Serverless Auto-scaling mechanism using Reinforcement learning

MSc Research Project
Cloud Computing

Vikrant Sonawane
Student ID: 21210403

School of Computing
National College of Ireland

Supervisor: Rashid Mijumbi

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Vikrant Sonawane
Student ID:	21210403
Programme:	Cloud Computing
Year:	2023
Module:	MSc Research Project
Supervisor:	Rashid Mijumbi
Submission Due Date:	14/08/2023
Project Title:	Serverless Auto-scaling mechanism using Reinforcement learning
Word Count:	23445
Page Count:	19

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	14th August 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Serverless Auto-scaling mechanism using Reinforcement learning

Vikrant Sonawane
21210403

Abstract

Serverless computing represents a paradigm of cloud computing that caters to the dynamic needs of users by providing computing resources on an as-needed basis, with charges being levied based on the actual usage of these resources. Infrastructure management is handled by cloud providers, thereby enabling developers to concentrate on business logic development. However, scalability management necessitates the optimization of resource provisioning based on workload, which can prove to be a daunting task and may impose administrative overhead. In order to overcome the challenges that arise when managing scalability for serverless applications, this paper delves into the use of Reinforcement Learning (RL) techniques for autoscaling mechanisms. To manage dynamic workloads while ensuring Quality of autonomous Service (QoS) guarantees and optimizing resource utilization, RL environments and agents are employed, utilizing Q-learning algorithms. Nonetheless, Q-learning algorithms are not without their constraints, such as the overestimation of action values and delays in training and action enforcement. To surmount these limitations, this research proposes the use of Double Q-learning as an alternative solution. The research is driven by a dual motivation. Firstly, it aims to assess the effectiveness of the Double Q-learning algorithm and investigate the feasibility of reducing enforcement time. Secondly, it aims to evaluate the performance of the developed agents in dealing with new workloads comprising multiple serverless applications and cloud services. The proposed mechanisms undergo evaluation in both real and simulated environments, leveraging the Knative open-source serverless platform. Their efficacy in efficiently managing scalability for serverless applications is subsequently validated.

1 Introduction

Serverless computing platforms assume responsibility for the majority of system administration tasks that are required for deploying workloads on the cloud. This results in numerous advantages such as enhanced resource utilization, potential cost savings, and simplified application development. Despite the fact that serverless functions exhibit faster startup and scaling when compared to traditional virtual machines, their performance metrics can still demonstrate variability, which may be unsuitable for customer-facing products.Eismann et al. (2021)

Current offerings in serverless computing exhibit a dearth of workload awareness, as they apply uniform policies to all functions, leading to underutilization of opportunities

for cost optimization, energy conservation, and performance enhancement. The establishment of an analytical model for performance evaluation could facilitate rapid capacity planning and system analysis, thereby obviating the need for costly experiments, and conferring advantages upon both application developers and serverless operators.Jonas et al. (2019) Hence, the primary objective is to scrutinize the efficiency of serverless computing platforms and discern methodologies to augment their efficiency.

The implementation of precise performance modeling for serverless computing platforms can contribute significantly to the preservation of satisfactory levels of quality of service, performance metrics, and workload costs. Moreover, it can offer valuable assistance to providers in optimizing their management strategies for different workloads, ultimately leading to a reduction in infrastructure and energy costs. The Cloud Native Computing Foundation characterizes serverless computing as a contemporary approach to cloud-native computing that prioritizes the development of business logic, its encapsulation in containers, and strict adherence to microservices-based software development principles.Eismann et al. (2021) The serverless platform created by a cloud provider undertakes the management of servers, encompassing activities such as provisioning and scaling. This is a noteworthy facet of the cloud provider's services.

One of the primary attributes of serverless computing is its ability to dynamically scale with minimal intervention from developers. The serverless framework efficiently makes real-time decisions on horizontal scaling efforts based on the varying workload of different components within the application.Pimpley et al. (2021) This methodology is notably advantageous in dealing with sporadic workloads, and can similarly be implemented in diverse application contexts. This approach is particularly advantageous for handling irregular workloads and can be applied in various application contexts as well. Nevertheless, these strategies do entail an administrative burden on the cloud provider, given that appropriate configuration of the serverless platform remains a requisite.

Cloud providers face a set of challenges in effectively managing dynamic and bursty workloads.Zafeiropoulos et al. (2022)Manual approaches to scaling management can lead to administrative burdens, resource inefficiencies, and breaches of resource restrictions. To govern fluctuating and abrupt workloads effectively, it is imperative to have agility and automation. The capacity to scale resources on an as-needed basis, without necessitating advanced application-level expertise, is indispensable. Swift reaction times, encompassing the deployment of new instances and the prompt determination of whether to provision or de-provision instances, assume utmost significance in this context.

Identifying the optimal conditions of equilibrium that stimulate elasticity actions presents an additional challenge. The objective of this equilibrium is twofold: firstly, to decrease the number of Service Level Agreement (SLA) violations between the cloud provider and the application provider and secondly, to optimize the utilization of computing resources in order to minimize deployment and operation costs of the serverless application. Striking the appropriate balance between these two objectives is a fundamental challenge in the management of serverless applications in a manner that is both cost-effective and compliant with SLAs.Lin and Khazaei (2021a) The goals of minimizing SLA violations and optimizing resource usage can at times be contradictory. Therefore, it is essential to strike a balance between the two objectives to fully realize the benefits

of adopting the serverless computing paradigm over other methods. Efficiently managing this trade-off can result in enhanced performance, reduced costs, and improved effectiveness in utilizing serverless computing resources.

Reinforcement Learning (RL) has the ability to autonomously solve decision-making problems in complex and ambiguous environments.Garí et al. (2020) The use of Reinforcement Learning (RL) is viewed as a promising technique to manage autoscaling in cloud environments, in contrast to conventional rule-based methodologies that are less dynamic and rely on the evaluation of time-series data. RL provides adaptive scaling policies that ensure Quality of Service (QoS) satisfaction, even when confronted with various performance-related issues. An advantage of RL is that it obviates the need for human intervention, as policy learning is accomplished through interaction with the environment.Kardani-Moghaddam et al. (2021) This facilitates the ability to be dynamic and adaptive, as the process of learning is ongoing and the policies generated have the capability to be modified in response to alterations that arise in the cloud environment.

Double Q-learning is a widely adopted approach in the field of reinforcement learning, which is often utilized to facilitate decision making in a complex environment that involves uncertain rewards. The fundamental principle that underpins Double Q-learning is to overcome the potential problem of action value overestimation, which is a common challenge encountered in standard Q-learning. van Hasselt et al. (2015) Within the Q-learning algorithm, the agent derives the value of each action through a procedural approach which involves assessing the maximum Q-value that corresponds to the next state. Nonetheless, this approach may result in overconfident approximations, whereby the agent may opt for actions that possess higher estimated Q-values, regardless of whether they are the most optimal actions to undertake.

Double Q-learning is a strategy that resolves this issue through the maintenance of two distinct sets of Q-values, colloquially referred to as the "online" and "target" Q-values. The former Q-value set is utilized by the agent to determine its actions, while the latter is employed to appraise the efficacy of these actions.van Hasselt et al. (2015) The avoidance of overestimation of action values and the consequent improvement in decision-making accuracy are key benefits of the approach adopted by the agent. In the Double Q-learning algorithm, the learning process involves alternating between two distinct steps, namely, action selection based on online Q-values, and value estimation based on target Q-values. The incorporation of this two-step approach is particularly advantageous in minimizing the overestimation of action values, resulting in more effective learning and decision-making by the RL agent.

The principal contribution of this study pertains to the establishment of a modeling framework designed for RL environments, which effectively employs a Double Q-learning algorithm to oversee the administration of autoscaling mechanisms in the context of serverless computing platforms. This achievement holds considerable significance, given the paucity of extant research works that are explicitly geared towards addressing the issue of autoscaling in serverless computing platforms.

2 Related Work

2.1 Serverless computing

Serverless computing, as previously mentioned, constitutes a cloud computing paradigm in which software developers can author and publish code without needing to oversee the foundational infrastructure. Within a serverless architecture, the cloud provider assumes responsibility for automatically allocating and overseeing the resources necessary to execute the code and assesses fees predicated upon the actual usage, usually quantified by the number of function invocations or the duration of execution. Jonas et al. (2019) This feature enables software developers to concentrate exclusively on software development and application construction without being concerned about the management of infrastructure.

Serverless computing offers a plethora of advantages, encompassing expedited development cycles, curtailed operational overhead, and financial savings predicated on pay-per use pricing. This methodology is especially suitable for event-triggered, stateless applications with fluctuating workloads and indeterminate resource requisites. Nevertheless, it also entails certain drawbacks, including the possibility of vendor lock-in, restricted customization alternatives, and deliberations regarding data privacy and security.

2.2 Autoscaling in Serverless computing

Serverless computing platforms effectively manage a vast majority of the system administration tasks necessary for deploying workloads on the cloud. Such platforms offer users an array of benefits that include optimized system administration operations, enhanced resource utilization, potential cost reductions, heightened energy efficiency, and simplified application development.Mahmoudi and Khazaei (2022) Although cloud functions have rapid deployment and scalability capabilities compared to conventional VM-based instances, they still exhibit unpredictability in their essential performance metrics, which may be unsatisfactory for products intended for customer interaction. A serverless strategy is often an apt alternative for ephemeral and unencumbered applications that are disseminated in character and can manage asynchronous workloads that are divisible into autonomous assignments that can be executed in parallel. CNCF (2023). Serverless applications are frequently utilized for tasks with a short duration, low data volume, and bursty workloads. Moreover, they are commonly employed for crucial functions that necessitate low latency and high-volume processing. Analysis of 89 serverless applications at Pujol Roig et al. (2020) reveals that 84% of these applications serve bursty workloads, 82% consist of five functions or fewer, and 93% consists of 10 functions or fewer Eismann et al. (2021).

Developing precise performance models for serverless computing platforms is crucial in guaranteeing that the quality of service, performance metrics, and workload costs are maintained within acceptable limits. Furthermore, it can be advantageous for providers as it allows them to fine-tune their management strategies for each workload, which may result in lower infrastructure and energy expenses.Lin and Khazaei (2021b) The performance model employed to address performance-related issues in serverless computing platforms ought to be both controllable and comprehensive, inclusive of a diverse range of system parameters. As far as our understanding extends, no such performance model has yet been presented for contemporary serverless computing platforms.Mahmoudi and

Khazaei (2022)

The utilization of various techniques in serverless computing, including per-request scaling, concurrency-based scaling, and metrics-driven scaling, has been observed to facilitate the handling of scaling operations of functions based on diverse criteria. The scale per request pattern responds to the number of incoming requests, but may encounter challenges related to the cold start of functions and the absence of queuing of new requests. Concurrency value scaling, on the other hand, enables the processing of multiple requests concurrently, but has a predefined upper limit. Metrics-based scaling, which aims to ensure the preservation of key performance metrics, such as CPU usage, memory usage, throughput, or latency, within a predetermined range, has also been designed.Zafeiropoulos et al. (2022) While metrics-based scaling may not be the most optimal selection for workloads that exhibit sporadic bursts in terms of performance, it does consider various performance, resource, and cost indicators. This feature renders it a compelling option for cloud providers to proficiently manage scaling actions.

There exist several proposed mechanisms for achieving autoscaling in the context of serverless computing. These mechanisms can be categorized into two main approaches, namely static and dynamic approaches. In the case of static approaches, scaling rules are defined based on thresholds and time-series analysis. However, updating such rules requires manual intervention, which can be quite cumbersome for microservices-based applications with numerous functions. In contrast, dynamic approaches rely on machine learning techniques to improve the automation of the scaling process. This method exhibits promise in enhancing scalability and reducing administrative overhead.Zafeiropoulos et al. (2022) In Zafeiropoulos et al. (2022); Schuler et al. (2020) these problems are tackled using RL techniques. The approach presented in Zafeiropoulos et al. (2022) regarding the autoscaling of functions in serverless computing platforms involves implementation through the use of reinforcement learning (RL) agents, which employ decision-making processes.

2.3 Auto-scaling using Reinforcement Learning

The topic of scaling and scheduling in cloud autoscaling has been thoroughly explored in the literature of Garí et al. (2020). This area is characterized by stochastic environments and involves adjusting the number of virtual machines (VMs) in the virtual infrastructure, as well as assigning tasks to specific VMs. As a result of the inherent uncertainty in these subproblems, researchers have proposed modeling the autoscaling problem as a Markov Decision Process (MDP) and utilizing various Reinforcement Learning (RL) techniques to develop effective scaling or scheduling policies. These policies enable an autoscaler to dynamically determine the most suitable action at any given time to optimize a long-term objective.

The comprehensive exploration of auto-scaling through the utilization of Reinforcement Learning on platforms without server-less services is also extensively examined. Xiao and Hu (2022) has developed an automated scaling framework known as "Dscaler". Experimental evaluations of Dscaler's effectiveness in scaling microservices in Kubernetes have revealed that our proposed approach results in a significant reduction of resource usage by 19.90% and 10.80% in two distinct workloads. Moreover, the proposed method

also effectively mitigates SLA violations by 8.56% and 12.75% in the respective workloads, when compared to the Horizontal Pod Autoscaler (HPA). Santos et al. (2023), on the other hand, use Advantage Actor-Critic (A2C) and Recurrent The implementation of Proximal Policy Optimization (RPPO) has been demonstrated in literature as a means to inform autoscaling decisions. Empirical evidence indicates that Reinforcement Learning frameworks exceed threshold-based HPA mechanisms in terms of performance. Consequently, RL presents the potential to enhance the quality of auto-scaling decisions.

Reinforcement Learning (RL) constitutes a distinct subset of Artificial Intelligence (AI) algorithms that pivot around the fundamental concepts of environment, agents, and rewards. Garí et al. (2020). The agent engages in interactions with its surrounding environment by executing actions that have the potential to affect both its own state and that of the environment. The ultimate objective is to accomplish a specific goal. As a result of each action, the agent is subject to either rewards or penalties, contingent upon the extent to which the action aligns with the goal. By amassing information on a series of actions, the agent acquires knowledge that it uses to refine policies with the aim of maximizing the cumulative reward over time.

Reinforcement Learning (RL) has demonstrated impressive competencies in addressing decision-making challenges in intricate and indeterminate environments. Specifically, in the domain of cloud autoscaling management, RL is deemed a promising approach when compared to conventional static rule-based techniques that rely on the analysis of time-series data. RL provides the adaptability to modify scaling policies to ensure the attainment of Quality of Service (QoS) satisfaction in the presence of diverse performance-related predicaments.Kardani-Moghaddam et al. (2021) One of the advantages of Reinforcement Learning (RL) is its independence from human intervention in policy learning. This is due to the continuous learning process and the capability to adjust policies in accordance with changes in the cloud environment.Kardani-Moghaddam et al. (2021); Garí et al. (2020) The dynamic and adaptive nature of Reinforcement Learning renders it highly suitable for effectively managing autoscaling within the cloud environment.

The focus of the study discussed in Zafeiropoulos et al. (2022) is on the development and execution of unobstructed and compatible environments for reinforcement learning (RL). This is aimed at promoting the horizontal expansion of serverless applications. The utilization of these settings is feasible in both artificial and factual serverless computing environments. The investigation executed a series of RL settings and agents that were constructed on the basis of Q-learning, DynaQ+, and Deep Q-learning algorithms. While Q-learning algorithms were implemented on genuine environments, the others were applied to a combination of authentic and simulated environments. These RL agents were purposefully designed to steer autoscaling mechanisms that can independently operate dynamic workloads, ensuring Quality of Service (QoS) guarantees, while concurrently optimizing resource efficiency.

Nonetheless, certain deficiencies have been ascertained in the execution. The execution of serverless workloads is restricted to microservices' HTTP payloads and CPU-intensive applications. This may culminate in substandard performance owing to overfitting, suboptimal policies, and the arduousness of handling infrequent occurrences. Furthermore, it is worth noting that the Q-learning algorithm, as outlined in the publication

by Zafeiropoulos et al. (2022), can be further optimized by incorporating the Double Q-learning approach mentioned previously.

2.4 Proposal to overcome limitations

The work of Garí et al. (2020) presents a taxonomy of three distinct applications, which are workflows, independent tasks, and cloud services. Through the development of new workloads with varying workloads, there is potential to improve the modeled data and actions. These applications, specifically the utilization of multiple serverless functions interconnected in a graph or linked to other cloud services, as well as internal queues, can be effectively employed to train the environments. Such workloads can offer a diverse range of data, which is essential for enhancing the algorithm's performance.

The article by Hasselt (2010) introduces the innovative concept of Dual Q-learning, which utilizes a dual estimator approach to accurately estimate the value of the next state. Significantly, this represents the initial off-policy value-based reinforcement learning algorithm that effectively mitigates the positive bias that occurs in estimating action values in stochastic environments. Upon analysis, it was discovered that while Double Q-learning may occasionally underestimate action values, it does not suffer from the overestimation bias observed in Q-learning. The experimental results of the roulette game and maze problem demonstrate that Double Q-learning exhibits swifter convergence to good performance levels than Q-learning.

The study conducted by Google, as presented in van Hasselt et al. (2015), posits that the Q-learning algorithm, which is widely used, tends to overestimate action values in certain situations. Nonetheless, it remained ambiguous whether such overestimations were prevalent, had a negative impact on performance, and were avoidable. This research endeavors to respond to these inquiries, and validates the occurrence of overestimations in practical settings. Specifically, the DQN algorithm, which merges Q-learning with a deep neural network, was discovered to have significant overestimations in some Atari 2600 games. The article also proposes a modification to the DQN algorithm, founded on the Double Q-learning concept, initially developed for tabular settings, to handle large-scale function approximation. The proposed adaptation mitigates overestimation and enhances performance in various games.

The work by the authors in Kim et al. (2022) features an enhanced rendition of DQN (Deep Q Network) which utilizes the Markov Decision Process (MDP) algorithm, referred to as Double DQN. This approach effectively addresses the overestimation challenges encountered in the original DQN algorithm and presents superior performance.

2.5 Reinforcement Learning Environment

The environments are comprised of two distinct components. The initial component is the monitoring system, which provides real-time information on resource utilization, service limits, deployment status, number of services, and Quality of Service (QoS). The second component, known as the agent, processes this information and takes appropriate action. The actions taken by the agent include provisioning or scaling up based on the threshold value as outlined in the SLA agreement and similarly de-provisioning if the resources are

no longer necessary. The primary goal of this system is to reduce the number of services as much as possible without negatively impacting the SLA. A reward is generated based on the results, with penalties applied if the SLA is not honored. The RL agent is designed with the objective of maximizing the number of rewards obtained. Subsequently, the resulting action-state information is relayed back to the agent. This particular modus operandi is referred to as the on-policy approach, which will be employed in the upcoming implementations. The diagrammatic representation of the agent's operational framework is depicted in the 2.

In instances where the state is continuous and the action space is discrete, the Double Q-Learning algorithm is implemented. The determination of state involves the consideration of a set of metrics which can impact the decision-making process of the autoscaling policy. These metrics encompass various aspects such as resource usage, quality of service (QoS), configuration elements, and the status of deployed containers. Metrics pertaining to resource usage are related to the CPU and memory usage of the containers. On the other hand, QoS metrics relate to the latency (response time) and throughput (requests served per second) of the service offered. Moreover, information concerning active and terminated containers is also taken into account in the state representation. For a comprehensive list of the metrics utilized in constructing the state in the defined RL (Reinforcement Learning) environments, please refer to Table 1.

2.6 Double Q Learning Algorithm

Q-learning can be understood as utilizing a single estimator to approximate the value of the subsequent state. Specifically, $max_a Q_t(s_{t+1}, a)$ serves as an estimate for $Emax_a Q_t(s_{t+1}, a)$, which, in turn, is an approximation of $max_a\ EQ_t(st + 1, a)$. Here, the expectation should be interpreted as averaging over all potential runs of the same experiment, contrary to its typical usage in reinforcement learning contexts, where it often represents the expectation over the next state (denoted as $E\{|Pt\}$ and encountered in the subsequent subsection).

As a consequence, $max_a Q_t(st + 1, a)$ can be regarded as an unbiased sample drawn from an independent and identically distributed (iid) distribution with a mean of $Emax_a$ $Q_t(st + 1, a)$. However, it has been empirically shown that this approach can lead to significant overestimations, a problem that the following section seeks to address through a proposed algorithm called Double Q-learning.

Double Q-learning utilizes two separate Q functions: Q_A and Q_B. Each Q function is updated using the value from the other Q function for the next state. The action a^* in line 6 represents the action with the highest value in state s‘ according to the value function Q_A. However, instead of using the value $Q_A(s‘, a^*) = max_a Q_A(s‘, a)$ for updating Q_A, as Q-learning does, the value $Q_B(s0, a^*)$ is employed. Since Q_B was updated on the same problem but with a different set of experience samples, it can be considered an unbiased estimate for the value of this action. A similar update is applied to Q_B using b^* and Q_A.

It is crucial for both Q functions to learn from distinct sets of experiences, while during action selection, one can use the value functions of both. Hence, this algorithm is not less data-efficient than Q-learning. In their experiments, the authors computed the average of the two Q values for each action and then performed ϵ-greedy exploration using the resulting average Q values[hasselt paper]. The Double Q learning algorithm is

$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t)(r_t + \gamma max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t))$

The Horizontal Pod Autoscaler (HPA), which functions as a constituent of the Kubernetes orchestrator, is responsible for dynamically altering the number of replicas, also

Algorithm 1 Double Q-learning

Initialize Q_A, Q_B, s
repeat
Choose a, based on $Q^A(s, .)$ and $Q^B(s,)$, observe $r, s`$
Choose (e.g. random) either UPDATE(A) or UPDATE(B)
UPDATE(A) then Define $a^* = argmax_a Q^A(s`, a)$
$Q^A(s, a) Q^A(s, a) + \alpha(s, a)(r + \gamma Q^B(s`, a^*) - Q^A(s, a))$
else if UPDATE(B) then
$Define b^* = argmax_a Q^B(s`, a)$
$Q^B(s, a) \longleftarrow Q^B(s, a) + \gamma(s, a)(r + QA(s`, b^*) - Q^B(s, a))$
end if
$s \longleftarrow s`$
until end

Metrics used in RL environment spaces

Table 1: Cost of Computing

Metric name	Unit	Metric type
CPU Usage	Percentage(%)	Resource usage
Memory Usage	Percentage(%)	Resource usage
CPU Usage Threshold	Percentage(%)	Resource usage
Memory Usage Threshold	Percentage(%)	Resource usage
Active serverless functions	Percentage(%)	Configuration - Status
Terminated serverless function instances	Percentage(%)	Configuration - Status
Latency	Time(ms)	Quality of Service
SLA Latency	Time(ms)	Quality of Service
Throughput Rate	Number/Time	Quality of Service
Success Rate	Time(ms)	Quality of Service

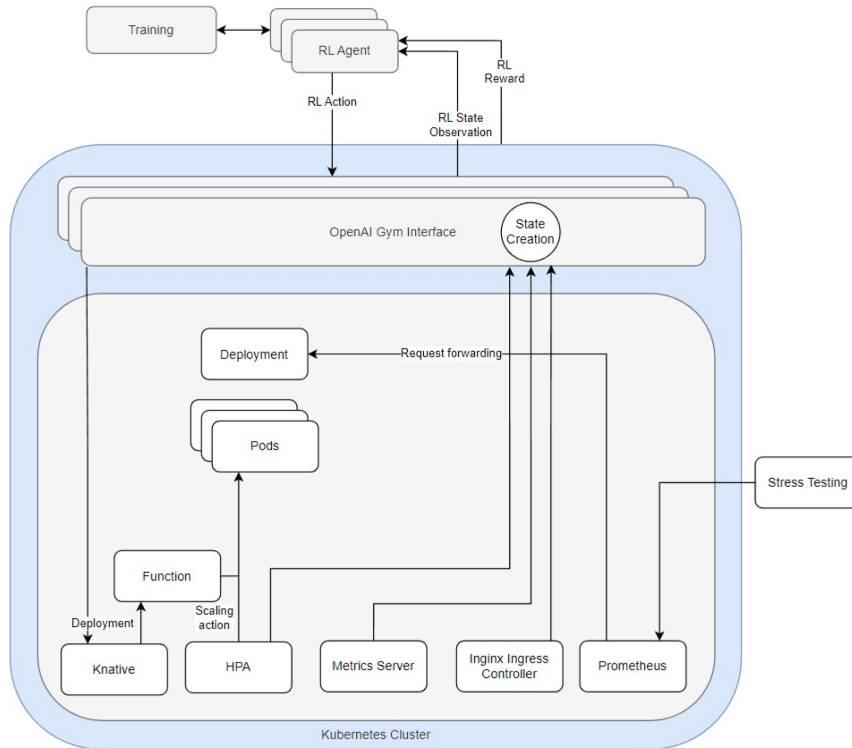

Figure 1: Architecture

known as instances, based on real-time metrics such as CPU utilization, memory utilization, or custom metrics. The establishment of a target threshold for scaling is performed by the system administrator. The Knative serverless platform implements Kubernetes HPA to effectuate the automatic scaling of functions predicated upon defined workload metrics *Horizontal Pod Autoscaling* (n.d.). Within the context of this paper, RL agents are employed to establish scaling targets and mechanize the scaling process. The HPA operates in a control loop, receiving metric information at periodic intervals to make scaling decisions. The determination of the suitable scaling action is predicated upon the calculation of the ratio between the current metric value and the target metric value.

3 Methodology

This research proposes to improve the current state of the art by implementing the Double Q-learning algorithm. Research on the interoperability of the RL agents with the current serverless platform is limited. This research aims to fill this gap by developing interoperable agents on the current serverless platforms. These agents can potentially integrate with serverless platforms and better optimize workloads.

In this research, we plan to implement an environment using OpenAI's Gym interface. This interface will have the Double-Q learning agent. This agent will be trained on feedback received from the environment. The environment consists of a Kubernetes cluster where we have deployed applications using the Knative serverless framework. The application deployed is a demo REST API using Python. Ingress is used as a load balancer for intercepting requests from the server ¡info on ingress¿. Vegeta service will be

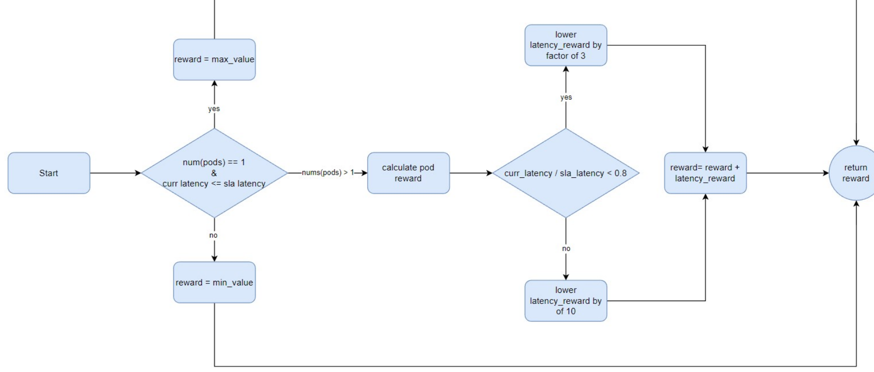

Figure 2: Agent flow diagram

used as a stress testing mechanism which will try to load test the application by sending it multiple requests. The load on the service will be monitored by the Metrics server running on the kubernetes cluster. The feedback from this Metric server will be given to the Agent. The agent will take action on the feedback provided by the metric server. The agent will take action using the HPA service present on the Kubernetes cluster. The high level details of the architecture are provided in 3

In the Kubernetes discrete environment, the state is comprised of four distinct metrics, namely CPU utilization, the applied CPU threshold for autoscaling, the ratio of active instances of the serverless function with the maximum permissible instances, and the ratio of measured latency with the latency defined in the service level agreement (SLA). In order to limit the overall state space, we have considered a set of discretized values per metric. For the ratio of CPU utilization, we have defined seven distinct states. These discrete states correspond to values ranging from 0%–20%, 20%–40%, 40%–60%, 60%–80%, 80%–100%, as well as values that range from 100%–150% or are greater than 150%.2 A similar classification applies to the ratio with the latency metric, wherein we have also defined seven distinct states. The present environment is characterized by 1225 states, which is the product of 7x7x5x5. The set of feasible actions available in this context encompasses three potential alternatives. In light of these actions, the threshold applied to CPU usage for triggering a scaling event can be subjected to a reduction of 20%, kept at the same level, or increased by 20%.

Upon the completion of each action, a reward is bestowed, which is extensively detailed in the rewards flow diagram. The reward function's definition takes into consideration the primary parameters of the serverless function's active instances and the average monitored latency. These parameters are evaluated based on different weightings, with a higher weight (0.7) being assigned to the latency component and a lesser weight (0.3) to the active instances component. The number of active instances is subject to a linear function with a maximum value (100) assigned to one active instance and a minimum value (0) assigned to the maximum number of active instances. Regarding latency, the highest reward is offered when the latency is as close as possible to 0.8 times the SLA_latency.2

The selection of a particular value takes into consideration the necessity to apply a scaling action promptly in response to high latency to prevent SLA violations in bursty workloads. The agent can be trained to effectively handle such situations. Conversely, for small latency, the agent has the flexibility to reduce the number of allocated resources

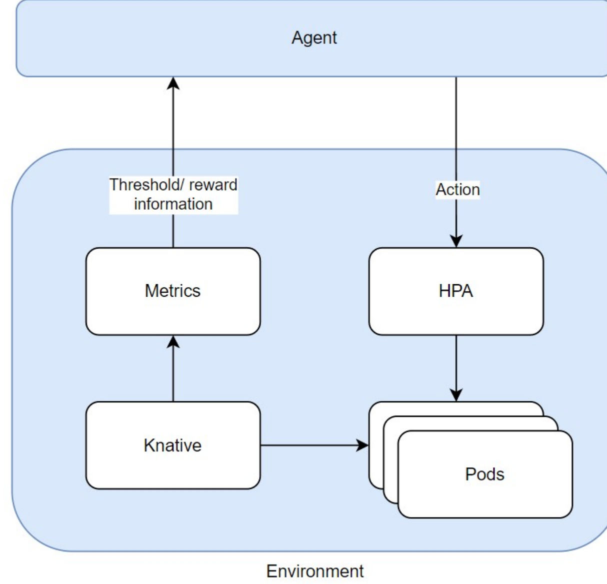

Figure 3: High Level diagram

to avoid high costs while still ensuring the desired QoS level for the application. When the value falls below or exceeds 0.8*SLA_latency, the reward offered is subjected to an exponential reduction, with steeper reductions for values exceeding 0.8*SLA_latency. In case of latency exceeding the SLA_latency, the occurrence is flagged as an SLA violation, leading to zero reward. The autoscaling is achieved with the help of Horizontal Pod Autoscaler.

$$desiredReplicas = ceil[currentReplicas * (currentMetricValue/desiredMetricValue)]$$

The above formula is used by Horizontal Pod Autoscaler to manage the number of replicas. The Horizontal Pod Autoscaler (HPA) is a constituent of the Kubernetes orchestrator which dynamically adjusts the number of replicas (instances) in accordance with real-time metrics such as CPU utilization, memory utilization, or custom metrics. The scaling target is established by the system administrator. Knative serverless platform employs Kubernetes HPA in order to automatically adjust functions in accordance with pre-established workload metrics. Within the context of this paper, RL agents are utilized to establish scaling targets and facilitate the scaling process. The HPA functions within a control loop and periodically receives metric information in order to make scaling decisions. The appropriate scaling action is determined by computing the ratio between the current metric value and the target metric value.

1 presents an inventory of the developed environments. The specifics of each environment are explicated, encompassing the nature of the state and its actions, as well as the metrics that constitute the state and those employed for the purpose of defining the reward function. Moreover, the overall count of the potential states and actions is enumerated, alongside the RL agents that are utilized and the deployment testbeds with which they interact.

4 Design Specification

4.1 Testbed serverless setup for environment

There exist numerous cloud platforms, both open-source and commercial, that can be employed to create a real-world testbed setup. Among these platforms are AWS Lambda, Azure functions, and Google Cloud functions, as well as open-source alternatives such as Nuclio, OpenFaaS, Knative, and Kubeless. Each of these serverless platforms, whether open-source or commercial, has its own set of advantages and limitations. While open-source platforms offer greater customization, flexibility, and cost-effectiveness, they may require more effort in terms of support and maintenance. Commercial platforms, on the other hand, may provide additional features, support, and optimizations, but may come with licensing costs, vendor lock-in concerns, and limitations on customization. In their study, Zafeiropoulos et al. (2022) utilized the Knative platform in conjunction with Kubernetes Horizontal Pod Autoscaler (HPA). The testbed setup is illustrated in Figure 1.

A cloud computing testbed has been established for evaluating the performance of the developed RL agents in specific environments, as illustrated in 1. The testbed combines real environments. For the real environment, a Kubernetes cluster has been set up on three nodes, each equipped with different hardware configurations, including Intel NUC PCs with varying specifications. The Knative serverless computing platform is deployed to manage function deployments on the cluster, and scaling mechanisms are applied using the Kubernetes Horizontal Pod Autoscaler (HPA) for efficient resource management (more about HPA in Section 4.2). The Kubernetes Metrics Server is utilized to collect resource usage and Quality of Service (QoS) metrics, which are then fed into the Prometheus Monitoring Engine functioning as a time series database. For experimentation purposes, a serverless microservice calculating Fibonacci numbers has been developed, and NGINX Ingress Controller acts as the load balancer for this microservice. Stress testing of the serverless functions is performed using the Vegeta HTTP load testing tool, with each serverless function instance deployed within a Pod.

The RL environment is modeled using the OpenAI Gym toolkit as shown in 1. OpenAI Gym facilitates the definition of test problems (environments) for training RL agents. These environments act as intermediaries between the RL agents and the testbed. As depicted in Figure 3, the environment gathers resource and SLA metrics from HPA, the Metrics Server, and Prometheus, presenting them as an observed state to the RL agent. Additionally, it receives the selected action from the RL agent and applies new thresholds in the Kubernetes cluster HPA mechanism accordingly. Furthermore, based on the observed state and the action provided, the environment calculates the reward and provides it as feedback to the agent. To expedite the training process of RL agents, distributed training techniques are employed, enabling parallel deployment of the developed environments and agents. This approach allows for a larger amount of experience, which is stored in the same Q function (Q-table or deep neural network). Prometheus is set up for monitoring the pods running on the Kubernetes cluster. The functions are setup using Knative, Ingress is used to handle the HTTPs requests.

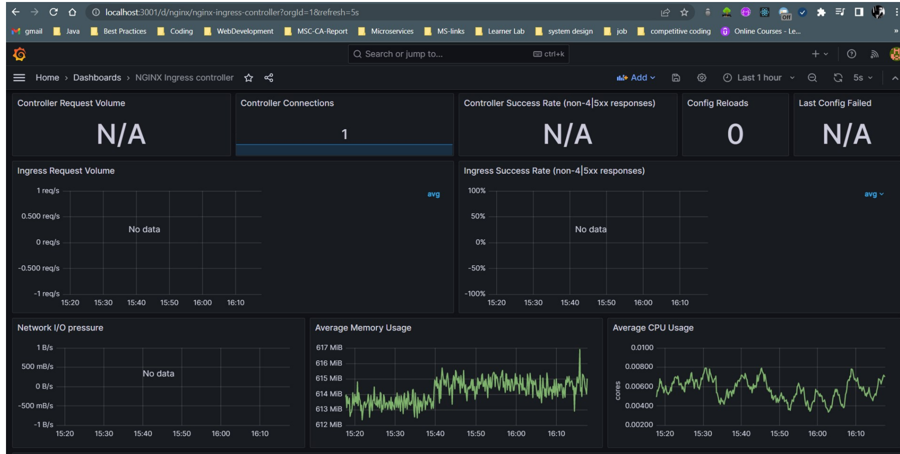

Figure 4: Grafana dashbord showing python app stats

5 Implementation

5.1 Create and configure python demo application

A simple Python application is created with a rest endpoint where the response is returned with a delay. This simulates the processing of a typical web server. All the code is available in (n.d.). A docker file is required to create a container of the python application. Docker containers are lightweight, portable, and self-sufficient units that can encapsulate an application and its dependencies, ensuring consistent behavior across different environments. The container image is published to the docker hub. Docker Hub provides a centralized platform for sharing your container images with others. It's a convenient way to distribute your application or software stack to colleagues, collaborators, or the wider community.

5.2 Setup Kubernetes, Prometheus, and Grafana

Prometheus and Grafana are two popular open-source tools commonly used in the field of monitoring and observability for applications, systems, and infrastructure. They work together to provide comprehensive monitoring, data collection, visualization, and alerting capabilities.
To set up Prometheus, we need to run the prometheus.yml file in kubernetes. The steps are provided in the implementation document. After setup up, we should be able to see the Grafana dashboard as shown in 4

5.3 Setup Knative

Knative is an open-source platform that aims to simplify the deployment, scaling, and management of serverless workloads and microservices on Kubernetes. It provides a set of building blocks and abstractions that enable developers to focus on writing code without having to worry about the underlying infrastructure complexities. Knative is built on top of Kubernetes and extends its capabilities to provide a higher-level abstraction for serverless development. In the context of this research, Knative Serving is required. Knative serving component focuses on automating the deployment, scaling, and routing

```
PS C:\Users\vikrant> kubectl get pods -n knative-serving
NAME                                    READY   STATUS    RESTARTS       AGE
activator-6446987c95-x4qwz              1/1     Running   2 (5h4m ago)   7d1h
autoscaler-55bdcd9565-b6cxl             1/1     Running   2 (5h4m ago)   7d1h
controller-6f44c549c7-5bctz             1/1     Running   2 (5h4m ago)   7d1h
domain-mapping-69d4b566b8-c2mk6         1/1     Running   2 (5h4m ago)   7d1h
domainmapping-webhook-697b9d5dd8-k8hbp  1/1     Running   4 (5h4m ago)   7d1h
net-kourier-controller-6d56cbf854-j6brn 1/1     Running   4 (5h4m ago)   7d1h
webhook-8c895dd9f-j9gtq                 1/1     Running   4 (5h4m ago)   7d1h
```

Figure 5: knative running on Kubernetes

of containerized applications. It introduces concepts like "Services" and "Routes" to define and manage workloads, as well as automatic scaling based on incoming traffic.

For installation, we need to apply serving yml files and create a deployment for the Python application. The detailed steps are provided in the configuration document. After installation, we are able to see the output shown in 5.

5.4 Install Ingress

In Kubernetes, an Ingress is an API object that manages external access to services within a cluster. It provides a way to route and control incoming network traffic to different services based on rules defined in the Ingress resource. In other words, Ingress acts as a traffic controller that allows you to expose and manage HTTP and HTTPS routes from outside the cluster to services inside the cluster.

5.5 Create environment

For creating the environment we need to install Anaconda initially so that we can Jupyter Labs. Creating OpenAI Gym environments involves defining the structure, behavior, and rewards of an environment to model a specific problem or task for reinforcement learning. OpenAI Gym is a toolkit that provides a standardized interface for creating and working with reinforcement learning environments. Here's a general outline of how you can create OpenAI. [the original research] implementation is reproduced as a basis of current process.

5.5.1 Training Reinforcement learning agents

The goal of the agent is to learn how to take action in the environment to maximize its long-term cumulative reward.

5.5.2 Global Variables

The code begins by initializing and declaring global variables (q_table, tau_table, and model) that are used for reinforcement learning. These variables hold information related to the Q-learning algorithm and the agent's knowledge of the environment.

5.5.3 Hyperparameters

Hyperparameters are defined, such as alpha, gamma, kappa, epsilon_init, and epsilon_min. These parameters control the learning rate, discount factor, exploration rate, and other aspects of the agent's learning process.

5.5.4 Environment Initialization

The code initializes an OpenAI Gym environment using the gym.make() function. It appears to be a custom environment that is specified by the gym_env variable. The environment is set up with various parameters, such as timestep_duration, app_name, sla_latency, prometheus_host, and
prometheus_latency_metric_name.

5.5.5 Training Loop

The code enters a loop for training the agent over multiple epochs. Within each epoch, the agent interacts with the environment for a certain number of steps defined by steps_per_epoch.

5.5.6 Resetting the Environment

The environment is reset using the reset() method, which returns the initial state and other information.

5.5.7 Main Loop

Within the training loop, the code enters a nested loop (for step in range(steps_per_epoch):). Within this loop, the agent interacts with the environment by taking action and observing the consequences.

5.5.8 Epsilon-Greedy Exploration

The code employs an epsilon-greedy exploration strategy to balance exploration and exploitation. It randomly selects an action with probability epsilon, encouraging exploration, or selects the action with the highest Q-value with probability 1 - epsilon, encouraging exploitation.

5.5.9 Taking Actions and Observing Outcomes

The agent takes an action based on the chosen strategy. It then receives information about the next state, observations, rewards, whether the episode is done, and additional information from the environment's step() method.

5.5.10 Updating Historical Data

The code updates a historical log (likely for analysis and debugging purposes) with various metrics, observations, and information related to the agent's actions and the environment's state.

5.5.11 Environment Manipulation

It appears that the code also manipulates the environment by starting and stopping a stress test (vegeta_process) on an endpoint, deleting and deploying services, and creating an Ingress. These actions might be part of a custom setup for the reinforcement learning environment.

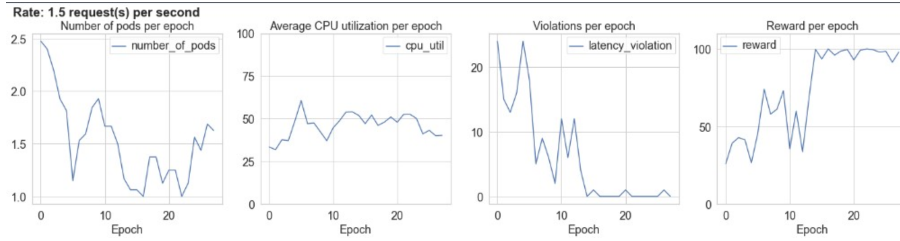

Figure 6: Results for slow requests

5.5.12 Epoch Management

After each epoch, if the episode is done, the environment is reset, and certain actions are taken related to the deployment of services and Ingress.

6 Evaluation

Performance evaluation results have been generated by deploying RL agents within custom-built RL environments. The data collected from various agents was aggregated based on the applied workload, allowing for performance assessment.

6.1 Experiment 1

For the k8s-env-discrete environment and the Q-learning agent, two scenarios were reviewed: one with low and another with high request rates. The learning rates were distinct due to varying SLA violations and rewards. In high request rate conditions, early epochs revealed more SLA breaches, yielding low rewards. As epochs progressed, the agent improved by adjusting thresholds to prevent violations as shown in 6, while optimizing pod numbers and CPU usage, resulting in increased rewards. Low request rates led to a steady reward increase, significantly reducing SLA violations, and slightly elevating pod CPU usage as shown in 7.

In the same k8s-env-discrete environment, the Double Q learning agent's results were explored. In low request rates, Double Q was theorized to swiftly reach an optimal policy with high rewards and minimal SLA breaches, showing minor pod and CPU fluctuations. But the result is that we can use the Double Q learning algorithm for creating RL agents. Further testing in this approach might yield more conclusive results.

It could be argued that the results of this study could serve as a foundation for future performance evaluation research. This may involve implementing the proposed solution on additional open-source serverless computing platforms and comparing their performance. Furthermore, comparing the obtained outcomes with forthcoming relevant datasets stemming from the application of autoscaling solutions on both open-source and commercial serverless computing platforms is also worth considering.

7 Conclusion and Future Work

Within this manuscript, we have outlined a strategy for addressing the scaling of functions within serverless computing platforms through the decisions of RL agents. This approach

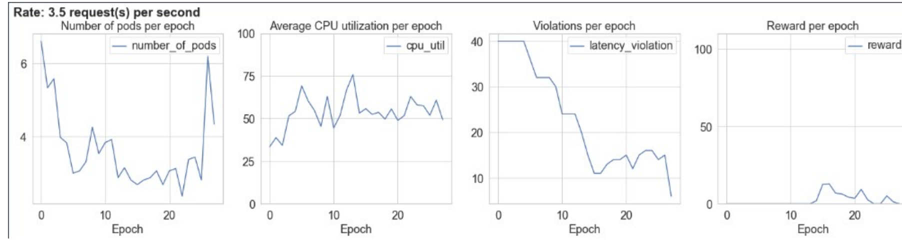

Figure 7: Results for high requests

emerged from recognizing a gap in existing literature concerning RL environments and agents to support real-world tests of serverless computing platform autoscaling. We designed and implemented an RL environment, encompassing both discrete state spaces, and developed an RL agent using a Double Q-learning algorithm. These environments and agents are accessible as open-source code on a Git repository (n.d.), aiming to encourage their adoption and expansion by fellow researchers, fostering further evaluations and comparisons.

Our study demonstrates evaluation outcomes in real environments, confirming the effectiveness of our approach for addressing autoscaling in serverless computing platforms. The benefits of employing environments with discrete state spaces are highlighted, given their quicker training and easier real-world applicability. Simulation environments also play a vital role, enabling the application of trained datasets to predict RL agent behavior and offer insights into their learning patterns. Further testing using Double Q learning algorithm on various difference applications can be helpful in identifying in the application behaviours and help make better decision during runtime.

References

, V. (n.d.). Github - Vikrant2691/research-project, https://github.com/Vikrant2691/research-project.

CNCF (2023). Github - cncf/wg-serverless: Cncf Serverless WG, https://github.com/cncf/wg-serverless.

Eismann, S., Scheuner, J., van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C. L. and Iosup, A. (2021). Serverless applications: Why, when, and how?, *IEEE Software* **38**(1): 32–39.

Garí, Y., Monge, D. A., Pacini, E., Mateos, C. and Garino, C. G. (2020). Reinforcement learning-based application autoscaling in the cloud: A survey.

Hasselt, H. (2010). Double q-learning, *in* J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel and A. Culotta (eds), *Advances in Neural Information Processing Systems*, Vol. 23, Curran Associates, Inc.

Horizontal Pod Autoscaling (n.d.). https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I. and Patterson, D. A. (2019). Cloud programming simplified: A berkeley view on serverless computing.

Kardani-Moghaddam, S., Buyya, R. and Ramamohanarao, K. (2021). Adrl: A Hybrid Anomaly-Aware Deep Reinforcement Learning-Based Resource Scaling in Clouds, *IEEE Transactions on Parallel and Distributed Systems* **32**(3): 514–526.

Kim, Y., Park, J., Yoon, J. and Kim, J. (2022). Improved q network auto-scaling in microservice architecture, *Applied Sciences* **12**(3).
URL: *https://www.mdpi.com/2076-3417/12/3/1206*

Lin, C. and Khazaei, H. (2021a). Modeling and optimization of performance and cost of serverless applications, *IEEE Transactions on Parallel and Distributed Systems* **32**(3): 615–632.

Lin, C. and Khazaei, H. (2021b). Modeling and Optimization of Performance and Cost of Serverless Applications, *IEEE Transactions on Parallel and Distributed Systems* **32**(3): 615–632.

Mahmoudi, N. and Khazaei, H. (2022). Performance Modeling of Serverless Computing Platforms, *IEEE Transactions on Cloud Computing* **10**(4): 2834–2847.

Pimpley, A., Li, S., Srivastava, A., Rohra, V., Zhu, Y., Srinivasan, S., Jindal, A., Patel, H., Qiao, S. and Sen, R. (2021). Optimal resource allocation for serverless queries.

Pujol Roig, J. S., Gutierrez-Estevez, D. M. and Gunduz, D. (2020). Management and Orchestration of Virtual Network Functions via Deep Reinforcement Learning, *IEEE Journal on Selected Areas in Communications* **38**(2): 304–317.

Santos, J., Wauters, T., Volckaert, B. and Turck, F. D. (2023). gym-hpa: Efficient auto-scaling via reinforcement learning for complex microservice-based applications in kubernetes, *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–9.

Schuler, L., Jamil, S. and Kühl, N. (2020). Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments.

van Hasselt, H., Guez, A. and Silver, D. (2015). Deep reinforcement learning with double q-learning.

Xiao, Z. and Hu, S. (2022). Dscaler: A horizontal autoscaler of microservice based on deep reinforcement learning, *2022 23rd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pp. 1–6.

Zafeiropoulos, A., Fotopoulou, E., Filinis, N. and Papavassiliou, S. (2022). Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms, *Simulation Modelling Practice and Theory* .