a simple placement algorithm such as RR or RP with DRA would yield the best outcome for a latency critical function, if cost is not a crucial factor. Similarly, for non time sensitive functions, a fixed resource allocation policy with DBP would lead to high resource efficiency and thus lower resource cost. Further, for any function with varying resource requirements with time, the DRA policy could be applied during any point in application execution. The d_{check} parameter and the cpu-quota increment levels are to be adjusted to suit different application scenarios in order to meet target execution times.

3.6 Summary

In this chapter, we explored how dynamic monitoring and managing of resource allocations to function instances and the careful scheduling of function requests on VMs could enable higher opportunities for meeting SLA requirements of the cloud user, while

also resulting in high provider resource efficiency. To this end, we proposed a technique for finer-grained control of provisioned resources to function instances during run time, which could mitigate effects of sub-optimal initial resource provisioning. Further, we presented a function placement algorithm which aggregates running requests on provider infrastructure so as to minimize resource wastage, while also being sensitive to a specific application requirement.

We conducted simulation based experiments on the extended ContainerCloudSim serverless simulation environment to validate the usability of our proposed solution. In the experiments, we compared our function placement approach with baseline scheduling policies and our dynamic resource alteration algorithm with standard resource provisioning techniques. As evidenced by our experiments, the proposed overall solution considerably outperformed existing techniques in terms of application specific SLA satisfaction and resource efficiency.

This chapter presented a technique for dynamically managing the allocated resources to a function request, enabling the satisfaction of application deadline requirements, despite any initial sub-optimal resource allocations. Further we focused on a policy for request scheduling on VMs, resulting in high resource efficiency while also fulfilling user requirements. Serverless systems are multi-tenant systems where applications of multiple users are co-located on the same infrastructure. Moreover, in contrast to the system architecture considered in this chapter, the majority of open-source serverless platforms operate under a system where a single function instance serves multiple concurrent requests. Dynamic workload patterns of multiple users in such an environment could cause significant resource contentions among applications. Thus in the next chapter, we explore techniques for workload and system aware scheduling of function instances in a multi-tenant environment, considering the dual objectives of function performance and provider resource cost efficiency.

Chapter 4

DRL-based Application Scheduling for Multi-tenant Serverless Computing

The dynamic and multi-tenant nature of the serverless workloads and systems, complicates the process of achieving the often conflicting, dual objectives of resource efficiency and function performance. In this chapter we study how a comprehensive understanding on the resource contention among applications along with knowledge on workload and system dynamism, could potentially reach a favorable outcome for both the end users and cloud service providers, during serverless application scheduling. 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 multiple variations of our DRL model depending on the targeted optimization objective, 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.

4.1 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

This chapter is derived from:

- **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "Deep reinforcement learning for application scheduling in resource-constrained, multi-tenant serverless computing environments", *Future Generation Computer Systems (FGCS)*, Volume 143, Pages 277-292, June 2023.

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 [155], [156], [157], 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 [39]. In subsequent research works, a few have studied resource congestion which is a major barrier on achieving the desired performance objectives in serverless systems [63], [62]. 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 [121], [120].

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 [39]. Azure Functions follow a spread placement policy to avoid co-location of concurrent instances of the same function on the same VM [39]. A hash-based first-fit heuristic is employed by IBM OpenWhisk, aimed at a better cache hit rate and instance re-use [61]. 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 [122], [50], 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 [66], [82]. 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 [44] 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 fol-

lows:

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 [30], which is deployed on a Kubernetes [158] cluster composed of heterogeneous VMs.

4. We conduct extensive experiments using real world single and multi-function serverless applications [47], [159] and function traces captured from Microsoft Azure Functions [46], to evaluate the performance and scalability of the proposed DRL model and compare it with baseline schedulers.

The rest of the chapter is organized as follows: Section 4.2 reviews existing related works. Section 4.3 presents the system model and the mathematical formulation of the problem. Section 4.4 introduces the DRL oriented framework for function scheduling, followed by the design details of the agent training environment in section 4.5. Sections 4.6 and 4.7 discuss the performance evaluation of the proposed technique and the potential for future work, respectively.

4.2 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.

4.2.1 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 [122]. 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. [160] presents a locality-aware scheduler to reduce function latencies. A preliminary design for a centralized scheduler is presented in [123], 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 [161]. [64] uses a first-fit heuristic for request load balancing in their serverless setup. A supervised Machine Learning (ML) based approach is presented in [67], 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. [55] also explores a greedy scheduling approach to improve cluster utilization. A heuristic based on function latency in each VM is used in [63] to schedule function requests. A request priority and a deadline based greedy heuristic is proposed in [62] to choose a VM. [44] 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 [162], 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 [163]. An input sensitive container allocation and request scheduling policy is presented in [164], where their focus is mostly on request batching and reordering to minimize SLO (Service Level Objective)

violations.

A hybrid scheduling framework is presented in [50], which uses a greedy algorithm to determine the order and placement of functions in either the private or the public cloud. [165] presents a heuristic approach for scheduling function workflows in a federated serverless environment. Their focus is limited to improving the makespan of function executions.

4.2.2 Application of RL for Serverless Resource Management

A number of works have explored RL techniques for task scheduling in traditional cloud computing environments. [166], [167], [168], [169], [170]. 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 [55]. 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 [66] 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 [82] to dynamically manage resource configurations of each function container. CPU and memory resources from idle functions

Table 4.1: 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					
[122]	✓		Heuristic	✓				✓		
[123]	✓		Heuristic	✓				✓		
[64]	✓		Heuristic	✓					✓	
[67]	✓		ML	✓			✓		✓	
[50]		✓	Heuristic	✓			✓		✓	
[55]		✓	Heuristic		✓			✓		
[63]	✓		Heuristic	✓	✓			✓		
[62]	✓		Heuristic	✓	✓			✓		
[44]		✓	Heuristic	✓				✓		
[173]	✓		DRL	✓			✓	✓		
[161]	✓		Heuristic	✓				✓		
[160]	✓		Heuristic	✓		✓	✓	✓		
[162]	✓		Heuristic	✓	✓	✓		✓		
[165]		✓	Heuristic	✓			✓	✓		✓
[163]	✓		Heuristic	✓	✓	✓		✓		
Our proposed work	✓	✓	DRL	✓	✓	✓	✓		✓	✓

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 [171] to minimize serverless function cold start frequency. A multi-agent Proximal Policy Optimization (PPO) approach is studied in [172] for horizontal and vertical scaling of serverless functions. In [173], 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 4.1 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. For example, some works consider the application resource sensitivities in their scheduling decisions, but not the resource pressure on the serverless platform. Also many existing researches follow simplified models of single function applications and homogeneous VM clusters. Our work in contrast is fo-

cused 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.

4.3 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.

4.3.1 System Model

We formulate our system model around the system architecture of majority of the existing open-source serverless frameworks serving many enterprise users [155], [156], [157]. 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.

Figure 4.1 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 par-

ticular 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

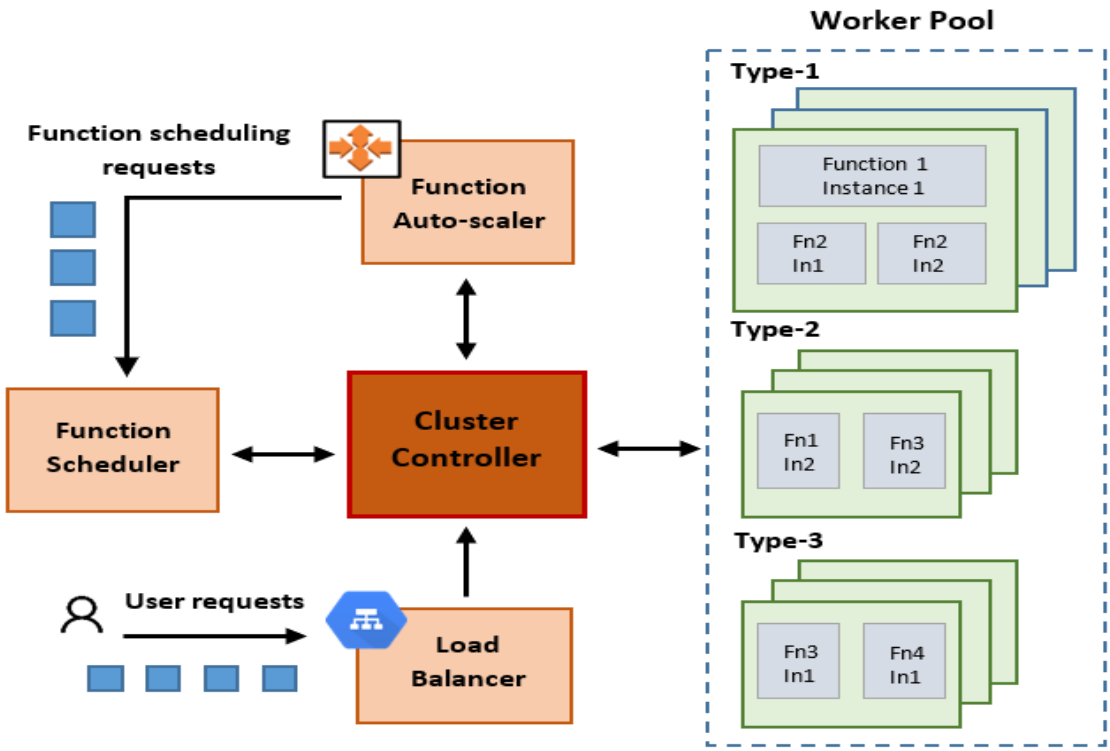


Figure 4.1: The system model of the serverless application scheduling environment.

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.

4.3.2 Problem Formulation

Consider a set $V = \{v_1, v_2, ..., v_N\}$, to be the set of available VMs in a serverless cluster environment, where N is the total number of VMs and v_i, $1 \leq i \leq N$ is the i^{th} 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 $\varepsilon = \{1, 2, 3, ..., Q\}$ to be the index set of all the different functions deployed in the cluster. Let $P^k = \{p_1^k, p_2^k, ..., p_{M_k}^k\}$ be the sequence of pod (function instance) scheduling requests received at the scheduler for the k^{th} function, where $1 \leq k \leq Q$ and M_k is the total number of scheduling requests. Also p_j^k, $1 \leq j \leq M_k$ is the j^{th} 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 [149] 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.

$$p_j^{kc} \leq v_i^C(t) \tag{4.1}$$

$$p_j^{km} \leq v_i^M(t) \tag{4.2}$$

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

$$v_i^C(t) = v_i^c - \sum_{k=1}^{Q} \sum_{j=1}^{M_k} u_{kji}(t) p_j^{kc}(t) \tag{4.3}$$

$$v_i^M(t) = v_i^m - \sum_{k=1}^{Q} \sum_{j=1}^{M_k} u_{kji}(t) p_j^{km}(t) \tag{4.4}$$

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;

$$u_{kji}(t) = \begin{cases} 1, & \text{if pod } p_j^k \text{ is deployed on } v_i \text{ at time } t \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

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 k^{th} 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 $p^k_{Con}(t)$ as follows:

$$p^k_{Con}(t) = \frac{k_r(t)}{k_n(t)} \tag{4.6}$$

Note that $k_r(t)$ above is a resultant of the cumulative arrival rate of requests of all user applications consuming the k^{th} function. The time t in the above expressions: 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6 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 $\gamma = \{1, 2, 3, ..., A\}$ 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 b^{th} application. R^b_q is the total response time of the constituent functions of the q^{th} request of application b, $1 \leq q \leq L^b$. R^b_{q0} 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:

$$Minimize : Sum\ RART = \sum_{b=1}^{A} \frac{1}{L^b} \sum_{q=1}^{L^b} \frac{R^b_q}{R^b_{q0}} \tag{4.7}$$

When calculating R^b_q and R^b_{q0} 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^b_{q0}) 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:

$$Minimize : Cost_{Total} = \sum_{i=1}^{N} price_i \times t_i \tag{4.8}$$

where $price_i$ is the unit price of VM v_i and t_i is the total active time of the i^{th} 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 chapter 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 objectives as follows:

$$Minimize : \beta \times Sum\ RART + (1 - \beta) \times Cost_{Total} \tag{4.9}$$

Table 4.2 summarizes the important notations and descriptions presented in this section.

4.4 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

Table 4.2: 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, $\delta = \{1, 2, ..., N\}$
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, $\varepsilon = \{1, 2, ..., Q\}$
M_k	Total number of instance scheduling requests for a function, $k \in \varepsilon$
p_j^k	j^{th} 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, $\gamma = \{1, 2, ..., A\}$
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 q^{th} request of an application, $b \in \gamma$
R_{q0}^b	Total standard response time of the constituent functions of the q^{th} request of an application, $b \in \gamma$
$price_i$	Unit price of VM, v_i
t_i	Total active time of VM, v_i

elaborate on the specific RL techniques we have incorporated in this work.

4.4.1 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 chapter, 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 k^{th} function on average

9. Sum of disk read and write bytes during a single request execution of k^{th} function on average

10. Request concurrency on each function instance of type p^k calculated using Equation 4.6

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 4.3, 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 Equation 4.7, 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 Equation 4.8.

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:

$$Reward = -(\beta \times (\frac{R_1 - R_{1min}}{R_{1max} - R_{1min}}) + (1 - \beta) \times (\frac{R_2 - R_{2min}}{R_{2max} - R_{min}})) \tag{4.10}$$

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

4.4.2 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.

$$\mathbf{E}[\sum_{t=0}^{\infty} \gamma^t r_t] = \mathbf{E}[r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + ...] \tag{4.11}$$

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:

$$Q^*(s, a) = \mathbf{E}[r + \gamma \max_{a'} Q^*(s', a')] \tag{4.12}$$

Deep Q Learning (DQN): Due to the high-dimensional nature of the environment modelled 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.

Algorithm 3 DQN Based Function Scheduling Algorithm

Initialize the main network parameter θ with random weights
Initialize the target network parameter θ' by copying the weights from the main network
Initialize the N-step buffer D' and replay buffer D
Initialize the training parameters ϵ, α, γ
for episode = 1 to E **do**
    Reset the environment
    for step = 1 to T **do**
        Observe the state s
        Select an action a using the ϵ-greedy policy
        Execute the action a, move to the next state s' and observe the reward r
        Store the transition (s, a, r, s') in the buffer D'
        if size of $D' = N$ **then**
            Translate and move the N-step transition data to D
        Randomly sample a mini-batch of K transitions from D
        Compute the loss $L(\theta)$
        Update the main network:
            $\theta = \theta - \alpha \nabla_\theta L(\theta)$
        Update the target network every P steps
return

$$L(\theta) = \frac{1}{K} \sum_{i=1}^{K} (y_i - \hat{y}_i)^2 \tag{4.13}$$

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 [174]. This technique is summarized in Algorithm 3.

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 4 presents the specific steps of the agent's behavior during online scheduling of function instances in the context of our modelled environment.

4.5 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

Algorithm 4 Online Scheduling

```
upon event Submission of a new pod do
    Enqueue pod in pod-waiting queue
while P do od-waiting queue is not empty
    Dequeue a pod from queue
    Retrieve current cluster state info
    Retrieve function resource requirements and behavioral status
    Compose the state space
    Action a = Agent(state)
    Create pod in the selected worker node
return
```

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 [155], Knative [156], Fission [157]. In this section we discuss the fundamental architectural setup of the designed system.

4.5.1 System Architecture

Figure 4.2 presents the overall architecture of our system. We have deployed this framework using 23 VM nodes on the Melbourne Research Cloud [175] which is part of the ARDC Nectar Research Cloud, the national research cloud of Australia [176]. 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 16GB 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

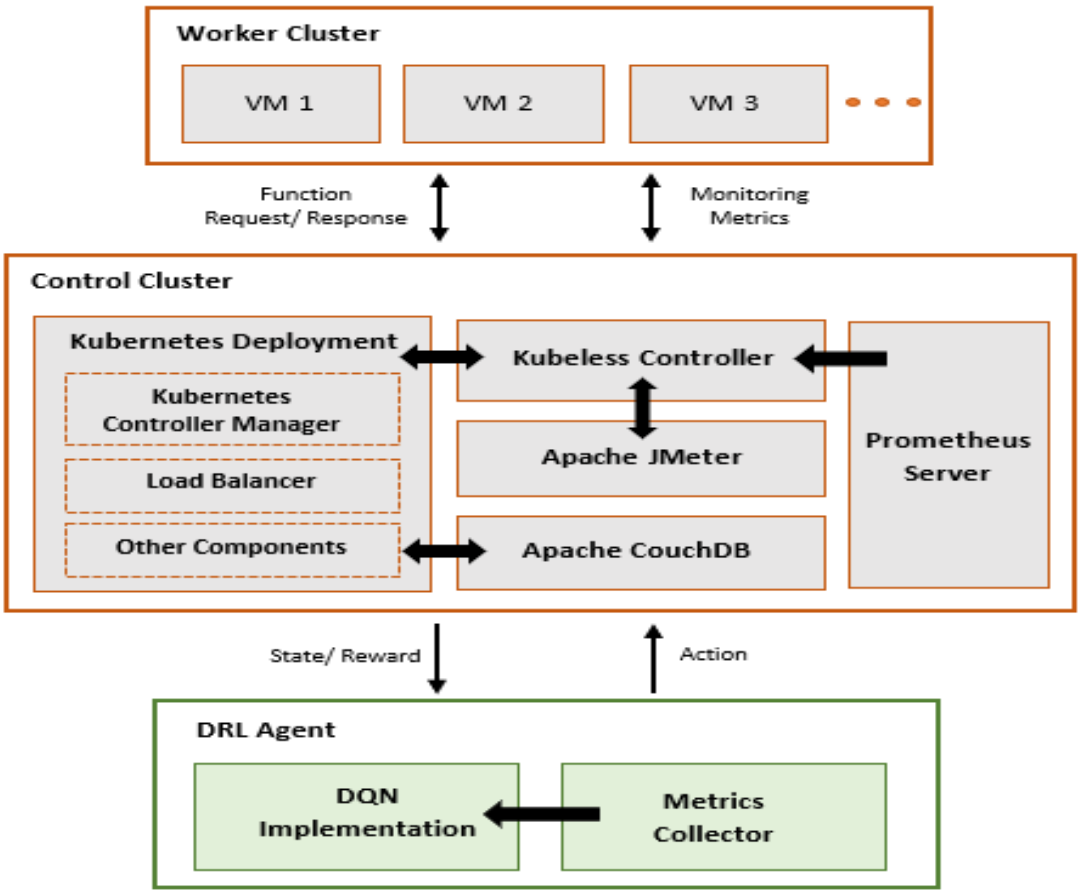


Figure 4.2: The proposed system architecture of the practical testbed for training and evaluating the DRL agent.

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 [177] 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 (30GB each) of the control plane nodes. The Prometheus metrics monitoring tool [178] is installed in our cluster setup, and the Prometheus server is deployed on the second node along with the Kube-

less 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 [179], 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 3 in our framework, is implemented in Python using Keras [32] and Tensorflow2 [31], 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 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.

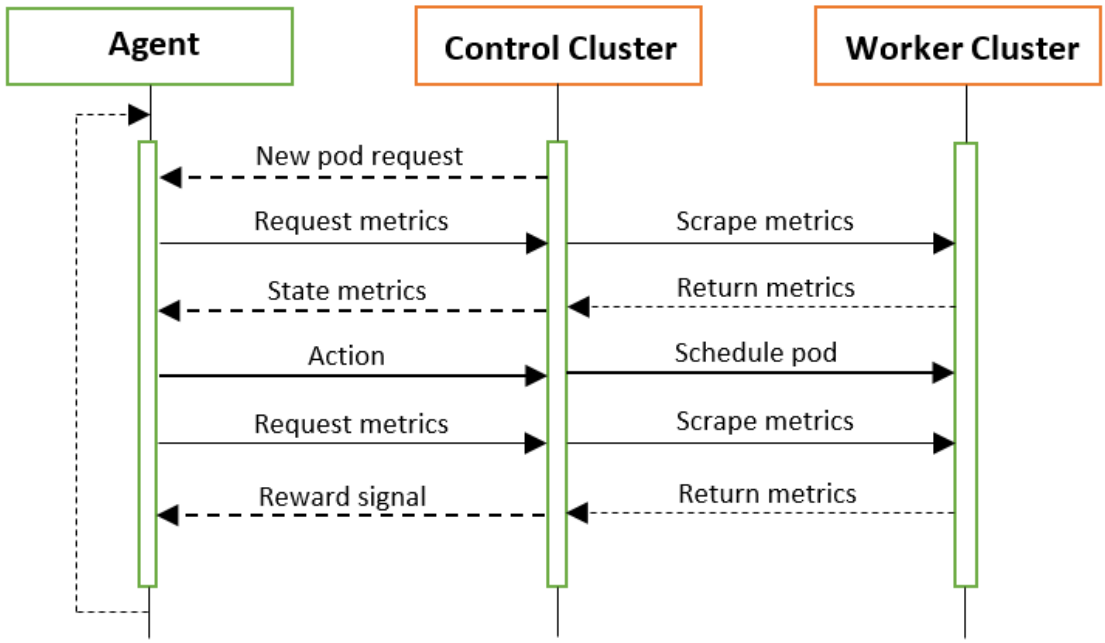


Figure 4.3: The communication process flow of the DRL agent with the cluster during the training phase.

4.5.2 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. Figure 4.3 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.

Table 4.3: Worker Cluster Resource Details.

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

4.6 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.

4.6.1 Experimental Settings

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) [180]. 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 4.3 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.

Workload Specifications

Serverless Applications: We refer to the ServiBench [159] and FunctionBench [47] 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 4.4 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 [46] 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

Table 4.4: 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

running the created workloads multiple times and recording the calculated R_1 and R_2 values at each time step.

Hyper-parameter Configurations

Table 4.5 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.

4.6.2 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 Equation

Table 4.5: 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

4.7. 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 Equation 4.8. 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.

4.6.3 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 [39]. 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

resource efficiency. This is an approach often followed in cluster scheduling scenarios to help resolve congestion [181].

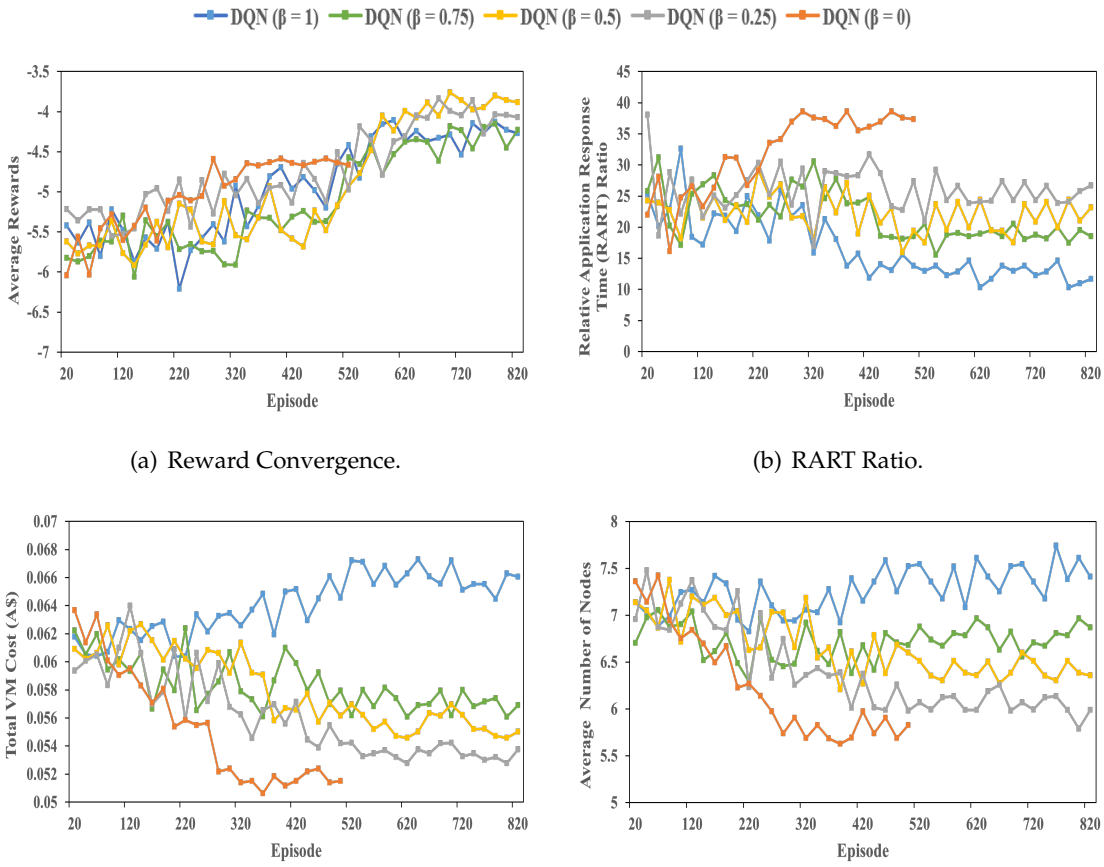
Dynamic Time Cost Aware (DTCA): A scheduler similar to STCA but uses state parameters derived by the DRL agent in a dynamic manner to select a VM. Ranking and scoring of VMs is done as in STCA but the decision of choosing the highest or lowest overall score is taken based on the Relative Function Response Time (RFRT) and p^k_{con} calculated using Equation 4.6, 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 [63] 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 [182] 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.

4.6.4 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 Equation 4.9), 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



(a) Reward Convergence.

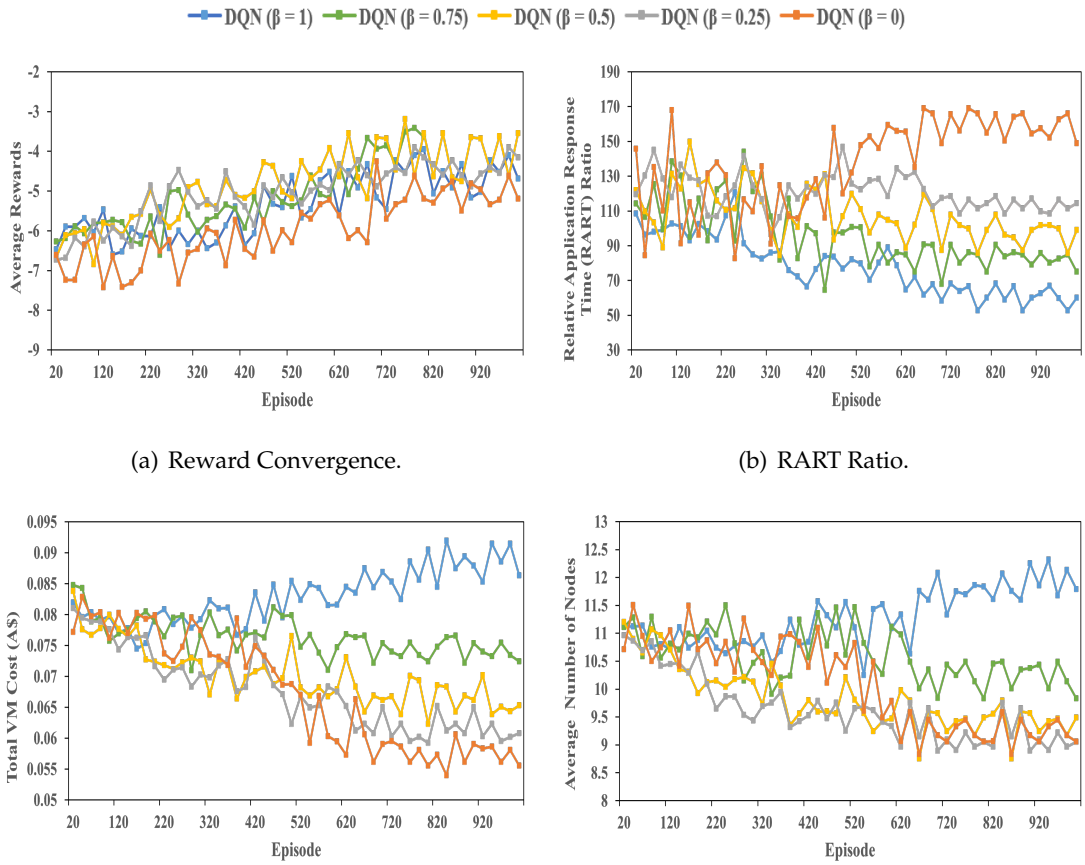
(b) RART Ratio.

(c) Total VM Cost.

(d) Average Number of Nodes.

Figure 4.4: 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.

the awarded reward is solely dependent on function response time while the focus is only on improving VM cost efficiency when $\beta = 0$. Figures 4.4 and 4.5 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.



(a) Reward Convergence.

(b) RART Ratio.

(c) Total VM Cost.

(d) Average Number of Nodes.

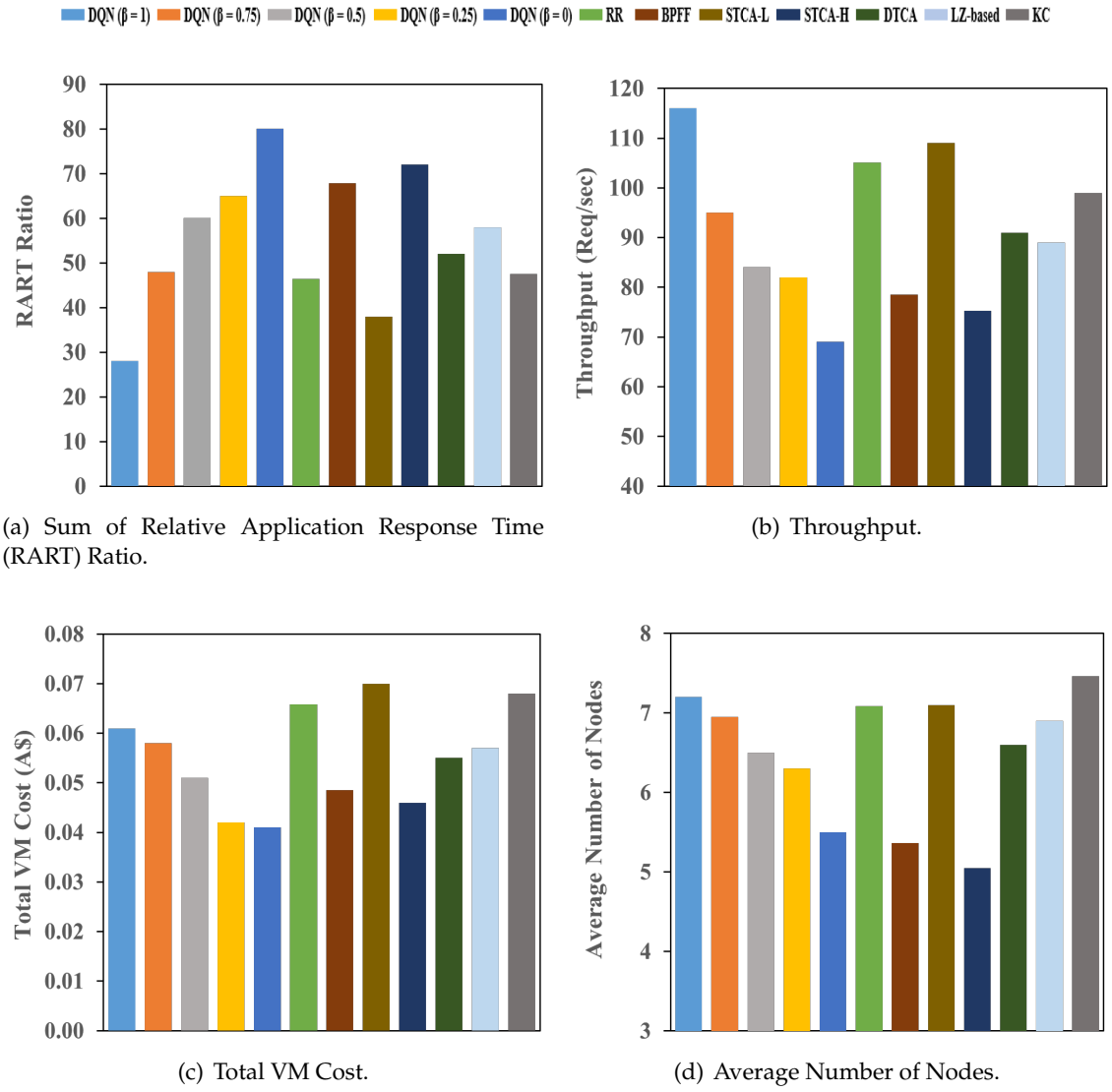
Figure 4.5: Convergence process of the trained DRL models in the 20 VM cluster in terms of reward, RART ratio, total VM cost, and the average node number.

Figures 4.4(a) and 4.5(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 600^{th} episode, and the training took about 60 hours. In the 20 VM cluster, the same model converged around the 800^{th} episode, requiring approximately 80 hours 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 4.4(b) and 4.5(b) as training progresses. In contrast, in the corresponding graphs in 4.4(c), 4.5(c) and 4.4(d), 4.5(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 dif-

ferent 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 (Figures 4.4(c), 4.5(c) and 4.4(d)),4.5(d)) while the RART deteriorates visibly (Figures 4.4(b) and 4.5(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 300^{th} episode which required about 30 hours 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 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 600^{th} 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 4.4(b), 4.4(c), 4.4(d) and 4.5(b), 4.5(c), 4.5(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 $\beta = 0.5$.

4.6.5 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 4.6.1. 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



(a) Sum of Relative Application Response Time (RART) Ratio.

(b) Throughput.

(c) Total VM Cost.

(d) Average Number of Nodes.

Figure 4.6: 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.

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/second overall), for a duration of 5 minutes. All the evaluation parameters in Figures 4.6 and 4.7 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.

Evaluation of application response time

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

10 VM Cluster: Figure 4.6(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 Figure 4.6(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.

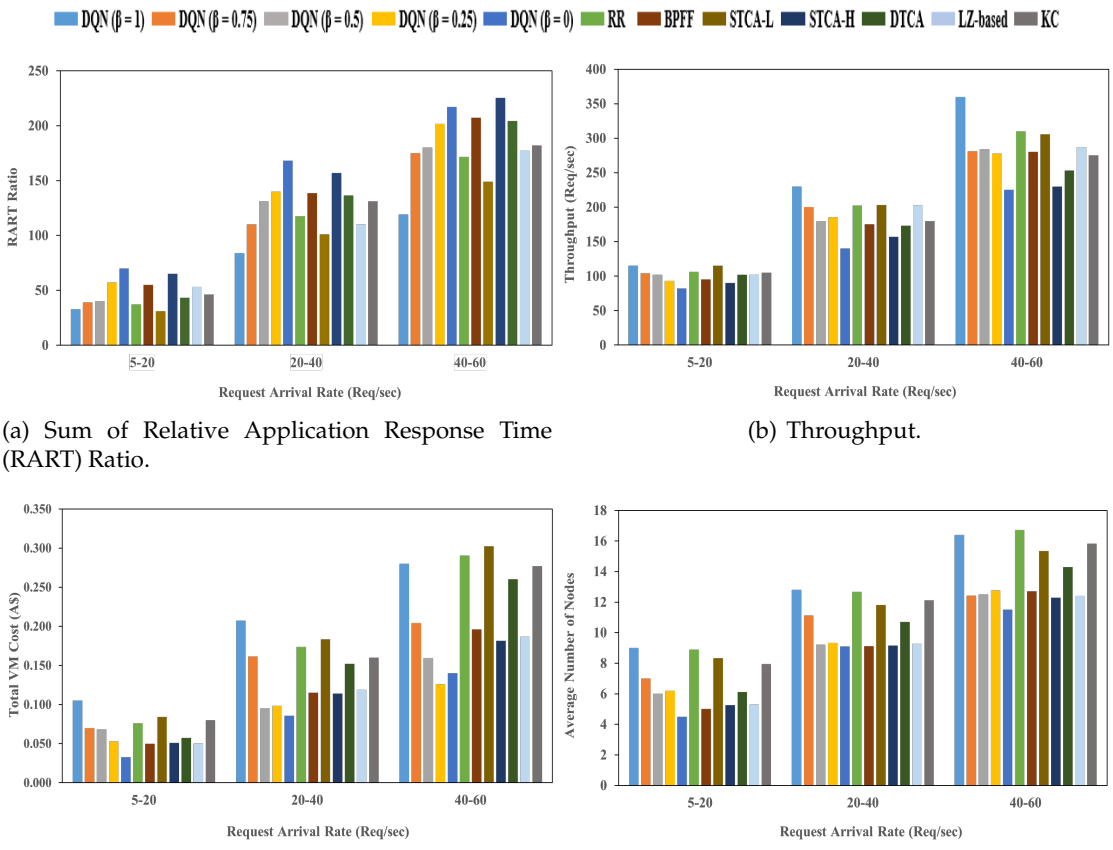
As expected, our DQN ($\beta = 0$) model performs worst in terms of response time since the agent is trained to fully focus on improving resource cost efficiency and thus largely compromises on application performance. This is demonstrated by the RART ratio in Figure 4.6(a) and the throughput graph in Figure 4.6(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 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: Figure 4.7 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 (Figure 4.7(a)).

At the lowest request traffic level of 5-20 req/sec, 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/sec, 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



(a) Sum of Relative Application Response Time (RART) Ratio.

(b) Throughput.

(c) Total VM Cost.

(d) Average Number of Nodes.

Figure 4.7: 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.

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.

Evaluation of resource cost efficiency

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

10 VM Cluster: Figure 4.6(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 Figure 4.6(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 be-

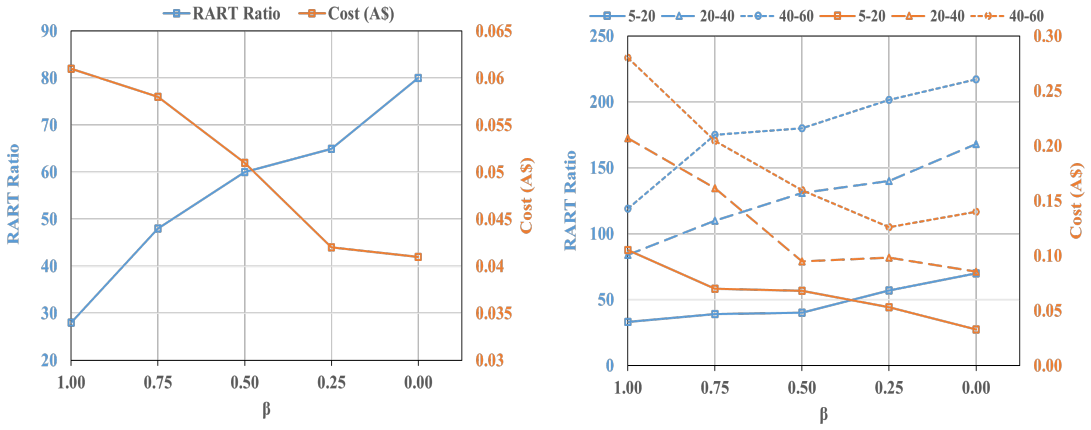
cause its strategy is to use the system parameters to determine high capacity nodes with least number of running functions and minimum resource utilization, and use them for function scheduling. This leads to more cluster nodes often operating drastically below their capacities. The next highest resource 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 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 Figure 4.7(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 (Figure 4.7(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/sec, 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 scheudling 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



(a) 10 VM Cluster.

(b) 20 VM Cluster.

Figure 4.8: The effect of the β parameter in optimizing dual objectives in DRL model training.

nodes, since simply packing them on to fewer VMs has only a limited ability to improve costs.

Evaluation of multiple reward maximization

Figures 4.8(a) and 4.8(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 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.

4.6.6 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 hours 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 hours 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 analyse 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.

4.7 Summary

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.

This chapter proposed an effective technique for scheduling of applications in a multi-tenant serverless environment considering the existing resource contentions and also allowing for a user desired level of optimization between function performance and provider resource cost. In the next chapter, we focus on the autonomic scaling of these deployed applications of multiple users, so as to achieve the same objectives, to the satisfaction of all the parties.

Chapter 5

Time and Cost Optimized Autonomous Scaling of Serverless Applications

Among the many benefits of the serverless computing model, the rapid auto-scaling capability of user applications takes prominence. However, the adhoc scaling of user deployments at function level adds cold start delays and failures in function request executions due to the time consumed for dynamically creating new resources. Existing solutions to address this limitation mostly focus on predicting and understanding function load levels in order to proactively create required resources. Although they improve function performance, the lack of understanding on the overall system characteristics in making these scaling decisions often leads to the sub-optimal usage of system resources. Further, the multi-tenant nature of serverless systems requires a scalable solution adaptable for multiple co-existing applications, a limitation seen in most current solutions. In this chapter, we introduce a novel multi-agent Deep Reinforcement Learning based intelligent solution for both horizontal and vertical scaling of function resources, based on a comprehensive understanding on both function and system requirements. Our solution elevates function performance reducing cold starts, while also offering the flexibility for optimizing resource maintenance cost to the service providers. Experiments conducted considering varying workload scenarios show improvements of up to 23% and 34% in terms of application latency and request failures, while also saving up to 45% in infrastructure cost for the service providers.

This chapter is derived from:

- **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "A Deep Reinforcement Learning based Algorithm for Time and Cost Optimized Scaling of Serverless Applications", *Future Generation Computer Systems (FGCS)* [Under Review, April 2024].

5.1 Introduction

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

A number of existing works have studied the auto-scaling techniques employed by both the commercial and open source serverless computing platforms, and how they affect application performance [183], [184]. [39] compares AWS Lambda, Google Cloud Functions and Microsoft Azure in terms of their function cold start delay. These platforms maintain idle function instances from previous executions for a particular time duration before recycling, in order to have more ready-to-serve warm instances for new executions. AWS and Google seem to have relatively stable cold start delays while Azure platform showed more varying values at the time of their experimentation. Relationships also exist between factors such as the programming language used and the memory size of a function instance and the resulting resource start up delays. The majority of open source serverless frameworks including Fission [157], Kubeless [30], OpenFaas

[155] and Knative [156] are built utilizing Kubernetes [158] as the function orchestrator [185]. The auto-scaling functionality of these frameworks is usually based on a set resource utilization threshold of the existing function instances or the number of requests per second, which determines the required number of function replicas required to meet the current load.

Research works which address the issues related to serverless auto-scaling delays are identified under two categories. One set of solutions is directed towards reducing the frequency of the occurrence of cold start delays, while the other is focused on reducing the measured cold start delay of an individual function instance [184]. In order to reduce the delay itself, various techniques are presented to improve sandbox creation times, including the creation of the required network elements beforehand, utilizing snapshots of previously used containers and designing and developing customized sandbox environments [75], [76], [96]. On the other hand, minimizing the frequency of cold starts is achieved by employing techniques for creating pre-warmed containers, reusing warm containers and adjusting the level of concurrently served requests by a function instance. These approaches often times try to predict the arrival rates and demand levels for individual functions in order to proactively create the required resources [70], [44], [64]. The techniques used for such predictions mostly incorporate the resource consumption characteristics of the serverless functions in order to determine the size of the resource pool to be maintained. They rarely consider the resource availability status or the cost of maintaining such idle resource pools to the serverless resource provider. The distinguished billing model in serverless platforms favours its end users by charging them only when the resources are actively being used with a millisecond level accuracy. This means that even though additional resource pools are maintained to meet Quality of Service (QoS) requirements of the user, the provider is able to recover the costs of such resources only to the extent of them being used, calculated at a very fine level. As such, careful calculations are required considering the status of the platform resources along with function characteristics, in order to make these scaling decisions. Moreover, the majority of solutions are limited in their capability of handling multi-tenancy in the function scaling process.

Considering the above highlighted challenges, our work is focused on carrying out

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

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

1. We formulate and present a RL based model of the function auto-scaling problem in a multi-tenant serverless computing environment.

2. We propose a novel multi-agent function scaling framework based on the policy gradient algorithm Asynchronous Advantage Actor Critic (A3C), which aims to attain a balance in optimizing application performance and provider resource cost. We adapt the A3C algorithm to suit a multi-discrete action space required in making the horizontal and vertical scaling decisions for a multitude of user applications residing in the platform at a time.

3. We train and evaluate our DRL model in a python based simulator environment. We also design a practical testbed based on the open-source serverless platform Kubeless which is deployed on a Kubernetes cluster. The simulator replicates the characteristics and behavior of the practical testbed and utilizes function profiling

data derived from the same, in all its experiments.

4. We evaluate and compare our approach with baseline scaling techniques using real world serverless applications, together with function traces captured from Microsoft Azure Functions.

The rest of the chapter is organized as follows: Section 5.2 reviews relevant literature. Section 5.3 presents the system model and formulates the function scaling problem mathematically. Section 5.4 introduces the proposed DRL oriented scaling framework. Sections 5.5 and 5.6 discuss the design and implementation details of the DRL agent training environment, evaluation of the proposed technique and the scope for future work.

5.2 Related work

5.2.1 Serverless Resource Scaling

Scaling of serverless functions could be discussed in terms of the horizontal and vertical scaling aspects. Horizontal scaling refers to varying the number of instances of a particular function that is available for request execution. As demand levels vary for an application with time, determining the optimum level of replica scaling required to meet the target objectives is a challenging task. [70] try to predict the required number of function instances in order to keep the new request waiting time below a set threshold, by using a heuristic technique. [44] use an exponentially weighted moving average model to estimate request arrival rates. Proactive allocation of sandboxes is done using this estimate. An oversubscribed static resource pool with pre-warmed containers of all resource sizes is proposed in [64]. [71] implement a lightweight middleware which uses the knowledge of function compositions to trigger cold starts, leading to provisioning of new containers before they are required. A container management system with three queues containing cold and warm containers based on their features is introduced in [72]. [74], [186], [187] and [73] propose maintaining a pool of function instances to face request demands. A heuristic solution is given in [188] to adjust the replica num-

ber without compromising on user budget. Time-series forecasting is used in [189] to determine the request workload to support the scaling decisions.

Q-Learning based approaches are used in [77], [171] and [190] to determine the number of function containers to scale-up/down at each point in time in order to maintain low application latency and failure rates. [191] use Q-Learning to decide the optimum level of maximum cpu usage in a function instance to trigger scaling. In [192] the DRL algorithm A2C is used to determine the idle time window for a used function instance and further a time series model is used to predict future invocations and thereby, create warm containers.

Vertical scaling deals with the up/down scale of the resource capacities of a function instance. This is seen as an alternative or used in conjunction with horizontal scaling in order to meet intended targets, in the face of changing traffic levels. An actor critic architecture with Proximal Policy Optimization (PPO) is used in [82] to harvest idle resources from functions and direct them to under-provisioned instances. A Q-Learning based solution is given in [66] to identify the level of concurrency, i.e the number of concurrent requests served per instance, to optimize function latency and system throughput. A DRL based multi-agent (MA) solution is analysed against a single agent implementation in [172] for the horizontal and vertical scaling of functions. They focus on function latency and resource efficiency for users and thereby lack focus on the overall platform resource utilization. A preliminary study is done in [193] on using Q-Learning for horizontal scaling decisions and a heuristic approach for vertical scaling.

5.2.2 RL Solutions for Serverless Resource Management

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

A policy gradient algorithm is proposed in [173], to identify the best node for scheduling a function request. [194] uses a DRL approach to determine the percentage of user requests to be processed by the cloud and offloaded to the fog layer. A distributed

Table 5.1: Summary of Literature Review.

Work	Application Model		Scaling	Scaling Type		Decision Parameters				Multi	VM
	Single Function	Function Chain	Technique	Horizontal	Vertical	Optimization Objective: Response Time	Optimization Objective: Provider Cost Efficiency	Workload Awareness	Overall System Awareness	Tenancy	Heterogeneity
[70]	✓		Heuristic	✓		✓	✓	✓		✓	
[64]	✓		Heuristic	✓		✓					
[186]	✓		Heuristic	✓		✓					
[73]		✓	ML	✓		✓	✓	✓			
[71]		✓	Heuristic	✓		✓		✓			✓
[72]	✓		Heuristic	✓		✓		✓		✓	
[74]		✓	Heuristic	✓		✓	✓	✓			
[77]	✓		Q-Learning	✓		✓					
[171]	✓		Q-Learning	✓		✓					
[44]	✓	✓	Mathematical modelling	✓		✓				✓	
[187]	✓		Mathematical modelling	✓		✓		✓		✓	
[82]	✓		PPO		✓	✓		✓		✓	
[66]	✓		Q-Learning		✓	✓					
[188]	✓		Heuristic	✓		✓		✓		✓	
[189]	✓		Mathematical Modelling	✓		✓		✓			
[190]	✓		Q-Learning/DQN	✓		✓					
[191]	✓		Q-Learning	✓		✓		✓			✓
[192]	✓		A2C/Mathematical Modelling	✓		✓		✓			
[172]	✓		MA-PPO	✓	✓	✓		✓		✓	
[193]	✓		Q-Learning/Heuristic	✓	✓	✓		✓			
Our proposed work	✓	✓	MA-A3C	✓	✓	✓	✓	✓	✓	✓	✓

task scheduling approach is presented in [195] for serverless edge computing networks. They explore a multi-agent dueling Deep Q Learning (DQN) architecture to assist the edge network in making resource allocation and scheduling decisions. A distributed, experience-sharing, function offloading framework for the edge is proposed in [196]. They suggest an improved actor critic algorithm for deciding whether to execute functions on the IoT device or on an edge device. [197] introduce a multi-agent DRL solution for caching packages required for running serverless functions at edge nodes, based on their importance and popularity. They aim to improve per function response time while managing resources consumed while caching. A multi-step DQN based solution is proposed in [198] for function scheduling, in a multi-tenant serverless environment, which aims to optimize application performance as well as provider resource cost.

We summarize the reviewed works specifically in the area of serverless resource scaling in Table 5.1. This comparison considers the aspects of application model, used technique, type of scaling, optimization objective, workload-awareness (consideration for request arrival rate fluctuations), system awareness (knowledge on individual cluster VM resource usage metrics), multi-tenancy (adaptability to suit multiple concurrent applications) and VM heterogeneity.

Majority of existing works propose solutions based on only one aspect of scaling, either horizontal or vertical, which is not ideal for optimizing resource cost. Thus such solutions lack the motivation to be used in their serverless platforms by service providers.

Further, many solutions lack overall system status awareness, and also are not scalable for decision making under a multi-tenant scenario. In contrast, our work captures the dynamic function workload as well as system parameters, with adaptability to suit multi-tenant clusters, and targets optimizing dual objectives concerning both the user and the provider.

5.3 Adaptive Function Scaling

5.3.1 System Model

Our system model represents the common serverless system architecture in use across a majority of open-source serverless frameworks [155], [156], [157]. The main components of this architecture include, a cluster of Virtual Machines (VMs), a load balancer acting as the entry point for user requests, a function auto-scaler, a function scheduler and a controller which coordinates the communication between all the other functional units.

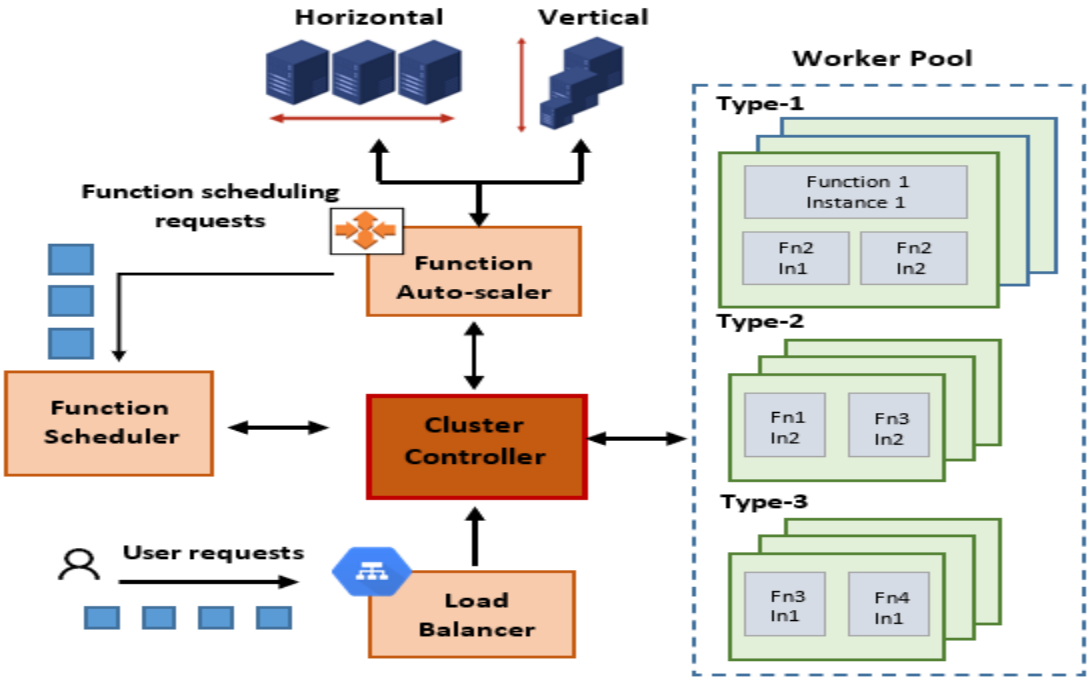


Figure 5.1: The system model of the serverless application execution environment.

The highlevel system model is illustrated in Figure 5.1.

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

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

5.3.2 Problem Formulation

Suppose N is the total number of VMs in a serverless cluster. Each VM is of varying size in terms of its CPU (number of vCPU cores) and memory (MB) capacity. Q is the total number of different function types deployed in the cluster. Let p^k and req^k denote

a pod/instance and a single request of the k^{th} ($k \in [1, Q]$) function respectively, while $M^k(t)$ is the number of existing pods of the same at time t. Each function instance of type k has four attributes at time t, i.e., allocated pod CPU ($p^{kc}(t)$), pod memory ($p^{km}(t)$), CPU and memory consumption of a single request (req^{kc} and req^{km}), standard response time (r_0^k) and the arrival rate of requests of the type. The standard response time for a request refers to the average request response time for a function when executed in a pre-created pod without any resource creation delays.

Compute power is often identified to be a main source of resource pressure in serverless functions, leading to poor application performance [63]. Based on this logic, the default horizontal auto-scaler in our system model triggers a new pod creation and scaling down of existing pods based on a target average CPU utilization of a pod of that type, $T^k_{cpu.util}(t)$. i.e., if the number of new pods of type k to be created is N^k_Δ and the maximum allowed number of pods of any type at any time is M^{max},

$$N^k_\Delta(t) = min[M^k(t) \times \frac{C^k_{cpu.util}(t)}{T^k_{cpu.util}(t)}, M^{max}] - M^k(t) \tag{5.1}$$

where $C^k_{cpu.util}(t)$ refers to the current average pod CPU utilization. Our task in terms of horizontal scaling is to determine the ideal $T^k_{cpu.util}(t)$ value at a given time. Pods which are not currently being used are scaled down as required where $N^k_\Delta(t)$ is a negative value.

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

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

$$\begin{aligned} p^c_{min} < p^{kc}(t) + cpu^k_\Delta(t) < p^c_{max} \\ p^m_{min} < p^{km}(t) + mem^k_\Delta(t) < p^m_{max} \end{aligned} \tag{5.2}$$

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

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

$$[p^{kc}(t) + cpu^{k}_{\Delta}(t)] \times M^{k}_{v_i}(t) < v^{C}_{i}(t) \quad \forall i \in [1, N] \tag{5.3}$$

$$[p^{km}(t) + mem^{k}_{\Delta}(t)] \times M^{k}_{v_i}(t) < v^{M}_{i}(t) \quad \forall i \in [1, N] \tag{5.4}$$

where $M^{k}_{v_i}(t)$ is the number of replicas of k^{th} function residing in v_i, while $v^{C}_{i}(t)$ and $v^{M}_{i}(t)$ is the available CPU and memory of the same at time t.

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

$$p^{kj}_{cpu.util}(t) < [p^{kc}(t) + cpu^{k}_{\Delta}(t)] \quad \forall j \in [1, M^{k}(t)] \tag{5.5}$$

$$p^{kj}_{mem.util}(t) < [p^{km}(t) + cpu^{k}_{\Delta}(t)] \quad \forall j \in [1, M^{k}(t)] \tag{5.6}$$

where $p^{kj}_{cpu.util}$ and $p^{kj}_{mem.util}$ are the cpu and memory utilization levels of the j^{th} pod of function k. The time t in the above expressions: (5.1), (5.2), (5.3), (5.4), (5.5), and (5.6) indicates the time steps in which scaling decisions are taken.

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

latency and request failure rates to be metrics which directly reflect the effects of the platform scaling decisions.

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

$$Average\ RART = \frac{1}{A}\sum_{b=1}^{A}\frac{1}{V^b}\sum_{q=1}^{V^b}\frac{R^b_q}{R^b_{q0}} \tag{5.7}$$

The sum of response times of each function which form the considered execution sequence of an application, is used for calculating R^b_q and R^b_{q0}. The preceding function in a chained application simply evokes the next function in the sequence. Hence this calculation is possible as this process is devoid of any data communication delay. The total standard response time for an application's request (R^b_{q0}) is expressed as a function of q, since the relevant function sequence is dependent on the request input.

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

$$Minimize : [Average\ RART, RFR] \tag{5.8}$$

In this work we also plan to optimize the infrastructure cost of the provider. In the calculation of the resource costs we incorporate VM instance pricing, as we consider a

heterogeneous serverless cluster composed of VMs of different CPU and memory sizes. The provider cost optimization objective is formulated as follows:

$$Minimize : Cost_{Total} = \sum_{i=1}^{N} price_i \times t_i \tag{5.9}$$

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

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

$$\begin{aligned} Minimize : \beta \times Sum\ (Average\ RART + RFR) + \\ (1 - \beta) \times Cost_{Total} \end{aligned} \tag{5.10}$$

Table 5.2 summarizes the various symbols introduced in this section.

5.4 Reinforcement Learning Model

5.4.1 Learning Model for Function Scaling

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

In this work we explore the applicability of the concept of RL in developing an adap-

Table 5.2: Definition of Symbols.

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

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

State space: The state information needs to encapsulate both the resource metrics of the serverless platform infrastructure as well as the resource requirements, traffic levels and the current performance of the function to be scaled. Accordingly, the state in our environment could be presented as a 1-dimensional vector, where the first part describes the cluster VM specifications: [$v_i^{cpu.util}$, $v_i^{mem.util}$, $v_i^{cpu.alloc}$, $v_i^{mem.alloc}$, $v_i^{cpu.cap}$,

$v_i^{mem.cap}$, $v_i^{replicas}$]. $v_i^{cpu.util}$ and $v_i^{mem.util}$ refer to the actual cpu and memory utilization levels of the VMs, $v_i^{cpu.alloc}$ and $v_i^{mem.alloc}$ refer to the percentage of cpu and memory that is allocated to pods, $v_i^{cpu.cap}$ and $v_i^{mem.cap}$ denote the cpu and memory capacities (this is representative of the VM unit prices), while $v_i^{replicas}$ represent the number of replicas of the scaling function, that is currently present in the VM. These VM resource metrics are gathered for all the cluster VMs to form the state space. Note that the resource utilization and allocation levels identify as two separate metrics since although resources are allocated to function pods, the VM resources actually utilized depend on the function requests in execution in those pods. The second part of the state vector is composed of the function specifications: [p^{kc}, p^{km}, req^{kc}, req^{km}, p^k_{rate}, p^k_{RFRT}, p^k_{RFR}, $C^k_{cpu.util}$, $C^k_{mem.util}$]. p^{kc} and p^{km} represent the requested cpu and memory by a function instance, req^{kc} and req^{km} represent the resource consumption of a single request of the type, p^k_{rate} represents the current request rate, p^k_{RFRT} represent the Relative Function Response Time (RFRT), which is the ratio of the actual function response time to the standard response time, p^k_{RFR} represent the function request failure rate, while $C^k_{cpu.util}$ and $C^k_{mem.util}$ represent the average cpu and memory utilization of all the pods of type k. These metrics when consolidated, give the RL agent a comprehensive understanding on the current system status, in order to reach the best scaling policy with time. At the start of each time step, the agent gathers the required data and forms this state vector before determining the scaling action.

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

Reward: The reward assigned to the agent at each step immediately after an action,

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

1. R_1: The sum of the average RFRT and RFR of all the deployed functions in the cluster a set time interval after the implementation of action a_t. Since application response time is a function of that of its constituent functions, RFRT acts as a proxy to measure our performance optimization objective of RART in Equation (5.7). In addition, it is more closely identifiable with each function scaling decision of the DRL agent.

2. R_2: The difference in the total cluster VM up time cost (Equation (5.9)) just before and a set time interval after the implementation of action a_t.

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

$$Reward = -((\beta \times R_{1normalized}) + ((1-\beta) \times R_{2normalized})) \tag{5.11}$$

5.4.2 Actor-Critic based Multi-agent Scaling Framework

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

to overcome many shortcomings of vanilla policy gradient methods. Thus we form the basis of our scaling framework using the actor-critic algorithm.

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

Actor and critic networks learn by either maximizing their objective functions or by minimizing the loss functions. Accordingly, the actor learns the optimal policy by calculating the policy gradient, i.e., the gradient of the network and periodically updating the network parameter θ using gradient ascent (Equation (5.12)).

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta log \pi_\theta(a_t|s_t) A(s,a) \\ \theta &= \theta + \alpha \nabla_\theta J(\theta) \end{aligned} \tag{5.12}$$

where $J(\theta)$ is the objective function which aims to increase the probability of occurrence of the actions which maximize the expected return of a given trajectory. As seen in Equation (5.12) above, we calculate the policy gradient in the actor-critic methods using $A(s,a)$, the advantage function, hence the name Advantage Actor Critic (A2C). Expanding the advantage function;

$$\nabla_\theta J(\theta) = \nabla_\theta log \pi_\theta(a_t|s_t)(r + \gamma V_\phi(s'_t) - V_\phi(s_t)) \tag{5.13}$$

Thus, the advantage function reveals how good action a is compared to the average actions in state s. This essentially helps actor-critic methods to overcome inefficiencies of vanilla policy gradient algorithms by reducing the high variance of policy networks

and stabilizing the model. Similarly, the critic learns by minimizing the loss of the critic network, i.e., the Temporary Difference (TD) error, which is the difference between the target value of the state $(r + \gamma V_\phi(s'_t))$ and the value of the state predicted by the network. During the course of training, the gradient of the critic network is calculated and the network parameter is updated using gradient descent (Equation (5.14)), thus allowing the critic to learn the actual state-value function.

$$\begin{aligned} J(\phi) &= r + \gamma V_\phi(s'_t) - V_\phi(s_t) \\ \phi &= \phi - \alpha \nabla_\phi J(\phi) \end{aligned} \tag{5.14}$$

DRL techniques trained with a single agent have proven to be able to provide effective solutions for many single function scaling scenarios [193], [77], [171]. But serverless platforms are usually multi-tenant environments with a number of deployed functions with various resource characteristics co-existing with each other. Further, these different functions have dynamically changing workload patterns, lowering the sample efficiency of many single agent RL solutions in the context of the multi-tenant scaling problem considered in our work. Thus in this work, we explore the applicability of the DRL technique A3C [199], which employs several DRL agents who engage in learning in parallel, and aggregate the overall experience. The process of parallel learning helps explore the combination of state and action spaces much faster.

In A3C we work with two types of networks, the global network and the local or worker networks. Each worker agent interacts independently with its own copy of the environment, and shares the gathered experiences with the global agent asynchronously. Both the worker agents and the global agent follow an actor-critic architecture. Under A3C, in order to encourage sufficient exploration and reaching a global optimum, we add the term 'entropy' to the previously discussed (Equation (5.13)) actor loss, i.e.;

$$J(\theta) = log\pi_\theta(a_t|s_t)(r + \gamma V_\phi(s'_t) - V_\phi(s_t)) + \beta H(\pi(s)) \tag{5.15}$$

where $H(\pi)$ refers to the entropy of the policy while β controls the significance of

the entropy. As discussed under section 5.4.1, we express our action space for scaling as a novel multi-dimensional discrete action space. We then adapt the technique described for discretized multi-dimensional action spaces in [200], to design our actor network architecture.

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

Algorithm 5 presents the pseudo-code for the multi-agent scaling framework training process flow. We first initialize the global actor and critic network parameters which would be shared among and updated by the worker agents during the training process. Each worker would have its own copy of the environment and separate actor and critic networks (lines 3-6). At the start of each episode, workers reset their local environment. At each time step, the agent retrieves the state information with regard to the platform and the function to be scaled, and feed it to the local actor network. Next, the marginal probability distribution for each action dimension is used to determine the combined action for the current step (lines 11-12). Upon execution of the generated action, the environment transitions to a new state, and the agent receives a reward. All the transition information which includes the environmental state, executed action, awarded reward, and the next state are stored in memory (line 14). If the network update frequency or the maximum step count for an episode is reached, the agent starts the network parameter

Algorithm 5 Actor-Critic based Multi-agent Scaling Algorithm

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

sharing and update process. First the advantage estimates, the loss and the network gradients are calculated for each transition stored in memory (lines 16-18). Then each worker agent asynchronously updates the global actor and critic network parameters using the calculated gradients. Finally, the local networks are updated with new weights pulled from the global model. After each network update, the local memory is cleared (lines 19-21).

5.5 Performance evaluation

5.5.1 RL Environment Design and Implementation

We implement a practical experimental serverless framework on the Melbourne Research Cloud [175] for preliminary data collection related to profiling serverless functions and also for deriving realistic system parameters exhibited during function executions. The testbed comprises of the Kubeless [30] open source serverless framework deployed on a 20 node Kubernetes [158] cluster on top of which the Prometheus [178] monitoring tool is installed for monitoring cluster metrics.

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

1. Requests arriving at deployed function instances are loaded to an event queue in the order of arrival at the start of the simulation.

2. In the event that a suitable function instance is unavailable to accommodate an incoming request, the request is queued and subsequently dropped, after multiple scheduling retries at set time intervals.

3. Time steps for scaling decision making for each agent are scheduled at regular time intervals so that the agent's learned policy is capable of supporting proactive scaling of function resources independent of any workload specifics.

4. At each time step of the DRL agent, the serverless environment exposes the cluster state metrics which include the VM resource usage statistics and the workload nature of the function to be scaled.

5. After the execution of each of the combined horizontal and vertical scaling actions, the agent waits for a set time duration for the environment to reflect the action consequences, before deriving the step reward.

6. Each agent in the implemented multi-agent model, follows these steps in parallel, on copies of the same cluster environment.

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

5.5.2 Experimental Settings

Cluster Setup

Our simulated VM cluster comprises of 20 heterogeneous VMs of 4 different vCPU and memory configurations. The clock speeds of the CPU cores were set to be similar to that of VMs in AWS Lambda serverless platform as identified in [39]. We use the AWS instance pricing of EC2 VMs (in Australia) [180] closely matching the clock speed, vCPU and RAM configurations, as our pricing model. These cluster resource details are summarized in Table 5.3. Our practical testbed too follows these VM cpu and memory configurations. We conduct our experiments under two scenarios, letting the multi-agent model to be comprised of 3 and 5 parallel actor-learners (agents) under each scenario, in order to observe the training time and data efficiency in state and action space exploration with more agents. Each individual agent works in a cluster environment of similar configuration as above during training.

Workload Specifications

Serverless Applications: We choose 12 benchmark applications from ServiBench [159] and FunctionBench [47] benchmark suites, which are formed of either a single or a chain of functions. Each of these applications have varying demands on CPU and

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

Table 5.3: Worker Cluster Resource Details.

Instance Type	vCPU cores	Memory(GB)	Quantity	Price($/hr)
m6g.medium	1	4	5	0.048
t4g.large	2	8	5	0.0848
t4g.xlarge	4	16	5	0.1696
t4g.2xlarge	8	32	5	0.3392

memory resources based on their constituent functions. Thus their diverse sensitivities to different horizontal and vertical actions in the action space provide a good learning experience for the DRL agents. We use our practical setup for conducting function profiling for all the selected applications. An instance of each individual function is deployed on a VM in isolation and the JMeter [179] load generation tool is used for sending a series of user requests to this instance. The results obtained from this tool and the cluster data recorded by Prometheus are averaged across multiple such workload executions to determine the resource consumption of a single function request ($req^{kc}(t) and req^{km}(t)$), standard response time (r_0^k) of a request and the instance creation time. p^{kc} and p^{km} for each function is initially set as the resources required to handle a defined number of requests. Table 5.4 captures the nature of these benchmark applications.

Workload Creation: We leverage function traces from the publicly available data set from Microsoft Azure's Serverless Platform [46] in order to derive request arrival patterns when creating the function workloads for both training and evaluating the DRL agents. In all the experiments, we maintain request arrival rates at 10-60 requests per second, maximum compute power and memory allocated to a single function instance at 1 vCPU core and 3GB respectively and the execution time of a request below 10 seconds. Accordingly, we analyse the Azure function data collected over a 24 hour period and filter a set of multi and single function applications with these characteristics and extract their request arrival patterns in the workload creation. For each function the per minute request arrival rates recorded in Azure traces for a given function is considered

as a per second rate. In a given workload, the function arrival rates for a single function is fluctuated over time using these traces. Multiple such function request loads are combined to form a single workload. A workload consumed by a single agent is incorporated with traffic from no more than 4 functions at a time in order to maintain a sufficient load in the cluster for the training process. During each time step, the function subjected to scaling configuration changes is decided arbitrarily during the training process while the function with the highest RFRT is chosen during model evaluation, in order to attain optimum application performance.

Table 5.4: Serverless Application Details.

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

Table 5.5: Hyper-parameters Used for DRL Model Training.

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

Hyper-parameter Configurations

Hyper-parameters for the actor critic networks of each worker agent are decided on a trial and error basis. The discount factor is maintained at a lower value since the rapidly fluctuating nature of serverless workloads reduce the relevance of distant rewards towards current actions and thus a higher discount factor would force irrelevant information on the agent hindering the learning process. The learning rate for the actor and critic networks is maintained low enough so as not to cause a gradient blowup or lead to a sub-optimal solution too fast, and high enough so as the model converges with sufficient training. Each action dimension is discretized in to 11 actions as described under section 5.4.2. This was arrived at after a series of initial experiments in order to maintain action space exploration costs at a manageable level while reaching good optimization levels for the target metrics. The maximum number of replicas for a single function at a time was restricted in order to suit the capacity of a 20 VM cluster while an upper limit of CPU scaling threshold was also set to ensure proactive scaling for all functions even at very low traffic levels. The hyper-parameter settings for the A3C agents along with these environmental parameters in use for all the experiments are listed in Table 5.5.

5.5.3 Performance Metrics

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

1. **Average Relative Application Response Time ratio (RART)**: The sum of the average, relative response times of all the applications in a workload during an episode, divided by the number of applications, calculated using Equation (5.7).

2. **Request Failure Rate (RFT)**: The ratio of the number of dropped function requests to the total number of requests received in an episode.

3. **VM Usage Cost**: The cost of maintaining the VMs active during an episode. The calculation of this metric is as in Equation (5.9).

5.5.4 Baseline Scaling Techniques

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

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

Knative: The opensource serverless platform Knative [201] allows users to set a target pod concurrency value, limiting the number of concurrent requests handled by a function instance. Further a target utilization value is set determining the actual percentage of the target that we should meet. Horizontal scaling of functions is triggered in order to maintain the set level of request concurrency and the target utilization (we set these at 4 and 75%).

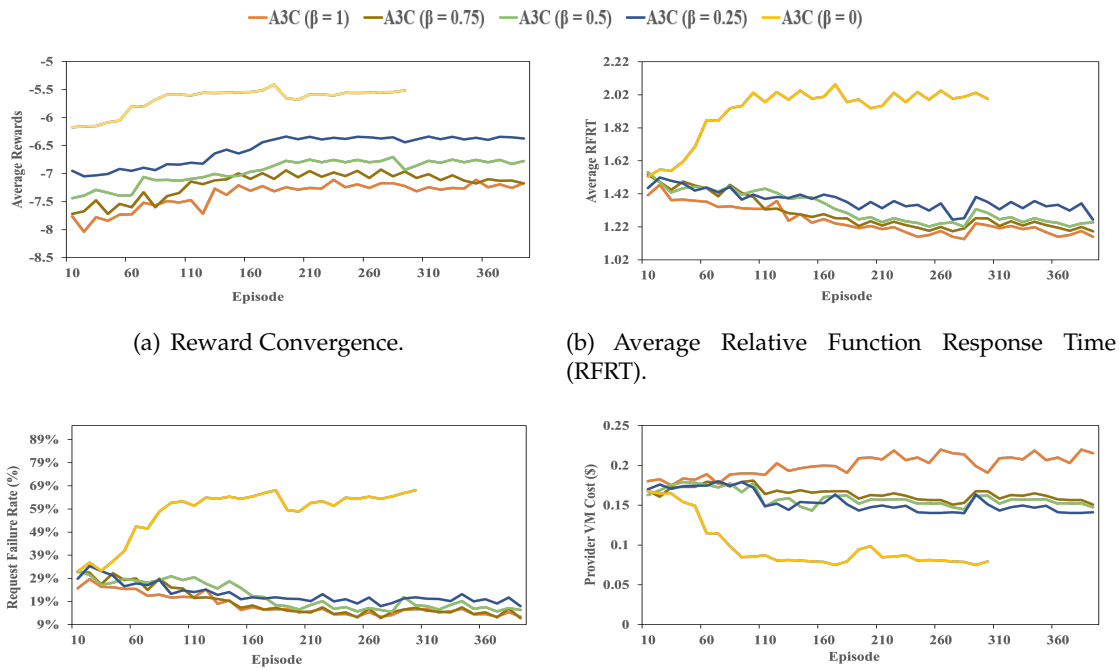
Kube-cpu: Kubernetes default horizontal pod auto-scaler scales function instances based on a set threshold on function level resource usage metrics. Here we consider scaling

function replicas in order to maintain the average CPU utilization across all instances of a function at or below the set value (we set the CPU utilization threshold at 50%).

OpenFaaS: The opensource serverless platform OpenFaas [202] offers a mix of three modes of scaling to be used based on the function requirements. For long-running functions which can handle only a limited number of requests at a time, the 'capacity' mode triggers scaling based on the number of in-flight requests (we consider a threshold set at 4). For functions which execute quickly and have a high throughput, the 'rps' mode enables scaling based on the number of requests per second completed by a function replica (threshold set at 8). All the other workloads which do not support the 'capacity' and 'rps' scaling profiles use the 'cpu' mode which triggers scaling based on the average CPU utilization (set at 50%) across pods, similar to Kube-cpu. We dynamically decide on the scaling mode used for each function type based on their execution times and request rates. Accordingly, functions with execution times greater than 2 seconds are scaled using the 'capacity' mode, functions with less than 2 seconds execution time and request rate higher than 20 requests per second at the time, use 'rps' mode and the rest are scaled using the 'cpu' mode.

5.5.5 Convergence of the DRL Model

Under each of the multi-agent model scenarios comprising of different number of parallel workers, we train 5 variations of the A3C model considering the significance given to each optimization objective. We use the β parameter to signify the priority assigned to each objective in the agent reward structure. Accordingly, $\beta = 1$ refers to a 100% focus on improving function performance while $\beta = 0$ indicates model training leading to infrastructure cost optimization only. The graphs in Figures 5.2 and 5.3 display the training progress achieved over time as the agents gradually learn to optimize the cumulative reward over an episode. The progress is demonstrated in terms of the episodic reward and the target optimization metrics themselves, i.e: average relative function response time (RFRT), request failure rate (RFR) and the VM cost over each episode. Note that the marked values on the graphs are averaged values over 10 episodes for clarity in presentation.



(a) Reward Convergence.

(b) Average Relative Function Response Time (RFRT).

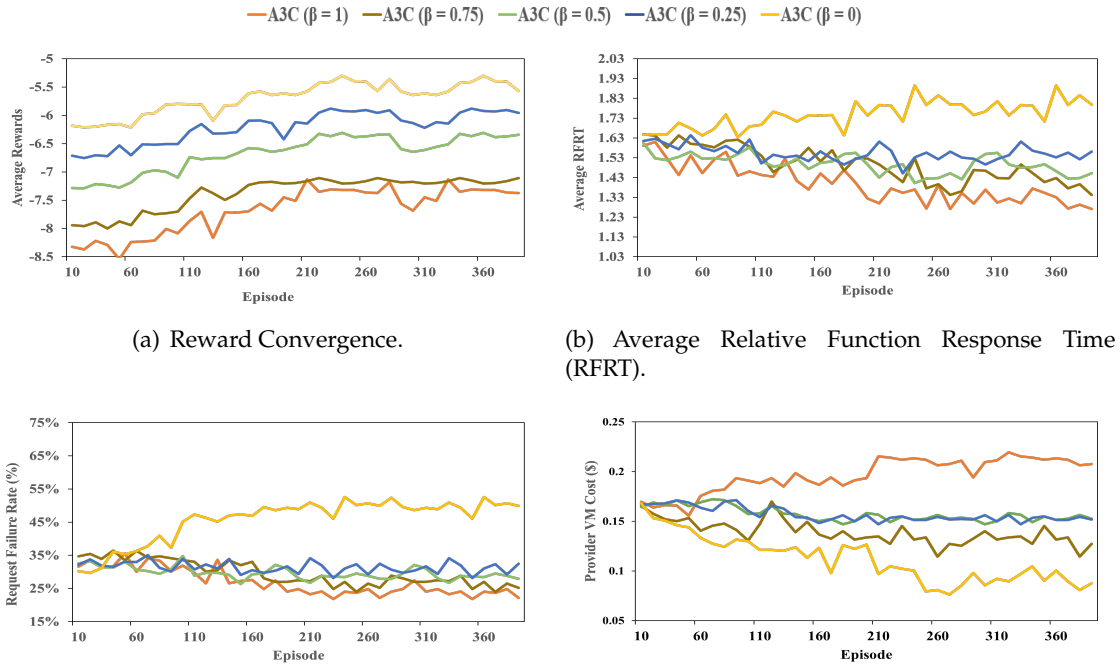
(c) Request Failure Rate (RFR).

(d) Provider VM Cost.

Figure 5.2: Training progress of the 3 worker A3C models in terms of reward, average RFRT, request failure rate, and the total VM cost.

In the first scenario with 3 actor-learners working in parallel, we train the model with 9 different functions, deployed in the cluster in total. These are chosen to be a mix of functions that are common to multiple applications from Table 5.4. In the next scenario, we employee 5 actor-learners in the learning process, in order to analyse the achieved efficiency in state space exploration with more worker agents. In this second set of experiments, we deploy 15 different functions altogether in the cluster and observe the agent behavior leading to model convergence.

Figures 5.2(a) and 5.3(a) show the convergence of the episodic reward under each scenario with varying β parameters. As seen from the graphs, all of the models achieve convergence around the 300^{th} iteration, despite the considerably expanded state space size in the second scenario. This is due to the speed-up in data exploration achieved by having more workers learning in parallel, which results in the global model reaching its maximum optimization levels faster, effectively improving both time and data efficiency. Figure 5.2(a) also shows that when $\beta = 0$, the model convergence happens



(a) Reward Convergence.

(b) Average Relative Function Response Time (RFRT).

(c) Request Failure Rate (RFR).

(d) Provider VM Cost.

Figure 5.3: Training progress of the 5 worker A3C models in terms of reward, average RFRT, request failure rate, and the total VM cost.

relatively faster compared to other scenarios in the set of 3 worker models. Since $\beta = 0$ only incentivizes reducing the VM cost, the agent seems to easily learn to take actions that lead to maintaining the lowest number of replicas possible in the cluster while also limiting their CPU and memory capacities. On the other hand, improving function performance is not as straightforward for the agent to learn, since expanding the pool of function replicas or vertically scaling pod resources would not always lead to the optimum solution. This is because while horizontal scaling creates new resources for request execution, it also adds a resource set up delay which causes increased latency and request failures. Thus a more intelligent strategy needs to be learned depending on the cluster state at each scaling step.

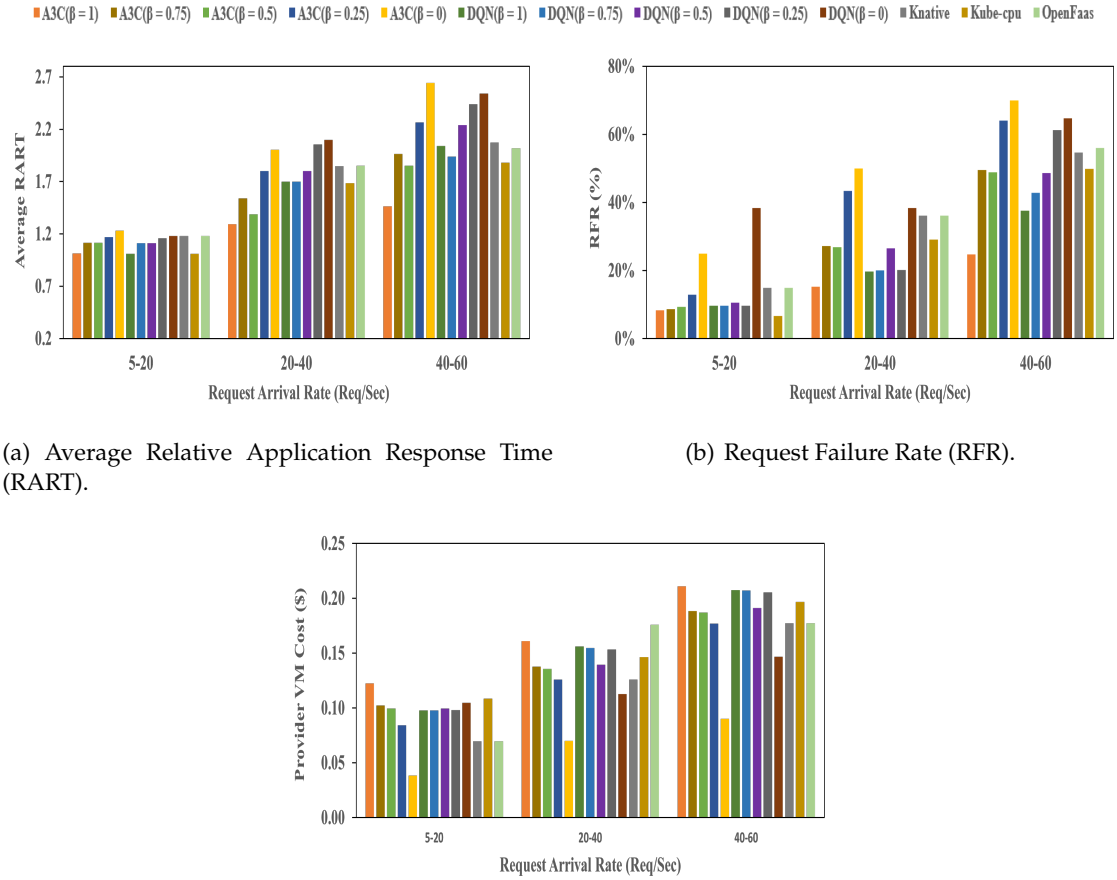
Figures 5.2(b), 5.2(c), 5.2(d) and 5.3(b), 5.3(c), 5.3(d) represent the average function latency, failure rates and the VM costs incurred over the corresponding training episodes. The $\beta = 1$ graphs in both 5.2(b) and 5.3(b) maintain a steady decrease in RFRT with each iteration. The $\beta = 0.75$, $\beta = 0.5$, $\beta = 0.25$ graphs too show a gradual decrease in

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

5.5.6 Analysis of Model Performance on the Evaluation Data Sets

The performance of our trained multi-agent models is evaluated and discussed mainly in terms of our target optimization objectives of serverless application performance and resource cost efficiency. We extract 1800 function traces in total from Azure function traces using the procedure described in section 5.5.2 in creating the evaluation data set. Our model evaluation is conducted under three request traffic levels as 5-20, 20-40 and 40-60 requests per second, for both the 3 and 5 parallel agent scenarios. Accordingly, for both these scenarios, we create 60 workloads each for the 3 load levels, i.e. a total of 360 workloads. When creating each workload for evaluating the variations of the models trained with 3 agents, we include simultaneous user requests from 5 different serverless applications (single and multi-function) created using the 9 functions used during the training process. Similarly workloads are created for the 5 agent models incorporating requests from applications created using 15 different functions. Further, the request arrival rates for a single application are varied over time in a given workload, each of which spans over five minutes.

Figures 5.4 and 5.5 demonstrate the performance of our A3C models against the



(a) Average Relative Application Response Time (RART).

(b) Request Failure Rate (RFR).

(c) Provider VM Cost.

Figure 5.4: Comparison of the Average RART, RFR and provider VM cost in the system during an episode, by the 3 worker A3C models and the baseline algorithms.

baseline scaling techniques under the two agent scenarios. Each bar graph corresponds to the achieved performance metric derived by averaging over the 60 workload runs under each load level.

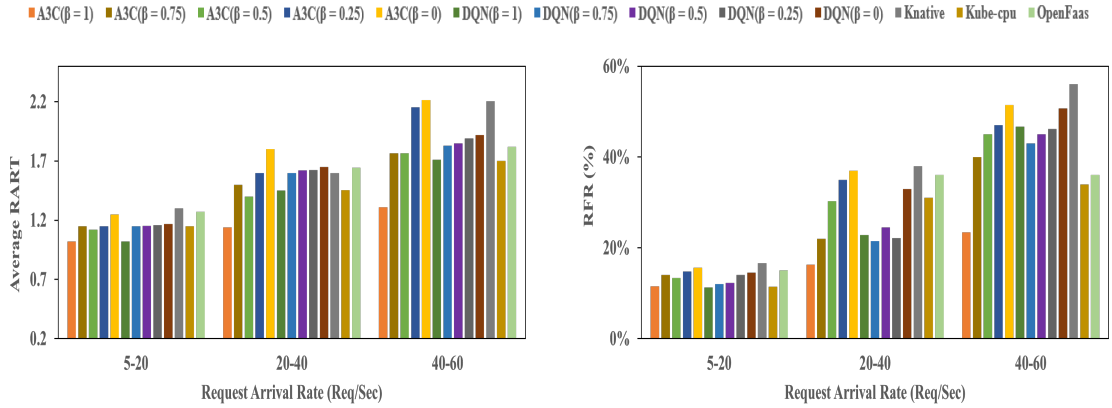
Evaluation of application performance

Application performance is evaluated in terms of RFRT and RFR performance as shown in graphs 5.4(a), 5.5(a) and 5.4(b), 5.5(b). Overall we can see that the latency and request failure rates increase gradually with the rise in request rates due to increased wait times for request executions arising from resource limitations in the cluster. Also, it is evident

from the plotted graphs that the behavior of the models trained using both 3 and 5 actor-learner architectures, is similar in most aspects and thus our discussion below would entail a common analysis for both scenarios for the most part.

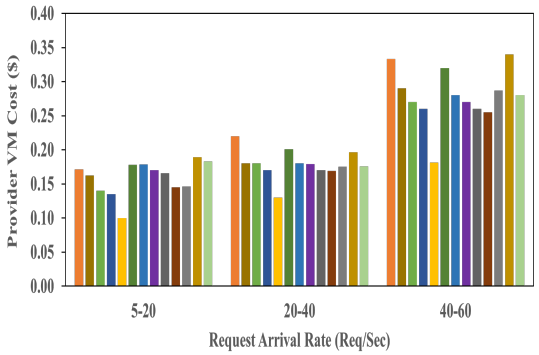
At the lowest traffic level of 5-20 req/sec, we do not observe a significant improvement from the A3C($\beta = 1$) model compared to the rest, where the DQN($\beta = 1$) and Kube-cpu models exhibit almost similar or better performance. This is because, at lower traffic levels, the cluster is less congested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

As request rates increase to 20-40 req/sec, a more distinct performance upgrade is seen to be achieved by the trained models. In both the graphs for RFRT, 5.4(a) and 5.5(a) and for RFR, 5.4(b), 5.5(b), we observe the best performance from the A3C($\beta = 1$) model. The A3C model outperforms the next best performing baseline by up to 23% in RFRT and 24% in RFR. Since our models in this case are purely rewarded for better function performance during the training process, they learn to maintain lower cpu thresholds for function scaling, leading to more proactive instance creation. Further, when an existing instance is reaching its maximum utilization levels, the agent learns to vertically scale its capacity after which it could immediately accommodate more requests without any additional wait times. In this process, the agent also learns to weigh between horizontal and vertical scaling as although vertical scaling expands capacity immediately, it limits future resource expansions. Thus if the current cluster load could sustain some delays in resource creation without excessive request failures, horizontal scaling could lead to long term performance benefits. The DQN($\beta = 1$) model exhibits next best performance as it too follows an intelligent scaling policy in contrast to other baselines. However, the DQN model lacks the fine grained learning capability of the A3C model for many reasons. As a single agent model, it lacks the state space exploration capability even when trained for longer periods of time as we observed during model training. Also, since we had to create compound actions out of the 3 action dimensions with high level discretization, the extensiveness of action space exploration too was far weaker compared to the A3C model. The Knative, Kube-cpu and OpenFaas techniques which follow fixed



(a) Average Relative Application Response Time (RART).

(b) Request Failure Rate (RFR).



(c) Provider VM Cost.

Figure 5.5: Comparison of the Average RART, RFR and provider VM cost in the system during an episode, by the 5 worker A3C models and the baseline algorithms.

threshold base scaling, do not perform well in a congested resource constrained cluster. They apply a blanket threshold for all the application functions facing varying request rates, which lead to increasingly poor performance as the cluster load rises.

At 40-60 req/sec we see even more distinguished performance improvements in the A3C models with high β values, with up to 34% reduction in request failures when $\beta = 1$. We also note that at times, the A3C($\beta = 0.5$) model shows slightly better latency performance than the A3C($\beta = 0.75$) model under high traffic levels. When the agent is rewarded equally to improve both latency and cost ($\beta = 0.5$), it has indirectly resulted in better function latency than when focused more on latency itself. This is because,

scaling actions which lead to efficient cluster resource usage could also result in reduced request wait times and thus latency, which is an added advantage. The DQN model too shows similar behavior in the latency and request failure graphs for DQN($\beta = 1$) and DQN($\beta = 0.75$). The A3C($\beta = 0$) models show worst performance in terms of application performance.

Evaluation of resource cost efficiency

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

In contrast to our observations for latency performance at lower traffic levels, we see clear cost improvements of upto 45% in the A3C($\beta = 0$) models trained for that purpose. This is because with lesser load, if cost is not a concern (i.e. at higher β values), horizontal scaling is encouraged and the cluster could maintain a lot of idling instances. This leads to high VM maintenance costs. On other hand, where resource efficiency is rewarded, the agent learns to take vertical scaling actions more, which leads to higher utilization levels for the active VMs. Subsequently, any idling VMs could be switched off, which saves resource costs. Although not as efficient, the DQN models too show a decreasing trend in cost with β at low load levels, in the second scenario (Figure 5.5(c)), which has higher multi-tenancy in the cluster with more applications. Kube-cpu scaling style triggers proactive horizontal scaling without a deeper understanding on the workload patterns, thus leading to large resource inefficiencies.

At 20-40 and 40-60 load levels too we observe a significant improvement in our A3C($\beta = 0$) model, although the opportunity for gaining a huge resource efficiency level reduces as the cluster utilization levels increase. As expected , the next best performance is seen in the DQN($\beta = 0$) model, as it closely follows the reward structure of the A3C model, falling only short of the state and action space exploration capabilities of the actor-critic architecture. A3C($\beta = 1$), DQN($\beta = 1$) agents exhibit worst performance in terms of cost, closely followed by the Knative, Kube-cpu and OpenFaas techniques which are unable to handle complex load scenarios to achieve a particular target.

5.6 Summary

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

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

While in the previous chapters we explored techniques for efficiently managing the computing resources in serverless computing environments, in the next chapter, we study how we could navigate the deployment of user application workloads in order to reap the highest benefits from serverless platforms.

Chapter 6

DRL-based Application Scheduling in Serverless and Serverful Hybrid Clouds

The serverless eco-system is able to accommodate many application domains successfully. However, some of its inherent properties such as cold start delays and relatively high per unit charges appear as a shortcoming for certain application workloads, when compared to a traditional VM based execution scenario. A few research works exist, that study how serverless computing could be used to mitigate the challenges faced by certain applications traditionally executed in a Virtual Machine (VM) based cluster environment. In contrast, this chapter proposes a generalized framework for determining which workloads are best able to reap benefits of a serverless computing environment. In essence we present a potential hybrid scheduling solution for exploiting the benefits of both a serverless as well as a VM based serverful computing cluster environment. Our proposed framework leverages the actor-critic based deep reinforcement learning architecture coupled with the proximal policy optimization technique, in determining the best scheduling decision for workload executions. Extensive experiments conducted using various application scenarios demonstrate the effectiveness of such a fine-grained hybrid scheduling approach both in terms of the incurred user cost and achieved application performance, with improvements of up to 44% and 11% respectively. .

6.1 Introduction

This chapter is derived from:

- **Anupama Mampage**, Shanika Karunasekera, and Rajkumar Buyya, "Deep Reinforcement Learning for Scheduling Applications in Serverless and Serverful Hybrid Computing Environments", *IEEE Transactions on Services Computing (TSC)* [Under Review, November 2023].

Due to the attractive nature of its eco-system, many new applications nowadays are designed to suit the stateless, short running nature of serverless functions, for easy transition in to this novel computing paradigm. Nevertheless, even when an application in its design is fully compatible with a serverless architecture, the nature of the application workload may render it unsuitable for execution in this environment. A key feature of any serverless platform is its ability to auto-scale the deployed application at a very granular level, with scaling to zero at its extreme. This distinct function level auto-scaling property is established by closely following the demand patterns and scaling up and down function resources just-in-time as required. With such an adhoc scaling policy, the occurrence of certain delays in user request executions due to time taken for resource creation known as cold start delays, is unavoidable. While for some applications, this occasional and rather small delay could be deemed insignificant, a latency sensitive application with a very short runtime could face detrimental effects from such unprecedented delays.

Serverless platforms charge their users only for the resources consumed for application execution, calculated with a millisecond accuracy. To the service provider this may incur a loss at times when load levels tend to be irregular, since their infrastructure would need to stay alive throughout, including the idling periods. In order to compensate for this potential unrecoverable cost, it is observed that the per unit billing rates for serverless services are considerably higher compared to traditional VMs [45]. Thus, if an application sustains a constant high level of traffic, the feasibility of using a serverless deployment need to be studied. However, if the load levels are irregular with sudden bursts of traffic, using serverless functions and paying for only the resource consumed time would be understandably cheaper.

On the other hand, a VM when rented out, could be used for a longer period without facing adhoc resource creation times. As long as the application is having a regular high traffic level to keep the resources busy, the usage of a VM based set up could prove to be much more cost effective compared to a serverless execution. In contrast, if load fluctuations are high with long periods of little or no traffic, maintaining a VM resource would not be as meaningful cost-wise. An on-off mechanism for VMs is also not viable considering the relatively high start up times.

In addition to the initial decision on determining the execution platform, the subsequent decision on choosing a host node on the selected platform too has implications on both time and cost factors for the users. While on the serverless platforms, choosing a host node with warm function instances saves up on request waiting times, careful load balancing on Infrastructure-as-a-Service (IaaS) clusters is directly related to maximizing rented resource utilization and in turn earning cost savings.

Considering the aforementioned facts, it is useful to be able to understand the workload patterns for an application and come up with a suitable schedule for executing user requests, targeting both time and cost effectiveness. A few existing research works have initiated the first steps in this regard, mostly by exploring how serveress computing can be used as an add-on to mitigate various shortcomings of a VM based serverful deployment [203], [93], [45]. The majority of these works target specific application scenarios such as web-services and High Performance Computing (HPC) workloads and try to determine at which point, a switch to a serverless execution would be useful. In our work, we aim to provide a fully generalized intelligent solution for request scheduling, which extracts the best in both a serverless and a serverful infrastructure. Our proposed framework is capable of determining not only the deployment environment, but also the specific resource in that environment that is ideal for running a particular user request considering both application performance and user cost. A fully automated hybrid framework such as this could be a potential offering by cloud service providers which would have many use cases.

Deep Reinforcement Learning (DRL) is very popular among researchers nowadays for solving cloud resource management related problems due to its experience based learning strategy which is proven to be effective in exploring dynamic cloud computing scenarios. Our proposed solution employs an actor-critic architecture enhanced with a hierarchical action space which first determines the ideal deployment environment after which the most suitable cluster node is selected. The key **contributions** of our work are as follows:

1. We formulate the problem of scheduling an application request on a serverless and serverful hybrid cluster environment, based on RL.

2. We propose a novel actor-critic architecture with a hierarchical action space which is capable of decision making at two levels. The mode of deployment is decided in the first level while the node for scheduling within the selected cluster is decided in the second. The DRL agent is trained to reach the best optimization levels in terms of application performance and user cost for a given workload.

3. The DRL agent is modeled to capture application workload as well as the serverless and serverful cluster resource details and behavioral patterns in order to gain a comprehensive understanding on its action environment.

4. We evaluate and compare our approach with baseline scaling techniques using real world applications, together with function traces captured from Microsoft Azure Functions.

The rest of the chapter is organized as follows: Section 6.2 highlights existing relevant literature. Section 6.3 describes our system model and presents the mathematical formulation of our problem. Followed by this, section 6.4 describes the DRL based hybrid scheduling framework. Section 6.5 presents the details of the DRL agent training environment and discusses the performance of our proposed solution. Finally, section 6.6 explains plans for future work.

6.2 Related Work

6.2.1 Serverless and Serverful hybrid scheduling

The concept of utilizing a hybrid serverless and VM based environment for application execution is still at its inception. A few works exist in literature which have attempted to explore this hybrid approach for various use cases.

[45] study how a serverless deployment could help alleviate shortcomings of VM auto-scaling for Machine Learning (ML) inference services. They propose a framework which uses serverless functions whenever VMs with free resources are not available. The required VM instances are then spawned based on either a reactive or predictive scaling policy. Load balancing on VMs is done using a bin-packing method. However,

Table 6.1: Summary of Literature Review.

Work	Decision Level		Technique	Decision Parameters					Generalizability	VM
	Deployment	Scheduling		Optimization Objective		Container & VM Cold Start	Workload	Overall System		Heterogeneity
	Mode	Node		Response Time	User Cost	Awareness	Awareness	Awareness		
[45]	✓		Heuristic/Linear Regression	✓	✓	✓	✓			✓
[204]	✓		Heuristic	✓	✓	✓	✓			
[93]	✓		Queueing Theory/Heuristic	✓	✓	✓	✓		✓	
[205]	✓		ML	✓	✓		✓			
[203]	✓		Heuristic	✓			✓		✓	✓
Our proposed work	✓	✓	DRL(A2C-PPO)	✓	✓	✓	✓	✓	✓	✓

they use dedicated VM clusters for serving different ML models and the heuristics used for execution node selection are not ideal for optimizing user cost. Another approach of using cloud functions for interim processing while VMs are being launched is discussed in [204]. [93] present a system to dynamically switch a micro-service deployment between functions and VMs. They mainly focus on resource contention that could occur among serverless functions and use input from a contention monitor when taking a decision to switch an application load from IaaS based deployment. They handle resource scaling by always directing traffic to one deployment mode and reactively creating the required resources on the other. With highly dynamic workload patterns, this may not result in the ideal usage of resources in both platforms. A time series analysis and a classification algorithm is used in [205] for deciding the best deployment environment for a given time range. An initial study on determining for which workload scenarios, a hybrid deployment approach would be beneficial is conducted in [206]. A solution for leveraging serverless computing for executing HPC workflows is presented in [203]. In order to determine which environment is ideal for running each task, they run everything on VMs and then on a serverless platform. The I/O overhead and the execution time on each platform are taken in to consideration for decision making. Cost efficiency is not included in the scope of their work. They suggest a heuristic for mitigating container cold-start delay, but do not account for VM start up delays.

6.2.2 Serverless Resource Management with RL

RL has been used successfully in a number of research works in recent times for enhancing resource management in serverless computing environments. The provided solutions mostly address function scheduling, scaling problems or used for determining optimum resources allocations for a function.

A multi-step DQN solution and a policy gradient algorithm are discussed in [198] and [173] for determining the ideal cloud based host node for running a serverless function. A number of DRL based frameworks are presented in [194], [196], [195], and [197] for serverless cloud-edge computing environments. Q-Learning solutions are presented in [77], [190] and [192] for horizontally scaling serverless functions. Further, DRL and RL implementations are also proposed for combined horizontal and vertical scaling decision frameworks in [172] and [193].

A summary of related works in literature, which explore the space of serverless and IaaS cluster hybrid scheduling, is provided in Table 6.1. The existing studies are compared in terms of their provided solution scope, used technique, the objectives of optimization, awareness on various system parameters, generalizability (adaptability for multiple application workloads) and consideration for VM heterogeneity in the serverful cluster. While many works discuss the switch between the two deployment modes, their scheduling decisions do not extend to the final place of execution of a function, which could have a considerable impact on overall application and system performance. Existing researches in this area are also confined in their applicability to specific application types. In our proposed solution, we try to overcome these limitations by providing a complete scheduling decision for applications irrespective of their specific resource requirements.

6.3 Hybrid Scheduling

6.3.1 System Model

Our system model is primarily composed of two service clusters, one for serverless deployments and one for a serverful (IaaS) deployment. A global load balancer and a resource manager component handles the forwarding of user requests to one of the clusters for execution. Figure 6.1 illustrates the system model of our proposed hybrid execution engine.

User requests are received at the hybrid load balancer as shown in the diagram. This global load balancer is the decision making body which outputs the best deployment

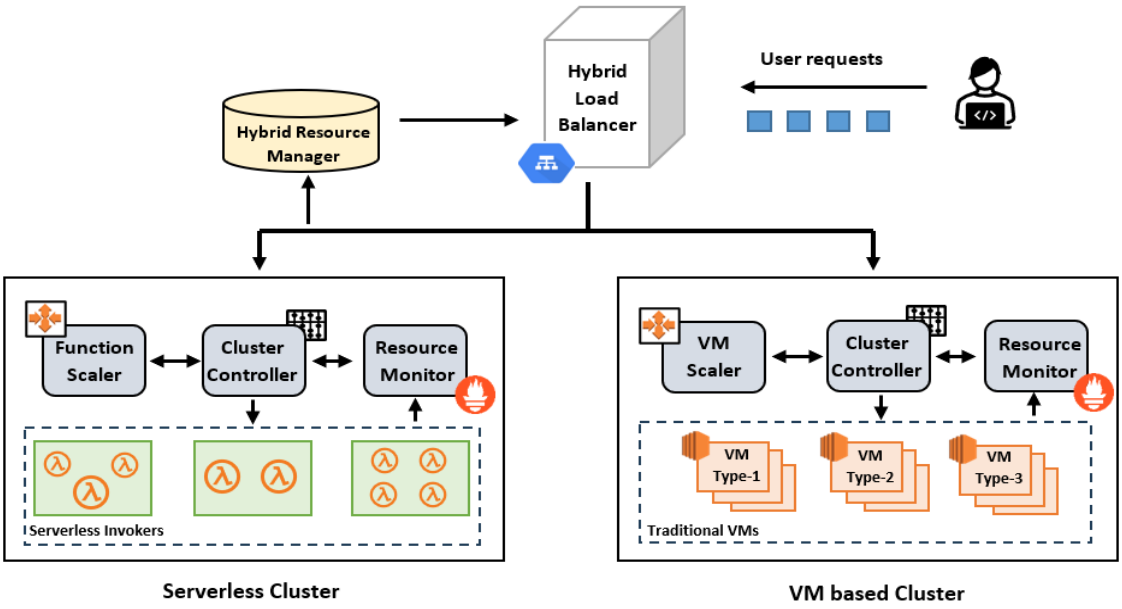


Figure 6.1: The System Model of the Hybrid Application Execution Environment.

mode/environment and also the specific scheduling node for an application request. Articulating the functionality of this novel functional component is the focus of this work. The hybrid resource manager is a database server which periodically derives and updates the resource as well as behavioral metrics of both the service clusters.

The serverless cluster is comprised of a set of invokers which host the containers for function execution. These could be servers or VMs, and referred to as 'nodes' from here onwards in the chapter. The cluster controller acts as the governing body which coordinates the communication and overlooks the actions of all the other functional components. It is also the entry point to the cluster. Any new request that is decided to be deployed in a serverless environment, is received at the controller along with the selected node information for scheduling the same. The resource monitor is a monitoring tool which scrapes cluster metrics including resource details of nodes and function containers and also the performance data of the requests in execution and passes on to the global resource manager. Upon receipt of a new request, the controller checks if the selected node contains warm function instances for the request type. If so, the request is forwarded to an existing ready container. In case such idling instances are not available, the function scaler spawns a new container in the preferred node and deploys

the function code along with any dependencies. Then the request is forwarded to the new instance. This results in a 'cold start' delay for the request. To have more warm instances in order to minimize this delay, the scale-in process allows a set idle time for the resource, once a request finishes its execution. The function executions are charged as per the billing model in commercial serverless platforms, i.e. at a GB-second rate with a millisecond (ms) granularity in addition to a per request rate.

The IaaS cluster is composed of rented VM resources with varying cpu and memory capacities, from a cloud provider under the IaaS model. We consider resources similar to on-demand EC2 VMs from Amazon [207], which are recommended for short term, unpredictable workloads that cannot be interrupted. Users are charged per second of usage derived from the hourly rate. The resource monitor gathers cluster metrics similar to that in the serverless cluster. The VM scaler is equipped with a cpu-threshold based scaling policy following Amazon EC2 auto scaling [208]. Accordingly, VMs are scaled out in order to maintain the average cpu utilization at the given threshold. Once a VM is freed after all the executions, it is scaled-in only after staying idle for a set time duration, so that the frequency of cold starting instances is minimized. We define a maximum cluster size, and maintain a record of the VMs that are active, stopped, and pending creation at a given time. Once a request arrives at the controller with an associated VM id, if the particular resource is already running, the request is forwarded to it. If it is pending creation, the request is queued until the resource comes alive.

6.3.2 Problem Formulation

Consider $N = \{n_1, n_2, ..., n_Q\}$ to be the set of nodes in a serverless computing environment. Each node has a cpu and memory capacity measured in terms of vCPU cores and Mega Bytes (MBs). The unallocated, available cpu and memory values of node n_i, $1 \leq i \leq Q$ at time t is $n^c_{i(t)}$ and $n^m_{i(t)}$ respectively. C_j is the container running the j^{th} ($1 \leq j \leq B$, B being the total number of requests for the function) request of an application/function deployed in the cluster. Following resource demand and capacity constraints need to be satisfied for C_j to be deployed on node n_i at time t.

$$C_j^c \leq n_i^c(t), \;\; C_j^m \leq n_i^m(t) \tag{6.1}$$

where C_j^c and C_j^m identify the requested resources by the container. Similarly, suppose $V = \{v_1, v_2, ..., v_M\}$ represent the cluster of VMs in the IaaS cluster. Any request to be executed in a VM needs to meet its resource availability. Going by the above notation, if R_j is the j^{th} request of an application and $1 \leq l \leq M$,

$$R_j^c \leq v_l^c(t), \;\; R_j^m \leq v_l^m(t) \tag{6.2}$$

A key metric that we target to improve in this work is application performance in terms of execution time latency. In order to not allow request processing time variations in different applications to hinder the overall performance tracking, we consider a relative response time parameter of a request when measuring performance. Relative Request Response Time (RRRT) is defined as the ratio between the standard (R_j^{r0}) and the actual response time (R_j^r) of a request. Standard response time is the time to response when the request is run in an isolated environment on a readily available resource. Accordingly, over the course of an application request workload, we target to minimize the average Relative Request Response Time (RRRT) ratio, i.e.,

$$Minimize : Average\ RRRT = \frac{1}{B}\sum_{j=1}^{B}\frac{R_j^r}{R_j^{r0}} \tag{6.3}$$

In addition to the performance goal, an equally important parameters for end users in any cloud service offering is the cost. Thus optimizing the overall infrastructure cost of running an application workload is our second key target. The cloud service provider cost model is different under a serverless and a serverful deployment model.

Existing commercial serverless platforms charge users based on the memory allocated (in MB) to the function instance (C_j^m), the execution time (in ms) of a request (E_j), and the number of requests received by the application (B). Thus if the charge per MB for 1 ms is W, and the charge per request is L, the total cost of executing an application workload on the serverless platform is,

$$Cost_s = \sum_{j=1}^{B} (C_j^m \times E_j \times W) + L \tag{6.4}$$

An IaaS platform on the other hand charges users for the whole period that the infrastructure is rented out. If t_l is the time period (in seconds) that v_l was rented and p_l is the price per second for the same,

$$Cost_{VM} = \sum_{i=1}^{M} p_l \times t_l \tag{6.5}$$

Thus the infrastructure cost optimization objective could be stated as below:

$$Minimize : Cost_{Total} = Cost_s + Cost_{VM} \tag{6.6}$$

Accordingly our overall target objective is summarized below:

$$Minimize : Average\ RRRT + Cost_{Total} \tag{6.7}$$

Table 6.2 summarizes the various symbols introduced in this section.

6.4 Deep Reinforcement Learning Model

This section introduces the application of the DRL concepts to our hybrid scheduling model, followed by a detailed discussion on our proposed actor-critic based framework.

6.4.1 Learning Model for Hybrid Scheduling

RL is a form of machine learning which predominantly works by learning through experience gathered by actively interacting with the problem environment. Based on the preferred outcome, the learning agent is rewarded whenever a 'better' action is taken. With enough experience, the agent ultimately learns to take actions leading to maximiz-

Table 6.2: Definition of Symbols.

Symbol	Definition
N	Set of nodes in the serverless cluster
V	Set of VMs in the serverful cluster
Q	Total number of deployed functions
$n^c_{i(t)}$	Available CPU in i^{th} node, $i \in [1, Q]$
$n^m_{i(t)}$	Available memory in i^{th} node, $i \in [1, Q]$
$v^c_{l(t)}$	Available CPU in l^{th} VM, $l \in [1, M]$
$v^m_{l(t)}$	Available memory in l^{th} VM, $l \in [1, M]$
R_j	The j^{th} request of an application, $j \in [1, B]$
C_j	Container running the j^{th} request of an application
C^c_j	Requested CPU by the j^{th} container
Cj^m	Requested memory by the j^{th} container
R^{r0}_j	Standard response time of the j^{th} request
R^r_j	Actual response time of the j^{th} request
E_j	Execution time of request, E_j
W	Per MB/s charge for serverless executions
L	Charge per request for serverless executions
p_i	Unit price of VM, v_l
t_l	Total active time of VM, v_l

ing the cumulative reward along experience trajectories.

In this work, the RL agent is tasked with traversing a serverless and VM based hybrid computing environment, in order to determine a request scheduling policy for application workloads. Each time step of the agent corresponds to the event of receiving a user request at the hybrid load balancer discussed under the system model. The policy to be developed is aimed at achieving our objectives of time and cost. The key elements of the RL model are discussed below.

State space: The state space captures the important metrics in both the serverless and VM based environments in addition to the workload characteristics. Accordingly, the first part of the state vector carries the serverless cluster node specifications: [n^c_i, n^m_i, n^{idle}_i], which identify the free cpu, memory capacities and the number of idle (warm) containers in each node. The second part includes the request details: [R^c_j, R^m_j, R^{rate}_j, R^d_j], representing the requested cpu, memory, moving average arrival rate and the deployment mode of the previous request, respectively. The moving average rate of arrival is calculated over a time frame which captures the total of the set up and the set idle time of a VM in the serverful cluster, which helps the model to gain an understanding on the historical load level as well as its duration. The mode of deployment in the preceding

step gives an indication of the availability of warm/active instances, and thus is helpful in decision making. The final portion of the state vector is composed of details of the IaaS cluster: $[v_l^c, v_l^m, v_l^s, v_l^t]$, referring to cpu, memory capacities, VM live status and the request waiting time for scheduling if selected. The last metric identifies how long it takes for the VM to come to 'ready' status and is calculated by the average time taken for a VM to start up and the remaining time since initializing the 'start' process.

Action space: Our action space takes a hierarchical form with two levels of decision making. The first level determines the deployment environment for the request while level two specifies the node of execution within the selected cluster. Accordingly, a complete action A can be represented as below:

$$A = [a_1, a_2] \begin{cases} a_1 \in \{serverless, serverful\} \\ a_2 \in \{n_1, n_2, ..., n_Q\} / \{v_1, v_2, ..., v_M\} \end{cases} \tag{6.8}$$

Compared to a combinatorial action space which does not distinguish between the two deployment modes, the RL agent is able to explore and learn the behavior of the hybrid environment faster with this formulation.

Reward: The step reward awarded to the agent after each action is aligned with the performance and cost objectives. We device two reward elements as below for action A_t:

R_1: The waiting time for the request to be scheduled. This is the resource creation time relevant as per the selected combined action. For a request directed to a node in the serverless platform, this would be equal to zero if there is any ready instance or one if a new container is to be created. If the VM cluster is selected for execution, this would either be zero or one for active or stopped VMs, else the remaining time for initializing as a fraction of the total, for a VM pending creation.

R_2: An approximation of the effective cost of running the request in the selected infrastructure. In the serverless cluster, this is the charge per request calculated as per Equation (6.4). For the VM based cluster, we use an approximate value calculated as the cost of keeping the selected VM active during the execution time of the request, multiplied by the percentage of free resources in the VM at the time.

The total reward is the aggregate of both R_1 and R_2 above. Since we need to minimize the cumulative of these rewards in order to reach our targets, we insert a negative sign

for this reward value when training the agent. Further, since R_1 and R_1 are in two different scales, we normalize them at each step.

6.4.2 Actor-critic based Hierarchical Scheduling Framework

Actor-critic methods in reinforcement learning make use of both the basic techniques of value based and policy based methods of finding the optimal policy for a given problem. Its fundamental architecture is designed with the use of two neural networks, the actor and the critic network. The actor is a policy network which uses an optimization method to train the network in the direction of the desired policy. Critic is driven by a value network which evaluates the policy generated by the actor.

Traditionally, actor-critic algorithms are implemented with one actor and one critic network. For our proposed hierarchical action space described above in section IV(A), we design a network architecture with two actor networks and one critic network adapting the hybrid actor-critic architecture presented in [209] for parameterized action spaces. The two parallel actors work together to generate a complete action. The first actor performs the first level of action selection by learning a stochastic policy π_{θ_1}, while the second actor learns a policy π_{θ_2} in order to select the second action.

The policy optimization in the actor networks could utilize any policy gradient algorithm which works with discrete environments and suits the basic actor-critic architecture, such as Trust Region Policy Optimization (TRPO) and Proximal Policy Optimization (PPO). Since TRPO is computationally intensive, we choose PPO for policy optimization in both the actor networks. It is proven to be an improved version of TRPO in terms of generalizability and its simplicity. PPO learns a stochastic policy π_θ by including a clipping function in its objective function and minimizing it.

$$L^{CLIP}(\theta) = E_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t] \tag{6.9}$$

where $r_t(\theta)$ identifies the probability ratio between the old policy and the new policy while ϵ is a hyper-parameter used to clip the objective function. The clipped function in PPO helps to keep the divergence of the old policy and the new policy within the trust region without having to use a constraint like in TRPO. In our proposed architecture, the

Algorithm 6 Actor-Critic based Hierarchical Scheduling Algorithm

1: Initialize the two actor network and critic network parameters θ_1, θ_2 and ϕ
2: Initialize the training parameters α, β, and γ
3: **for** episode = 1 to E **do**
4: Reset the environment
5: **for** step = 1 to T **do**
6: Input the state s of the environment to actor networks $\pi_{\theta_1}(a|s)$ and $\pi_{\theta_2}(a|s)$
7: Select action a_1 and a_2 using using the first and second actor networks
8: Execute the combined action $A = (a_1, a_2)$, move to the next state s' and observe the reward r
9: Store the transition (s, a, r, s') in memory D
10: **for** j = 1 to S **do**
11: Randomly sample a mini-batch of samples of size K from memory D
12: **for** sample i = 1 to K **do**
13: Compute the loss and the gradients of the loss of the two actor $\nabla_{\theta_1} J(\theta_1)$, $\nabla_{\theta_2} J(\theta_2)$ and critic $\nabla_{\phi} J(\phi)$ networks
14: Update actor and critic network parameters θ_1, θ_2 and ϕ
15: Clear memory D
return

two discrete policies π_{θ_1} and π_{θ_2} are updated separately by minimizing their respective clipped objectives during training.

The single critic network estimates the state-value function, $v_\pi(s_t|\phi)$ and learns by minimizing the difference between the target $(r + \gamma V_\phi(s'_t))$ and predicted values of the state, also known as the advantage function $A(s, a)$ as shown below.

$$\begin{aligned} J(\phi) &= r + \gamma V_\phi(s'_t) - V_\phi(s_t) \\ \phi &= \phi - \alpha \nabla_\phi J(\phi) \end{aligned} \tag{6.10}$$

where ϕ is the critic network parameter and $\nabla_\phi J(\phi)$ is the gradient of the network which is updated using gradient descent.

Algorithm 6 illustrates the pseudocode of the learning process of the proposed actor-critic based hierarchical scheduling framework. First we initialize the actor networks and the critic network with random weights and set the hyperparameters for model training (lines 1-2). At the start of each scheduling episode, the environment is reset. At each time step, the agent retrieves the state of the environment and feeds it to the

first and second actor networks. The first actor maps the request to either the serverless or VM based environment, after which the second actor determines a scheduling node in that environment. Once the request is forwarded for execution, the agent receives a reward and the environment transitions to the next state. The transition data are stored in a memory buffer (lines 5-9). When an episode comes to an end, we train the networks S times by sampling a batch of step data from the memory, computing the network losses for each step and by updating the network parameters (lines 10-14). Finally the memory is cleared before the start of the next episode.

6.5 Performance Evaluation

6.5.1 RL Environment Design and Implementation

We build a serverless testbed with the Kubeless [30] open source serverless framework deployed on a Kubernetes [158] cluster, set up on the Melbourne Research Cloud [175]. This prototype environment is used to do initial resource profiling for the applications used in our experiments in addition to gathering various system behavioral parameters such as container creation (cold start) times etc.

Following the system architecture in our serverless prototype described above and the overall system model presented under section 6.3.1, we have developed a simulation environment for serverless and serverful hybrid scheduling of applications. The serverful/VM based functionalities in the simulator are based on Amazon EC2 VM instances. Further, this event-based simulator written in python is integrated with Keras [32] and Tensorflow(TF) [31] libraries in order to support our DRL model training. Although in this work we explore only scheduling techniques, our simulator is capable of evaluating novel RL-based solutions for many resource management related tasks including resource provisioning, scheduling, and scaling etc. As mentioned above, it supports applications deployed in a serverless, VM based or a hybrid environment and the source code is publicly available as an open-source software, 'Hybrid_DRL[1]'.

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

6.5.2 Experimental Settings

Cluster Setup

Our experiments are designed to include 20 nodes in the simulated serverless cluster. The processing power of each of these node vCPUs is considered to follow the clock speeds of the AWS Lambda invokers identified in [39], with 4 different vCPU count and memory configurations. The simulated serverful cluster is also composed of 20 VMs and their configurations are derived from the Amazon EC2 VMs (in Australia) closely matching the clock speed, vCPU and RAM configurations of the serverless nodes. The pricing model of these VMs are set as per the instance pricing model of the EC2 VMs, while AWS Lambda GB-second and per request rates are utilized for the serverless cluster cost calculations. The VM-based cluster resource details are summarized in Table 6.3.

Workload Specifications

Serverless Applications: We select 10 applications from ServiBench [159] and FunctionBench [47] benchmark suites which are formed of a single function. These applications were chosen so that they have varying execution times which determine their sensitivity to cold start latencies. Further, each of them have different resource requirements and thus provides a diverse learning experience to the DRL agent, specially in

Table 6.3: VM-based Cluster Resource Details.

Instance Type	vCPU cores	Memory(GB)	Quantity	Price($/hr)
m6a.large	2	8	5	0.108
t4g.xlarge	4	16	5	0.1696
m5.2xlarge	8	32	5	0.48
m5a.4xlarge	16	64	5	0.864

terms of managing idling resources in the VM cluster. Our practical testbed is initially utilized for deriving resource metrics for the selected applications. A series of requests is sent to a ready instance of each application deployed in isolation, using the JMeter [179] load generation tool. The results from these tests collected by monitoring tools and averaged over multiple iterations are used to determine the resource consumption of a single function request (R_j^c, R_j^m) and its standard response time (R_j^{r0}). Table 6.4 summarizes the nature of the selected benchmark applications.

Workload Creation: We utilize metrics from function traces exposed by Azure Functions [46] for a set of single function applications, when creating the training workloads. The per hour arrival rates for a particular function are extracted as the request rates, and a poisson distribution is followed when determining the inter arrival times of requests. Each workload is created with a single application receiving requests at fluctuating arrival rates, with each rate prevailing for different time durations. The DRL agent is

Table 6.4: Application Details.

Name	**Resource Sensitivity**	
	CPU	Memory
Primary	High	High
Float	High	High
Matrix Multiplication	High	High
Linpack	High	High
Load	low	low
Dd	High	Medium
Gzip-compression	High	Medium
Thumbnail Generator	Low	Medium
Image Processing	Medium	Medium
Video Processing	High	High

trained with workloads of multiple applications with high and low request rates lasting for both short and long durations. In all the experiments, we maintain request arrival rates at 5-60 requests per second.

Hyper-parameter Configurations

Neural network training parameters for both the actor networks and also the critic network are decided on a trial and error basis. The discount factor is maintained at a high value since the scheduling decisions made over a period of time affect the success of the agent's decision making in terms of both the mode of deployment as well as the execution nodes. The second actor initially has a higher learning rate, so that the agent learns to take better node selection decisions during initial iterations, without which the first level decision of environment selection too will have no value. A decay factor is used to gradually bring down this learning rate to match that of the first actor subsequently. The critic constantly maintains a relatively higher discount rate since in the actor-critic architecture, the actors largely rely on the feedback and guidance of the critic network. These neural network parameters, and the other settings for training the DRL agent are listed in Table 6.5.

6.5.3 Performance Metrics

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

1. **Average Relative Request Response Time (RRRT)**: The average, relative request response time of an application workload during an episode, calculated using equation (6.3).

2. **User Cost**: The total cost of running an application workload in the hybrid cluster environment. The calculation of this metric is as in equation (6.6).

6.5.4 Baseline Scaling Techniques

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

Table 6.5: Hyper-parameters Used for DRL Model Training.

Parameter	Value
General	
Discount factor (γ)	0.99
Mini-batch size (K)	128
No. of training iterations per episode (S)	50
Optimizer	Adam
First Actor network parameters	
Learning rate (α_1)	1.00E-06
No. of input layers	1
No. of output layers	1
No. of hidden layers	2
No. of neurons in each hidden layer	150
Second Actor network parameters	
Learning rate (α_2)	1.00E-05
No. of input layers	2
No. of output layers	2
No. of hidden layers	4
No. of neurons in each hidden layer	150
Critic network parameters	
Learning rate (β)	5.00E-06
No. of input layers	1
No. of output layers	1
No. of hidden layers	2
No. of neurons in each hidden layer	150

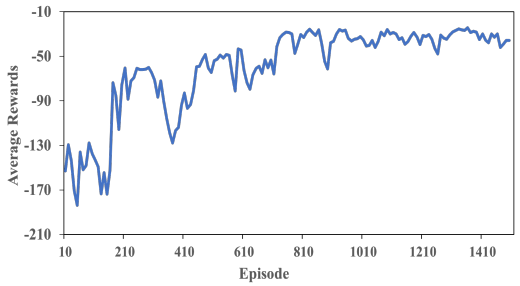
VM-only: The entire workload execution takes place on a VM-based cluster.

S-Only: The entire workload execution takes place on a serverless cluster.

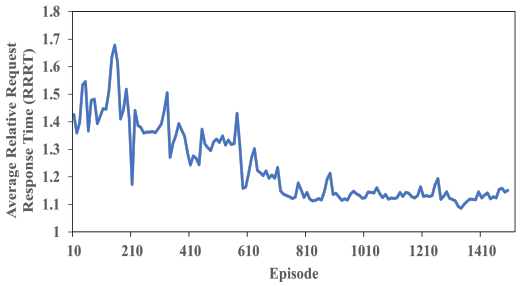
Std-A2C: DRL model trained with the standard actor-critic network architecture with a single actor network. The two levels of decision making are accommodated by composing a combinatorial action space where each action has two decision elements.

6.5.5 Convergence of the DRL Model

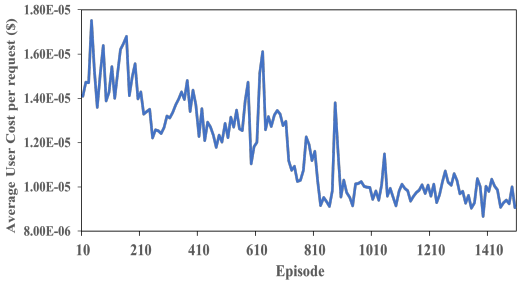
The graphs in Figure 6.2 illustrate the step-by-step training progress achieved by the hierarchical actor-critic agent, H-A2C across iterations. We demonstrate the progress in terms of the cumulative reward over an episode, the average RRRT experienced and the average user cost incurred for executing a request during an episode. Note that each marked value in the graphs corresponds to the average over 10 episodes for better readability and ease of understanding.



(a) Average Rewards.



(b) Average Relative Request Response Time (RRRT).



(c) Average User Cost per request ($).

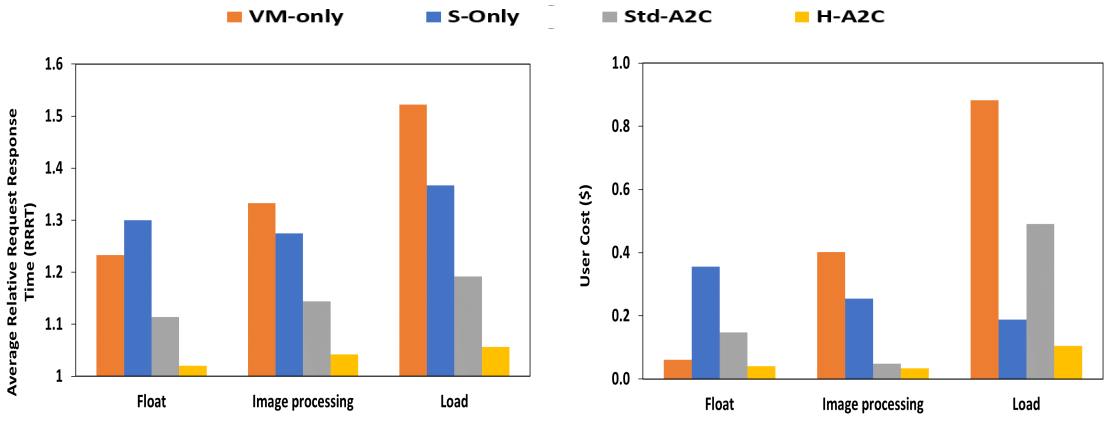
Figure 6.2: Training progress of the DRL agent in terms of the agent rewards, average RRRT, and the average user cost per request.

It is clear that the model reaches convergence around the 900^{th} iteration, after which the achieved progress is maintained. Figure 6.2(a) shows how the episodic reward gradually improves and reaches convergence, as the agent learns a policy that is able to take decisions that optimize the constituents of the reward metric at each scheduling step. Figures 6.2(b) and 6.2(c) illustrate the gradual reduction in overall RRRT and the incurred cost for requests with convergence, respectively. The logic behind this achievement is two-fold, since the agent has a hierarchical decision structure. The first actor network learns to select a deployment environment that leads to lower relative response times by targeting lesser frequency in cold starts, considering the nature of the application specially in terms of its execution time. The cold starts refer to the container creation time in a serverless execution as well as the time for setting up new resources in a VM setting if all active resources are exhausted. While aiming for a reduced RRRT, the model also learns to aim for an environment where the marginal cost for each request

would be minimal. This could be based on the availability of warm function instances in a serverless setting or active VMs with free resources in a VM setting. Subsequent to the first actor's decision, the second actor uses its learned policy to select an execution node in the selected environment, that further elevates the target metrics. A node with ready instances could be the better choice in a serverless setting, while an active VM with higher utilization levels which leads to lesser idling times, could be selected in a VM-based setting.

6.5.6 Analysis of Model Performance on the Evaluation Data Sets

The trained model is evaluated in terms of the response time and user cost performance achieved over the evaluation workloads. The workloads for these experiments are created by following trace snippets from Wikipedia [154] to simulate request arrival times. The Figures 6.3(a) and (b) show the results averaged over five different workloads, run for each of the applications float, load and image processing, which have been selected out of the ten applications used in the experiments, due to space limitations. These three applications were specifically chosen for demonstration of model performance due to their distinction in resource consumption and execution times. Further, Figs. 6.4 and 6.5



(a) Average Relative Request Response Time (RRRT).

(b) Total User Cost ($).

Figure 6.3: Comparison of the average RRRT and the total user cost incurred by different application workloads, achieved by the H-A2C model and the baseline algorithms.

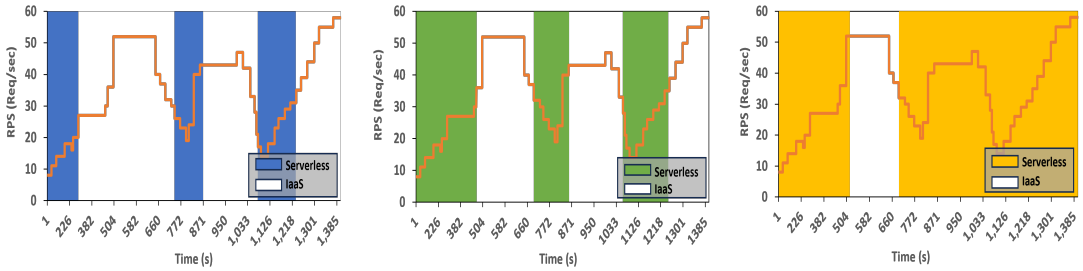
illustrate the behavior of the agent decision model for two of the evaluation workloads, with the points of switching deployments between the serverless and IaaS clusters.

Evaluation of application performance

Application performance is evaluated in terms of the achieved average RRRT (Figure 6.3 (a)) for each application by following the trained policy of our hierarchical A2C (H-A2C) model and the other baseline algorithms, for scheduling workload requests.

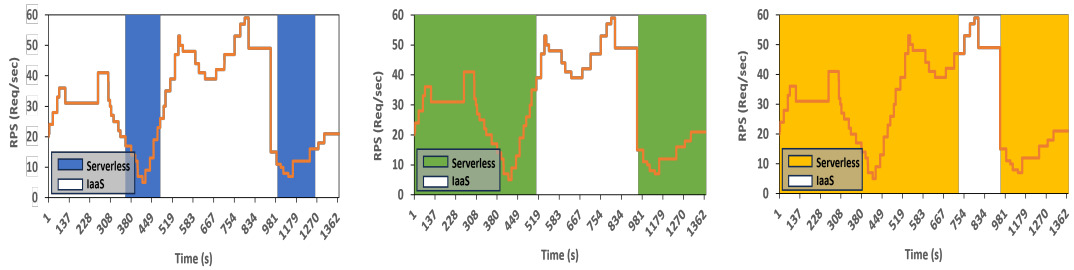
Our H-A2C model is able to demonstrate the best performance for all three applications, with overall relative response time improvements of up to 11%. This is achieved by carefully directing each request to the execution environment that is more likely to have ready resources for the upcoming request traffic as shown in Figures 6.4 and 6.5. Once the deployment mode is decided, the trained agent is also able to choose the best out of the available cluster nodes. Both these decisions are taken to suit each phase of the workload considering the prevailing traffic patterns. The trained S-A2C model shows the next best performance, but fails to capture the fine details of each cluster environment, that is enabled by the hierarchical nature of the H-A2C model.

The float application has the highest CPU and memory consumption and the longest execution time out of the three. Thus when the requests are scheduled solely on a VM cluster, it is able to maintain a relatively high utilization level in the rented resources even at low load levels, in the long term. This triggers auto-scaling of VM resources in the cluster, leading to lesser request latency effects arising from new VM initialization.



(a) Float. (b) Image Processing. (c) Load.

Figure 6.4: Workload-1: Deployment switch between Serverless and IaaS clusters for the three applications.



(a) Float. (b) Image Processing. (c) Load.

Figure 6.5: Workload-2: Deployment switch between Serverless and IaaS clusters for the three applications.

However, when the entire workload is executed as serverless functions, due to the high application response time, the availability of warm idling containers is often limited. Thus the s-only execution shows the worst performance resulting from frequent cold start of instances (first graph of Fig. 6.3 (a)). To overcome the shortcomings of both these scenarios, our H-A2C model resorts to serverless deployments during periods of very low and fluctuating traffic levels, and switches to the VM cluster as the load starts to increase and then stabilizes, as seen in the graphs 6.4(a) and 6.5(a).

In contrast, the load application results in the need for frequent startup of new VMs in the IaaS cluster to accommodate new requests. This is due to its very low execution time and resource requirements. For the same reasons, the effect of these added latencies on the relative response time too is high for the load application. This situation escalates during periods of low and irregular traffic patterns, when VMs are switched on and off often due to idling. On the other hand, the s-only execution performance is not much different from the float application scenario since the increase in latency is mostly only due to the higher relative effect of the cold start delays, arising from the low application response time. In comparison to these two strategies, we can see that the H-A2c model leverages both these clusters by executing the application requests as functions for the majority of the workload, while utilizing the VM cluster only during periods of lasting high traffic levels (Figures 6.4(c) and 6.5(c)). During these traffic bursts, the load level seems to be sufficient to maintain a set of ready VMs, thus compensating for the cold start latencies that the serverless execution would otherwise entail.

The image processing application possesses a median response time and resource needs. Thus its relative latency effects are less evident and more prominent compared to that of the load and float applications respectively.

Evaluation of incurred user cost

The cost charged to the user is measured as the total billed amount for the rented VM resources and function executions, by each application over the duration of the workload. Figure 6.3(b) shows that the scheduling decisions made using the H-A2C model result in the least cost charged to the user for all three applications, with the load application attaining a cost reduction of approximately 44%. The trained policy is equipped with knowledge on the workload and system behavioural patterns, so as to make informed decisions on when to switch deployments to each cluster environment, that are cost efficient. The reduction in VM idling time and maximizing the usage of the serverful cluster by maintaining just the right amount of resources active, is the key enabler for overall user cost minimization. Since the per request charge for a serverless function execution is quite high, the scheduler chooses to use it only during periods of fluctuating and low load levels, during which the use of rented VMs results in cost inefficiencies. The lowest overall cost under the H-A2C model is incurred by the image processing application due to its better use of both the environments compared to the load application, in addition to the lower actual resource consumption than the float application. The Std-A2c model too follows this cost pattern.

The VM-only deployment results in worst performance for the load application due to high VM idling times, with the highest incurred cost for the workload execution out of all the applications. The S-only execution cost difference for the three applications is only dependent on the consumed level of resources and the workload execution times, as per the serverless billing model.

6.6 Summary

With the increased adoption of the serverless cloud model for different application domains, studies have shown that limitations also exist in these environments that hinder the achievement of the best possible performance for certain workloads.

In this chapter, we presented a DRL based hybrid execution model for application workloads, which utilizes both a serverless and a serverful cluster environment. The proposed scheduling framework involves a hierarchical decision model, where the DRL agent first learns to choose the best mode of execution for an application request. Thereafter, it proceeds to decide the node that is most suitable for the request execution within the selected cluster environment. The DRL agent follows an actor-critic architecture and the reward model is set targeting relative application latency and the resource cost charged to the user as the optimization objectives. The model evaluation experiments show that the users are able to avoid application latencies arising from frequent container cold starts in a serverless environment, as well as the problem of over/under utilization of rented infrastructure in a VM setting, by adopting a carefully designed hybrid system architecture.

Chapter 7
Conclusions and Future Directions

This chapter concludes the thesis and summarizes our work highlighting its key contributions. It also proposes several potential future research directions that support the progress of the serverless computing eco-system.

7.1 Summary of Contributions

Serverless computing could be considered the latest step in the evolution of cloud computing technologies which has caused a vast transformation in the cloud deployment model. Its increasing popularity among industry giants is largely due to the convenience it brings to the end users in executing their applications in the cloud. Under the serverless model, the end users are only expected to provide their business logic in a modular form called 'functions', and thereafter the cloud vendor takes the full responsibility of the successful execution of the application. This results in reduced cost to the users in terms of any upfront infrastructure costs, expertise in server management as well as maintaining any redundant bare-metal servers or virtual resources when there is no application traffic. Seamless scale up/down of allocated resources and the exclusive pay-per-use billing model have further elevated the growing interest in this cloud model.

However, the serverless computing model does require the cloud vendor to undertake all of the tasks that the end users are now relieved of. This is a highly challenging situation since cloud service providers have to serve numerous users at a time with each having their own requirements. Minimal involvement of the application owners in the execution process essentially means that the vendors have access to only very little

information on the applications being deployed. Further, the highly anticipated auto-scalability feature and the granular billing model complicate the resource management process for the providers. Thus, the successful undertaking of the end-to-end resource management operations in a serverless system requires the platform operators to follow thoroughly tested solid techniques at each step. In this thesis, we investigated dynamic techniques for handling several key resource management processes including resource provisioning, scheduling and scaling, to the satisfaction of both the cloud vendor and their clients.

Chapter 1 introduced the serverless computing paradigm with a discussion on the evolution of various cloud deployment models over time. Next, the architecture of this novel paradigm is presented along with its distinguishing features. Then the existing serverless platforms are analyzed followed by a discussion on few popular use-cases for adopting this model. Further this chapter presented challenges in serverless resource management, forming the basis of our research. Finally the research questions addressed in this thesis are highlighted and the thesis contributions are summarized.

Chapter 2 identified the major aspects of resource management in a serverless computing environment, namely, workload characterization and performance prediction, resource scheduling and resource scaling. This is followed by an analysis on the challenges associated with these key decisions, which are specially significant in serverless computing environments. Next, a detailed taxonomy of the factors which need to be considered when developing successful resource management strategies overcoming these challenges, is presented. Further, existing related works are analysed in detail using the taxonomy, followed by a discussion on the identified research gaps with great potential for future studies.

Chapter 3 presented a dynamic resource provisioning and a function request placement technique, for overcoming inefficiencies of initial resource allocations to requests and under-utilization of provider cloud infrastructure. The deadline-sensitive placement algorithm is aimed at reducing provider resource wastage, while the allocated resources to functions are analysed in the run-time to ensure the satisfaction of application requirements. This work also entailed a novel addition to the existing CloudSim simulation environment which supports serverless function executions.

Chapter 4 proposed a DRL-based technique for function scheduling, focusing on the resource constrained and multi-tenant nature of serverless systems. The multi-step DQN algorithm adapted in this work formulates a comprehensive understanding on the workloads that they handle along with the corresponding system behavior patterns in reaching its solution. Further, our solution offers flexibility to its users for balancing the objectives of application response time latency and provider cost efficiency, as desired. A fully fledged practical serverless test-bed environment was also designed, on which all the experiments for training and evaluating multiple variations of the proposed model were conducted.

Chapter 5 presented a framework for response time and cloud provider resource cost optimized scaling of serverless applications. The proposed multi-agent DRL model for the function scaling problem is adapted from the policy gradient algorithm Asynchronous Advantage Actor Critic (A3C). The scaling solution is formulated to include both horizontal and vertical scaling decisions, by incorporating a novel multi-discrete action space. Moreover, the solution includes a configurable reward model for carefully balancing the two conflicting objectives of performance and resource efficiency. An event-based simulator environment integrated with TF agents in the back-end and following the architecture of our existing practical test-bed, was also designed as part of this work, for effective evaluation of the proposed scaling solution.

Chapter 6 introduced a serverless and VM-based hybrid solution for scheduling applications in order to leverage the optimum benefits of this novel computing model. The proposed solution utilizes the actor-critic architecture in DRL, along with the proximal policy optimization (PPO) technique. The model decision structure takes a hierarchical form with multiple actors working on deciding the environment of deployment and the scheduling node in the selected environment. The multi-objective reward model is capable of enhancing both application performance and user cost in the trained policy. This is achieved by capturing the application workload as well as the serverless and serverful cluster status and behavioral patterns in the decision model.

The chapters described above collectively presented elaborate techniques for the autonomous handling of various resource management aspects under the provider-centric serverless computing model, which is a timely contribution to the state-of-the-art. The

outcomes of these studies provide solid proof of the value of handling serverless resource management with a deep understanding on the whole eco-system, for the benefit of all stakeholders.

7.2 Future Research Directions

Based on the research presented in this thesis, here we present ideas with great potential for future investigation in to the aspect of resource management in serverless computing.

7.2.1 Multi-provider Serverless Support

With the advent of many cloud vendors in to the market with serverless computing offerings, there exist a variety of features in each platform that the users could benefit from. However, due to the lack of a common set of standards binding each of these products, users are easily subject to a state of vendor lock-in as soon as they start adopting one service. Thus today, interest is building up on studies on multi-provider serverless support with reduced provider lock-in effect for users. This is also in line with the concept of adopting multi-cloud infrastructure solutions by organizations looking out for the most cost effective model. Accordingly, users would be able to dynamically enjoy various offerings of multiple serverless vendors potentially via a brokering mechanism which would determine the best execution path for user traffic based on the desired objectives.

7.2.2 Hybrid Execution Models

In chapter 6 we studied the use of a hybrid execution model for applications for realizing benefits of both serverless and traditional serverful models. Our results provided an insight into how a single task application workload could be analyzed for identifying the most suitable load levels for deployment in each environment. This research could be further expanded to explore the most efficient mode of deployment for a variety of other applications, e.g., a DAG (Directed Acyclic Graph) based workflow application

with a complex structure. A product offering with such customized, hybrid deployment options, from a cloud vendor would have great future prospects.

7.2.3 Access to Specialized Hardware

Many diverse application domains are increasingly adopting the serverless computing model nowadays. Understandably, different user applications have varying resource requirements and there could even be the need to access specialized hardware resources for their successful execution. With the penetration of artificial intelligence in to almost all aspects of life today, deep learning model training has become a potential candidate for serverless computing, with a vast number of use-cases. However these applications commonly use GPUs or TPUs as an accelerator for processing compute-intensive tasks, which the current serverless platforms do not provide. Thus, availability of flexible hardware options along with techniques for managing applications tasks efficiently on them could reduce the barrier to entry for such prominent use-cases.

7.2.4 Dynamic Pricing Models

Current serverless platforms charge users based on a common pay-as-you-use billing model. However, depending on the criticality of their requirements, certain users may be willing to pay more or less for an enhanced/diminished level of service. Such flexible pricing models which address the money-to-performance trade-off would enhance the meaning of the term "Function-as-a-service," while luring in more diversified user pools.

7.2.5 QoS Guarantees

A major challenge for serverless system users is the lack of any QoS guarantees for their applications by the vendors. Specially for mission critical applications which require high availability and reliability guarantees, this shared platform architecture may not seem a viable option unless a guaranteed level of service could be agreed upon. Nevertheless, providing custom service level offerings to its users on multi-tenant infrastruc-

ture adds unnecessary complications to the system which may affect the maintenance of the distinguished characteristics of the serverless model such as auto-scalability. Thus, thorough investigation is required in this regard for designing possible solutions without compromising on the system fundamentals.

7.2.6 Provider Centric Optimization

With the many advantages it offers, serverless computing is cited as a lucrative mode of application deployment among the general cloud user. Auto-scalability, efficient billing models and complete omission of any upfront costs are just some of its many characteristics that benefit the end users. Moreover, many research works are constantly undertaken in order to study techniques for further optimizing the user experience in these systems. However, this being a provider-centric cloud model, where the control of all operations lies with the vendor, any such techniques would only be admitted to their systems if they benefit the provider as well. For example, the maintenance of large resource pools for mitigating application latency deterioration would have very low attractiveness to a provider if it leads to a large resource wastage. Thus in chapters 4, 5 and 6, we have largely directed our research focus on resource management techniques which are capable of achieving cost efficiency to the cloud provider. In line with the cost metric, another parameter of great interest to a service provider would be the aspect of energy efficiency of their infrastructure. In our work, we have incorporated the opportunity cost of running the physical machines, to arrive at provider cost. Potential next steps for research would be determining the energy consumption of the underlying resources, which is a direct contributor to service provider cost. Thus, critically analyzing the QoS requirements from a service provider's point of view too needs to be a key research goal.

7.2.7 Adaptation to the Edge

A lot of effort is seen today in the area of adapting the serverless model across the edge and fog computing networks. Internet of Things (IoT) applications running on resource constrained edge nodes could benefit a lot by following the modular application ar-

chitecture in serverless, where the consolidation of several functions would form one application. However in contrast to cloud deployed functions, edge deployments need to consider additional parameters such as network delays, resource limitations and vast heterogeneity of edge devices. For example, the data shipping architecture in serverless which essentially transports data to the processing element for each function execution could cause excessive latencies, where an intermediary storage or caching services could be possible solutions. Although several researches exist which focus on creating novel resource isolation mechanisms and architectures to overcome the constraints of the edge environment, currently there is an evident lack of specific focus to improve resource management in these network architectures. The techniques presented in this thesis could be easily adapted to suit these heterogeneous environments. Hence, the consolidation of the two concepts of edge and serverless computing paves the path to many new research topics.

7.2.8 Intelligent Solutions

The shared nature of serverless platforms creates a very complex environment for function execution, with many dependent factors influencing its performance. Further, functions being naturally short-lived code segments with no state maintenance, and the autoscaling of resources elevates the dynamism in these systems where the environment is subject to constant and rapid changes. As such, any resource management technique proposed for this cloud model need to be adaptable to these dynamic circumstances. Thus, employing intelligent methods for capturing these complex details is of high relevance. In this thesis, we have employed many such learning techniques for identifying and rectifying shortcomings in serverless resource allocation decisions, which have a massive potential for further studies.

7.2.9 Support for Dynamism in Environments and Workloads

Our thesis proposed strategies for resource management in serverless environments considering the nature of majority of workloads utilizing this architecture in existing commercial platforms. We have designed our experimental testbeds to closely resonate

with the underlying infrastructure of these systems in order to validate the practical adaptation of our solutions. However, there is further potential to examine the applicability of our findings under less common application workload scenarios (e.g.: big data applications) and diverse environmental settings (e.g.: varying cluster sizes and hardware setups). Studying the behaviour of these techniques under such diverse conditions, would be of great value to this area of research.

7.3 Final Remarks

Serverless computing has gained a lot of popularity over recent years among individual as well as industry cloud users due to its versatility in accommodating requirements of applications belonging to diverse domains. The unmatchable convenience that it delivers to the end users by seamlessly undertaking all the application execution operations with minimal user involvement, greatly elevates the efficiency of business processes. Fine-grained auto-scaling of application resources and favorable billing models have further heightened the interest in this new computing model among prospective users. However, maintaining the seemingly 'serverless' nature of operations from the end user perspective comes with increased levels of responsibility to the cloud vendors. From the moment an application is deployed, all end-to-end operations for its successful execution need to be handled by the platform itself, with little to no knowledge on the particulars of application characteristics. In addition, since all serverless platforms are multi-tenant shared environments, the demands of numerous users need to be met simultaneously. In this thesis, we investigated and designed techniques for autonomously and dynamically managing application resources in serverless systems to the satisfaction of both the cloud users and service providers equally. These include strategies and architectures for resource provisioning, scheduling and scaling which work towards harvesting the full potential of this computing model. The research outcomes of this thesis lay the groundwork and pave the way for further innovation and evolution in this novel cloud computing paradigm.

Bibliography

[1] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," in Research Advances in Cloud Computing. Springer, 2017, pp. 1–20.

[2] "Serverless computing market size 2023 with a cagr of 20.8% : Latest growth rate, new development, market segment, sales & revenue, global demand and regional outlook till forecast year 2030 research report," https://au.finance.yahoo.com/news/serverless-computing-market-size-2023-111100932.html, (Accessed on 08/10/2023).

[3] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," Communications of the ACM, vol. 62, no. 12, pp. 44–54, 2019.

[4] S. Werner, J. Kuhlenkamp, M. Klems, J. Müller, and S. Tai, "Serverless big data processing using matrix multiplication as example," in Proceedings of the IEEE International Conference on Big Data (Big Data). IEEE, 2018, pp. 358–365.

[5] Y. Kim and J. Lin, "Serverless data analytics with flint," in Proceedings of the IEEE 11th International Conference on Cloud Computing (CLOUD). IEEE, 2018, pp. 451–455.

[6] P. García-López, M. Sánchez-Artigas, S. Shillaker, P. Pietzuch, D. Breitgand, G. Vernik, P. Sutra, T. Tarrant, and A. Juan Ferrer, "Servermix: Tradeoffs and challenges of serverless data analytics," arXiv, pp. arXiv–1907, 2019.

[7] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: machine learning inference serving on serverless platforms with adaptive batching," in Proceedings of SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC). IEEE Computer Society, 2020, pp. 972–986.

[8] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in Proceedings of the ACM Symposium on Cloud Computing, 2019, pp. 13–24.

[9] L. Baresi and D. F. Mendonça, "Towards a serverless platform for edge computing," in Proceedings of the IEEE International Conference on Fog Computing (ICFC). IEEE, 2019, pp. 1–10.

[10] D. Bermbach, S. Maghsudi, J. Hasenburg, and T. Pfandzelter, "Towards auction-based function placement in serverless fog platforms," in Proceedings of the IEEE International Conference on Fog Computing (ICFC). IEEE, 2020, pp. 25–31.

[11] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "Numpywren: Serverless linear algebra," arXiv preprint arXiv:1810.09679, 2018.

[12] B. Cheng, J. Fuerst, G. Solmaz, and T. Sanada, "Fog function: Serverless fog computing for data intensive iot services," in Proceedings of the IEEE International Conference on Services Computing (SCC). IEEE, 2019, pp. 28–35.

[13] K. Kritikos and P. Skrzypek, "A review of serverless frameworks," in Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). IEEE, 2018, pp. 161–168.

[14] J. Kuhlenkamp, S. Werner, M. C. Borges, K. El Tal, and S. Tai, "An evaluation of faas platforms as a foundation for serverless big data processing," in Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing, 2019, pp. 1–9.

[15] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "A review of serverless use cases and their characteristics," arXiv preprint arXiv:2008.11110, 2020.

[16] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin et al., "Apache spark: a unified engine for big data processing," Communications of the ACM, vol. 59, no. 11, pp. 56–65, 2016.

[17] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," Bulletin of the

IEEE Computer Society Technical Committee on Data Engineering, vol. 36, no. 4, 2015.

[18] "Aws lambda - developer guide," https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf, (Accessed on 08/31/2020).

[19] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, "Serverless data analytics in the ibm cloud," in Proceedings of the 19th International Middleware Conference Industry, 2018, pp. 1–8.

[20] V. Giménez-Alventosa, G. Moltó, and M. Caballer, "A framework and a performance assessment for serverless mapreduce on aws lambda," Future Generation Computer Systems, vol. 97, pp. 259–274, 2019.

[21] J. Enes, R. R. Expósito, and J. Touriño, "Real-time resource scaling platform for big data workloads on serverless environments," Future Generation Computer Systems, vol. 105, pp. 361–379, 2020.

[22] A. W. Services, "Aws iot greengrass - amazon web services," https://aws.amazon.com/greengrass/, 2021, (Accessed on 01/08/2021).

[23] M. Azure, "Azure iot edge documentation — microsoft docs," https://docs.microsoft.com/en-us/azure/iot-edge/?view=iotedge-2018-06, 2021, (Accessed on 01/14/2021).

[24] T. Elgamal, "Costless: Optimizing cost of serverless computing through function fusion and placement," in Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC). IEEE, 2018, pp. 300–312.

[25] D. Pinto, J. P. Dias, and H. S. Ferreira, "Dynamic allocation of serverless functions in iot environments," in Proceedings of the IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC). IEEE, 2018, pp. 1–8.

[26] A. Das, S. Imai, S. Patterson, and M. P. Wittie, "Performance optimization for edge-cloud serverless platforms via dynamic task placement," in Proceedings of the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). IEEE, 2020, pp. 41–50.

[27] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar et al., "Cloud programming simplified: A berkeley view on serverless computing," arXiv preprint arXiv:1902.03383, 2019.

[28] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless linear algebra," in Proceedings of the 11th ACM Symposium on Cloud Computing, 2020, pp. 281–295.

[29] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," Software: Practice and experience, vol. 41, no. 1, pp. 23–50, 2011.

[30] Kubeless, "Kubeless," https://kubeless.io/, 2021, (Accessed on 01/13/2022).

[31] TensorFlow, "Tensorflow," https://www.tensorflow.org/, 2021, (Accessed on 01/20/2022).

[32] Keras, "Keras: the python deep learning api," https://keras.io/, 2021, (Accessed on 01/20/2022).

[33] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: a survey of opportunities, challenges, and applications," ACM Computing Surveys, vol. 54, no. 11s, pp. 1–32, 2022.

[34] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of serverless computing and function-as-a-service (faas) in industry and research," arXiv preprint arXiv:1708.08028, 2017.

[35] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," Communications of the ACM, vol. 64, no. 5, pp. 76–84, 2021.

[36] A. Raza, I. Matta, N. Akhtar, V. Kalavri, and V. Isahagian, "Sok: Function-as-a-service: From an application developer's perspective," Journal of Systems Research, vol. 1, no. 1, 2021.

[37] G. A. S. Cassel, V. F. Rodrigues, R. da Rosa Righi, M. R. Bez, A. C. Nepomuceno, and C. A. da Costa, "Serverless computing for internet of things: A systematic literature review," Future Generation Computer Systems, vol. 128, pp. 299–316, 2022.

[38] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in Proceedings of the Conference on Innovative Data Systems Research, 2019.

[39] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in Proceedings of the Annual Technical Conference, 2018.

[40] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in Proceedings of the 11th IEEE International Conference on Cloud Computing (CLOUD). IEEE, 2018, pp. 442–450.

[41] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," Journal of Systems and Software, vol. 170, p. 110708, 2020.

[42] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, and F. Leymann, "Faasten your decisions: A classification framework and technology review of function-as-a-service platforms," Journal of Systems and Software, vol. 175, p. 110906, 2021.

[43] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, and A. Iosup, "The state of serverless applications: Collection, characterization, and community consensus," IEEE Transactions on Software Engineering, no. 01, pp. 1–1, 2021.

[44] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, and A. Akella, "Atoll: A scalable low-latency serverless platform," in Proceedings of the ACM Symposium on Cloud Computing, 2021, pp. 138–152.

[45] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud," in Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD). IEEE, 2019, pp. 199–208.

[46] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and 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, 2020, pp. 205–218.

[47] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD). IEEE, 2019, pp. 502–504.

[48] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bermbach, "Befaas: An application-centric benchmarking framework for faas platforms," in Proceedings of the 9th IEEE International Conference on Cloud Engineering. IEEE, 2021.

[49] S. Eismann, J. Grohmann, E. Van Eyk, N. Herbst, and S. Kounev, "Predicting the costs of serverless workflows," in Proceedings of the ACM/SPEC International Conference on Performance Engineering, 2020, pp. 265–276.

[50] A. Das, A. Leaf, C. A. Varela, and S. Patterson, "Skedulix: Hybrid cloud scheduling for cost-efficient execution of serverless applications," in Proceedings of the IEEE 13th International Conference on Cloud Computing (CLOUD). IEEE, 2020, pp. 609–618.

[51] C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," IEEE Transactions on Parallel and Distributed Systems, vol. 32, no. 3, pp. 615–632, 2020.

[52] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "Cose: Configuring serverless functions using statistical learning," in Proceedings of the INFOCOM - IEEE Conference on Computer Communications. IEEE, 2020, pp. 129–138.

[53] R. Cordingly, W. Shu, and W. J. Lloyd, "Predicting performance and cost of serverless computing functions with saaf," in Proceedings of the IEEE International Conference on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech). IEEE, 2020, pp. 640–649.

[54] N. Mahmoudi and H. Khazaei, "Performance modeling of serverless computing platforms," IEEE Transactions on Cloud Computing, no. 01, pp. 1–1, 2020.

[55] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless era," in Proceedings of the 21st International Middleware Conference, 2020, pp. 280–295.

[56] M. Zhang, C. Krintz, and R. Wolski, "Edge-adaptable serverless acceleration for machine learning internet of things applications," Software: Practice and Experience, 2020.

[57] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in Proceedings of the 11th ACM Symposium on Cloud Computing, 2020, pp. 30–44.

[58] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "Faasdom: A benchmark suite for serverless computing," in Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems, 2020, pp. 73–84.

[59] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in Proceedings of the 22nd ACM International Middleware Conference, 2021.

[60] N. Mahmoudi and H. Khazaei, "Simfaas: A performance simulator for serverless computing platforms," arXiv preprint arXiv:2102.08904, 2021.

[61] M. Stein, "Adaptive event dispatching in serverless computing infrastructures," Ph.D. dissertation, Brunel University London, 2018.

[62] A. Mampage, S. Karunasekera, and 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.

[63] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and 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.

[64] W. Ling, L. Ma, C. Tian, and 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.

[65] G. De Palma, S. Giallorenzo, J. Mauro, and G. Zavattaro, "Allocation priority policies for serverless function-execution scheduling optimisation," in Proceedings of the International Conference on Service-Oriented Computing. Springer, 2020, pp. 416–430.

[66] L. Schuler, S. Jamil, and 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.

[67] N. Mahmoudi, C. Lin, H. Khazaei, and 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.

[68] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD). IEEE, 2019, pp. 329–338.

[69] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, "Barista: Efficient and scalable serverless serving system for deep learning prediction services," in Proceedings of the IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2019, pp. 23–33.

[70] M. Stein, "The serverless scheduling problem and noah," arXiv, pp. arXiv–1809, 2018.

[71] D. Bermbach, A.-S. Karakaya, and S. Buchholz, "Using application knowledge to reduce cold starts in faas services," in Proceedings of the 35th Annual ACM Symposium on Applied Computing, 2020, pp. 134–143.

[72] K. Solaiman and M. A. Adnan, "Wlec: A not so cold architecture to mitigate cold start problem in serverless computing," in Proceedings of the IEEE International Conference on Cloud Engineering (IC2E). IEEE, 2020, pp. 144–153.

[73] Z. Xu, H. Zhang, X. Geng, Q. Wu, and H. Ma, "Adaptive function launching acceleration in serverless computing platforms," in Proceedings of the 25th International Conference on Parallel and Distributed Systems (ICPADS). IEEE, 2019, pp. 9–16.

[74] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in Proceedings of the 21st International Middleware Conference, 2020, pp. 356–370.

[75] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19), 2019.

[76] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in Proceedings of the 21st International Middleware Conference, 2020, pp. 1–13.

[77] G. Somma, C. Ayimba, P. Casari, S. P. Romano, and V. Mancuso, "When less is more: Core-restricted container provisioning for serverless comput-

ing," in Proceedings of the IEEE INFOCOM - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). IEEE, 2020, pp. 1153–1159.

[78] A. U. Gias and G. Casale, "Cocoa: Cold start aware capacity planning for function-as-a-service platforms," in Proceedings of the 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, 2020, pp. 1–8.

[79] E. Hunhoff, S. Irshad, V. Thurimella, A. Tariq, and E. Rozner, "Proactive serverless function resource management," in Proceedings of the 6th International Workshop on Serverless Computing, 2020, pp. 61–66.

[80] Docker, "Runtime options with memory, cpus, and gpus — docker documentation," https://docs.docker.com/config/containers/resource_constraints/, 2021, (Accessed on 01/04/2021).

[81] P. Turner, B. B. Rao, and N. Rao, "Cpu bandwidth control for cfs," in Proceedings of the Linux Symposium. Citeseer, 2010, p. 245.

[82] H. Yu, H. Wang, J. Li, and S.-J. Park, "Harvesting idle resources in serverless computing via reinforcement learning," arXiv preprint arXiv:2108.12717, 2021.

[83] OpenFaas, "Home — openfaas - serverless functions made simple," https://www.openfaas.com/, 2021, (Accessed on 11/22/2021).

[84] A. F. Baarzi, G. Kesidis, C. Joe-Wong, and M. Shahrad, "On merits and viability of multi-cloud serverless," in Proceedings of the ACM Symposium on Cloud Computing, 2021, pp. 600–608.

[85] H. Wang, P. Shi, and Y. Zhang, "Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization," in Proceedings of the IEEE 37th international conference on distributed computing systems (ICDCS). IEEE, 2017, pp. 1846–1855.

[86] J. Liu, Z. Mi, Z. Huang, Z. Hua, and Y. Xia, "Hcloud: A serverless platform for jointcloud computing," in Proceedings of the IEEE International Conference on Joint Cloud Computing. IEEE, 2020, pp. 86–93.

[87] M. Ciavotta, D. Motterlini, M. Savi, and A. Tundo, "Dfaas: Decentralized function-as-a-service for federated edge computing," in Proceedings of the IEEE 10th International Conference on Cloud Networking (CloudNet). IEEE, 2021, pp. 1–4.

[88] B. Soltani, A. Ghenai, and N. Zeghib, "Towards distributed containerized serverless architecture in multi cloud environment," Procedia computer science, vol. 134, pp. 121–128, 2018.

[89] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen, "Function delivery network: Extending serverless computing for heterogeneous platforms," Software: Practice and Experience, vol. 51, no. 9, pp. 1936–1963, 2021.

[90] F. Gand, I. Fronza, N. El Ioini, H. R. Barzegar, and C. Pahl, "Serverless container cluster management for lightweight edge clouds," in Proceedings of the 10th International Conference on Cloud Computing and Services Science. SCITE Press, 2020.

[91] L. Baresi, D. F. Mendonça, and M. Garriga, "Empowering low-latency applications through a serverless edge computing architecture," in Proceedings of the European Conference on Service-Oriented and Cloud Computing. Springer, 2017, pp. 196–210.

[92] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in Proceedings of the International Conference on Internet of Things Design and Implementation, 2019, pp. 225–236.

[93] Z. Li, Q. Chen, S. Xue, T. Ma, Y. Yang, Z. Song, and M. Guo, "Amoeba: Qos-awareness and reduced resource usage of microservices with serverless computing," in Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2020, pp. 399–408.

[94] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in Proceedings of the 17th USENIX symposium on networked systems design and implementation ({nsdi} 20), 2020, pp. 419–434.

[95] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in Proceedings of the USENIX Annual Technical Conference, 2018, pp. 923–935.

[96] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "{SOCK}: Rapid task provisioning with serverless-optimized containers," in Proceedings of the USENIX Annual Technical Conference, 2018, pp. 57–70.

[97] V. Gupta, S. Phade, T. Courtade, and K. Ramchandran, "Utility-based resource allocation and pricing for serverless computing," arXiv preprint arXiv:2008.07793, 2020.

[98] M. Azure, "Azure functions documentation — microsoft docs," https://docs.microsoft.com/en-us/azure/azure-functions/, 2021, (Accessed on 01/01/2021).

[99] G. C. Functions, "Quotas — cloud functions documentation — google cloud," https://cloud.google.com/functions/quotas, 2021, (Accessed on 03/19/2021).

[100] H. D. Nguyen, Z. Yang, and A. A. Chien, "Motivating high performance serverless workloads," in Proceedings of the 1st Workshop on High Performance Serverless Computing, 2020, pp. 25–32.

[101] N. Pemberton and J. Schleier-Smith, "The serverless data center: Hardware disaggregation meets serverless computing," in Proceedings of the 1st Workshop on Resource Disaggregation, vol. 4, 2019.

[102] A. W. Services, "Aws inferentia - amazon web services (aws)," https://aws.amazon.com/machine-learning/inferentia/, 2021, (Accessed on 11/17/2021).

[103] ——, "Aws trainium - amazon web services (aws)," https://aws.amazon.com/machine-learning/trainium/, 2021, (Accessed on 11/17/2021).

[104] ——, "Amazon elastic inference - amazon web services," https://aws.amazon.com/machine-learning/elastic-inference/, 2021, (Accessed on 11/17/2021).

[105] M. Azure, "Use gpus on azure kubernetes service (aks) - azure kubernetes service — microsoft docs," https://docs.microsoft.com/en-us/azure/aks/gpu-cluster, 2021, (Accessed on 11/17/2021).

[106] ——, "Deploy ml models to fpgas - azure machine learning — microsoft docs," https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service, 2021, (Accessed on 11/17/2021).

[107] G. Cloud, "Cloud tpu — cloud tpu — google cloud," https://cloud.google.com/tpu, 2021, (Accessed on 11/17/2021).

[108] ——, "Edge tpu - run inference at the edge — google cloud," https://cloud.google.com/edge-tpu, 2021, (Accessed on 11/17/2021).

[109] A. W. Services, "Amazon braket quantum computers - amazon web services," https://aws.amazon.com/braket/quantum-computers/, 2021, (Accessed on 11/26/2021).

[110] A. Khandelwal, A. Kejariwal, and K. Ramasamy, "Le taureau: Deconstructing the serverless landscape & a look forward," in Proceedings of the ACM SIGMOD International Conference on Management of Data, 2020, pp. 2641–2650.

[111] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in Proceedings of the International Conference on High Performance Computing & Simulation. IEEE, 2010, pp. 224–231.

[112] D. M. Naranjo, S. Risco, C. de Alfonso, A. Pérez, I. Blanquer, and G. Moltó, "Accelerated serverless computing based on gpu virtualization," Journal of Parallel and Distributed Computing, vol. 139, pp. 32–42, 2020.

[113] B. Ringlein, F. Abel, D. Diamantopoulos, B. Weiss, C. Hagleitner, M. Reichenbach, and D. Fey, "A case for function-as-a-service with disaggregated fpgas," in Proceedings of the IEEE 14th International Conference on Cloud Computing (CLOUD). IEEE Computer Society, 2021.

[114] M. Bacis, R. Brondolin, and M. D. Santambrogio, "Blastfunction: an fpga-as-a-service system for accelerated serverless computing," in Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2020, pp. 852–857.

[115] C. Zhang, M. Yu, W. Wang, and F. Yan, "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in Proceedings of the USENIX Annual Technical Conference, 2019, pp. 1049–1062.

[116] S. Risco and G. Moltó, "Gpu-enabled serverless workflows for efficient multimedia processing," Applied Sciences, vol. 11, no. 4, p. 1438, 2021.

[117] S. S. Gill, "Quantum and blockchain based serverless edge computing: A vision, model, new trends and future directions," Internet Technology Letters, p. e275, 2021.

[118] S. S. Gill, A. Kumar, H. Singh, M. Singh, K. Kaur, M. Usman, and R. Buyya, "Quantum computing: A taxonomy, systematic review and future directions," Software: Practice and Experience, vol. 52, no. 1, pp. 66–114, 2022.

[119] B. Carver, J. Zhang, A. Wang, and Y. Cheng, "In search of a fast and efficient serverless dag engine," in Proceedings of the IEEE/ACM 4th International Parallel Data Systems Workshop (PDSW). IEEE, 2019, pp. 1–10.

[120] Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, and A. Y. Zomaya, "Automated fine-grained cpu cap control in serverless computing platform," IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 10, pp. 2289–2301, 2020.

[121] M. HoseinyFarahabady, Y. C. Lee, A. Y. Zomaya, and 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.

[122] G. Aumala, E. Boza, L. Ortiz-Avilés, G. Totoy, and 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.

[123] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Centralized core-granular scheduling for serverless functions," in Proceedings of the ACM Symposium on Cloud Computing, 2019, pp. 158–164.

[124] M. Zhang, C. Krintz, and R. Wolski, "Edge-adaptable serverless acceleration for machine learning internet of things applications," Software: Practice and Experience, vol. 51, no. 9, pp. 1852–1867, 2021.

[125] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: semantics for stateful serverless," Proceedings of the ACM on Programming Languages, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[126] I. Cloud, "Ibm cloud docs," https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-pkg_composer, 2021, (Accessed on 11/19/2021).

[127] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in Proceedings of the USENIX Annual Technical Conference, 2020, pp. 419–433.

[128] C. Cicconetti, M. Conti, and A. Passarella, "On realizing stateful faas in serverless edge networks: State propagation," in Proceedings of the IEEE International Conference on Smart Computing (SMARTCOMP). IEEE, 2021, pp. 89–96.

[129] Z. Jia and E. Witchel, "Boki: Stateful serverless computing with shared logs," in Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM, 2021, pp. 691–707.

[130] Y. Lee and S. Choi, "A greedy load balancing algorithm on serverless platforms maximizing locality."

[131] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 152–166.

[132] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, "A fault-tolerance shim for serverless computing," in Proceedings of the 15th European Conference on Computer Systems, 2020, pp. 1–15.

[133] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, 2020, pp. 1187–1204.

[134] A. Kanso and A. Youssef, "Serverless: beyond the cloud," in Proceedings of the 2nd International Workshop on Serverless Computing, 2017, pp. 6–10.

[135] A. Poth, N. Schubert, and A. Riel, "Sustainability efficiency challenges of modern it architectures–a quality model for serverless energy footprint," in Proceedings of the European Conference on Software Process Improvement. Springer, 2020, pp. 289–301.

[136] Z. Al-Ali, S. Goodarzy, E. Hunter, S. Ha, R. Han, E. Keller, and E. Rozner, "Making serverless computing more serverless," in Proceedings of the 11th International Conference on Cloud Computing (CLOUD). IEEE, 2018, pp. 456–459.

[137] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling quality-of-service in serverless computing," in Proceedings of the 11th ACM Symposium on Cloud Computing, 2020, pp. 311–327.

[138] V. M. Bhasi, J. R. Gunasekaran, P. Thinakaran, C. S. Mishra, M. T. Kandemir, and C. Das, "Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms," in Proceedings of the ACM Symposium on Cloud Computing, 2021, pp. 153–167.

[139] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," Future Generation Computer Systems, vol. 114, pp. 259–271, 2021.

[140] G. C. Functions, "Cloud functions — google cloud," https://cloud.google.com/functions/, 2021, (Accessed on 01/01/2021).

[141] A. OpenWhisk, "Documentation," https://openwhisk.apache.org/documentation.html, 2021, (Accessed on 01/01/2021).

[142] Kubeless, "Kubeless," https://kubeless.io/, 2021, (Accessed on 11/22/2021).

[143] Prometheus, "Overview — prometheus," https://prometheus.io/docs/introduction/overview/, 2021, (Accessed on 11/25/2021).

[144] "Quotas — cloud functions documentation — google cloud," https://cloud.google.com/functions/quotas, (Accessed on 08/31/2020).

[145] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in Proceedings of the 2017 Symposium on Cloud Computing, 2017, pp. 445–451.

[146] "Deploy services to a swarm — docker documentation," https://docs.docker.com/engine/swarm/services/, (Accessed on 08/31/2020).

[147] A. Singhvi, K. Houck, A. Balasubramanian, M. D. Shaikh, S. Venkataraman, and A. Akella, "Archipelago: A scalable low-latency serverless platform," arXiv preprint arXiv:1911.09849, 2019.

[148] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "Containercloudsim: An environment for modeling and simulation of containers in cloud data centers," Software: Practice and Experience, vol. 47, no. 4, pp. 505–521, 2017.

[149] "Runtime options with memory, cpus, and gpus — docker documentation," https://docs.docker.com/config/containers/resource_constraints/, (Accessed on 10/16/2020).

[150] "Apache openwhisk is a serverless, open source cloud platform," https://openwhisk.apache.org/, (Accessed on 11/23/2020).

[151] "docker update — docker documentation," https://docs.docker.com/engine/reference/commandline/update/, (Accessed on 09/03/2020).

[152] "vbootcamp_performance_benchmark.pdf," https://www.cisco.com/c/dam/global/da_dk/assets/docs/presentations/vBootcamp_Performance_Benchmark.pdf, (Accessed on 10/01/2020).

[153] Z. Zhong, J. He, M. A. Rodriguez, S. Erfani, R. Kotagiri, and R. Buyya, "Heterogeneous task co-location in containerized cloud computing environments," in Proceedings of the IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC). IEEE, 2020, pp. 79–88.

[154] "Wikipedia access traces — wikibench," http://www.wikibench.eu/?page_id=60, (Accessed on 12/02/2020).

[155] OpenFaaS, "Home — openfaas - serverless functions made simple," https://www.openfaas.com/, 2022, (Accessed on 04/08/2022).

[156] T. K. Authors, "Home - knative," https://knative.dev/docs/, 2022, (Accessed on 04/08/2022).

[157] F. Project, "Fission," https://fission.io/, 2022, (Accessed on 04/08/2022).

[158] Kubernetes, "Kubernetes," https://kubernetes.io/, 2022, (Accessed on 04/08/2022).

[159] J. Scheuner, S. Eismann, S. Talluri, E. Van Eyk, C. Abad, P. Leitner, and A. Iosup, "Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications," arXiv preprint arXiv:2205.07696, 2022.

[160] A. Fuerst and P. Sharma, "Locality-aware load-balancing for serverless clusters," in Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing, 2022, p. 227–239.

[161] P. Zuk, B. Przybylski, and K. Rzadca, "Call scheduling to reduce response time of a faas system," in Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2022, pp. 172–182.

[162] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: principled and practical scheduling for serverless functions," in Proceedings of the 13th Symposium on Cloud Computing, 2022, pp. 289–305.

[163] H. Tian, S. Li, A. Wang, W. Wang, T. Wu, and 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.

[164] V. M. Bhasi, J. R. Gunasekaran, A. Sharma, M. T. Kandemir, and 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.

[165] S. Ristov and P. Gritsch, "Faast: Optimize makespan of serverless workflows in federated commercial faas," in Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2022, pp. 183–194.

[166] A. Asghari, M. K. Sohrabi, and F. Yaghmaee, "A cloud resource management framework for multiple online scientific workflows using cooperative reinforcement learning agents," Computer Networks, vol. 179, p. 107340, 2020.

[167] ——, "Task scheduling, resource provisioning, and load balancing on scientific workflows using parallel sarsa reinforcement learning agents and genetic algorithm," The Journal of Supercomputing, vol. 77, no. 3, pp. 2800–2828, 2021.

[168] Y. Wang, H. Liu, W. Zheng, Y. Xia, Y. Li, P. Chen, K. Guo, and H. Xie, "Multi-objective workflow scheduling with deep-q-network-based multi-agent reinforcement learning," IEEE access, vol. 7, pp. 39 974–39 982, 2019.

[169] Y. Qin, H. Wang, S. Yi, X. Li, and L. Zhai, "An energy-aware scheduling algorithm for budget-constrained scientific workflows based on multi-objective reinforcement learning," The Journal of Supercomputing, vol. 76, no. 1, pp. 455–480, 2020.

[170] Z. Peng, D. Cui, J. Zuo, Q. Li, B. Xu, and W. Lin, "Random task scheduling scheme based on reinforcement learning in cloud computing," Cluster computing, vol. 18, no. 4, pp. 1595–1607, 2015.

[171] S. Agarwal, M. A. Rodriguez, and 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.

[172] H. Qiu, W. Mao, A. Patke, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and 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.

[173] H. Yu, A. A. Irissappane, H. Wang, and 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.

[174] P.-H. Chiang, H.-K. Yang, Z.-W. Hong, and C.-Y. Lee, "Mixture of step returns in bootstrapped dqn," arXiv preprint arXiv:2007.08229, 2020.

[175] MRC, "Melbourne research cloud documentation," https://docs.cloud.unimelb.edu.au/, 2022, (Accessed on 07/22/2022).

[176] ARDC, "Ardc nectar research cloud - ardc," https://ardc.edu.au/services/nectar-research-cloud/, 2022, (Accessed on 07/22/2022).

[177] A. S. Foundation, "Apache couchdb," https://couchdb.apache.org/, 2022, (Accessed on 07/22/2022).

[178] Prometheus, "Prometheus - monitoring system & time series database," https://prometheus.io/, 2022, (Accessed on 09/08/2022).

[179] A. S. Foundation, "Apache jmeter - apache jmeter™," https://jmeter.apache.org/, 2021, (Accessed on 11/08/2021).

[180] AWS, "Aws pricing calculator," https://calculator.aws/#/addService/EC2, 2022, (Accessed on 07/25/2022).

[181] A. Kuriata and 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.

[182] R. C. Chiang, "Contention-aware container placement strategy for docker swarm with machine learning based clustering algorithms," Cluster Computing, pp. 1–11, 2020.

[183] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion). IEEE, 2018, pp. 181–188.

[184] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in Proceedings of the International Conference on Omni-layer Intelligent Systems (COINS). IEEE, 2020, pp. 1–7.

[185] S. K. Mohanty, G. Premsankar, M. Di Francesco et al., "An evaluation of open source serverless computing frameworks." CloudCom, vol. 2018, pp. 115–120, 2018.

[186] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," arXiv preprint arXiv:1903.12221, 2019.

[187] K. Suo, J. Son, D. Cheng, W. Chen, and S. Baidya, "Tackling cold start of serverless applications by efficient and adaptive container runtime reusing," in Proceedings of the 2021 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2021, pp. 433–443.

[188] X. Li, P. Kang, J. Molone, W. Wang, and P. Lama, "Kneescale: Efficient resource scaling for serverless computing at the edge," in Proceedings of the 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid). IEEE, 2022, pp. 180–189.

[189] H.-D. Phung and Y. Kim, "A prediction based autoscaling in serverless computing," in Proceedings of the 13th International Conference on Information and Communication Technology Convergence (ICTC). IEEE, 2022, pp. 763–766.

[190] A. Zafeiropoulos, E. Fotopoulou, N. Filinis, and S. Papavassiliou, "Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms," Simulation Modelling Practice and Theory, vol. 116, p. 102461, 2022.

[191] P. Benedetti, M. Femminella, G. Reali, and K. Steenhaut, "Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications," in Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops). IEEE, 2022, pp. 674–679.

[192] P. Vahidinia, B. Farahani, and F. S. Aliee, "Mitigating cold start problem in serverless computing: A reinforcement learning approach," IEEE Internet of Things Journal, 2022.

[193] Z. Zhang, T. Wang, A. Li, and W. Zhang, "Adaptive auto-scaling of delay-sensitive serverless services with reinforcement learning," in Proceedings of the IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE, 2022, pp. 866–871.

[194] C. K. Dehury, S. Poojara, and S. N. Srirama, "Def-drel: Systematic deployment of serverless functions in fog and cloud environments using deep reinforcement learning," arXiv preprint arXiv:2110.15702, 2021.

[195] Q. Tang, R. Xie, F. R. Yu, T. Chen, R. Zhang, T. Huang, and Y. Liu, "Distributed task scheduling in serverless edge computing networks for the internet of things: A learning approach," IEEE Internet of Things Journal, 2022.

[196] X. Yao, N. Chen, X. Yuan, and P. Ou, "Performance optimization of serverless edge computing function offloading based on deep reinforcement learning," Future Generation Computer Systems, vol. 139, pp. 74–86, 2023.

[197] H. Jeon, S. Shin, C. Cho, and S. Yoon, "Deep reinforcement learning for qos-aware package caching in serverless edge computing," in Proceedings of the IEEE Global Communications Conference (GLOBECOM). IEEE, 2021, pp. 1–6.

[198] A. Mampage, S. Karunasekera, and R. Buyya, "Deep reinforcement learning for application scheduling in resource-constrained, multi-tenant serverless computing environments," Future Generation Computer Systems, vol. 143, pp. 277–292, 2023.

[199] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in Proceedings of the International conference on machine learning. PMLR, 2016, pp. 1928–1937.

[200] Y. Tang and S. Agrawal, "Discretizing continuous action space for on-policy optimization," in Proceedings of the aaai conference on artificial intelligence, vol. 34, no. 04, 2020, pp. 5981–5988.

[201] Knative, "About autoscaling - knative," https://knative.dev/docs/serving/autoscaling/, 2023, (Accessed on 04/18/2023).

[202] "Autoscaling - openfaas," https://docs.openfaas.com/architecture/autoscaling/, (Accessed on 04/19/2023).

[203] R. B. Roy, T. Patel, V. Gadepally, and D. Tiwari, "Mashup: making serverless computing useful for hpc workflows via hybrid execution," in Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2022, pp. 46–60.

[204] J. H. Novak, S. K. Kasera, and R. Stutsman, "Cloud functions for fast and robust resource auto-scaling," in Proceedings of the 11th International Conference on Communication Systems & Networks (COMSNETS). IEEE, 2019, pp. 133–140.

[205] S. Horovitz, R. Amos, O. Baruch, T. Cohen, T. Oyar, and A. Deri, "Faastest-machine learning based cost and performance faas optimization," in Economics

of Grids, Clouds, Systems, and Services: 15th International Conference, GECON 2018, Pisa, Italy, September 18–20, 2018, Proceedings 15. Springer, 2019, pp. 171–186.

[206] A. Reuter, T. Back, and V. Andrikopoulos, "Cost efficiency under mixed serverless and serverful deployments," in Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, 2020, pp. 242–245.

[207] "Ec2 on-demand instance pricing – amazon web services," https://aws.amazon.com/ec2/pricing/on-demand/, (Accessed on 07/07/2023).

[208] "Instance auto scaling - amazon ec2 autoscaling - aws," https://aws.amazon.com/ec2/autoscaling/, (Accessed on 07/07/2023).

[209] Z. Fan, R. Su, W. Zhang, and Y. Yu, "Hybrid actor-critic reinforcement learning in parameterized action space," in Proceedings of the 28th International Joint Conference on Artificial Intelligence, 2019, pp. 2279–2285.