Simulation Modelling Practice and Theory 116 (2022) 102461

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

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

# Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms

Anastasios Zafeiropoulos [*], Eleni Fotopoulou, Nikos Filinis, Symeon Papavassiliou

*School of Electrical and Computer Engineering, National Technical University of Athens, Iroon Polytechneiou 9, Athens, 15780, Greece*

ARTICLE INFO

*Keywords:*
Serverless application
Serverless computing
Reinforcement learning
Autoscaling
Intelligent orchestration

ABSTRACT

Serverless computing is emerging as a cloud computing paradigm that provisions computing resources on demand, while billing is taking place based on the exact usage of the cloud resources. The responsibility for infrastructure management is undertaken by cloud providers, enabling developers to focus on the development of the business logic of their applications. For managing scalability, various autoscaling mechanisms have been proposed that try to optimize the provisioning of resources based on the posed workload. These mechanisms are configured and managed by the cloud provider, imposing non negligible administration overhead. A set of challenges are identified for introducing automation and optimizing the provisioning of resources, while in parallel respecting the agreed Service Level Agreement between cloud and application providers. To address these challenges, we have developed autoscaling mechanisms for serverless applications that are powered by Reinforcement Learning (RL) techniques. A set of RL environments and agents have been implemented (based on Q-learning, DynaQ+ and Deep Q-learning algorithms) for driving autoscaling mechanisms, able to autonomously manage dynamic workloads with Quality of Service (QoS) guarantees, while opting for efficient usage of resources. The produced environments and agents are evaluated in real and simulated environments, taking advantage of the Kubeless open-source serverless platform. The evaluation results validate the suitability of the proposed mechanisms to efficiently tackle scalability management for serverless applications.

## 1. Introduction

Serverless computing is emerging as a new computing paradigm aimed to make it easier for programmers to use the cloud by supporting management operations, while saving cost through a billing model associated with the exact usage of the cloud resources [1,2]. It is envisaged that serverless computing will become the default computing paradigm, replacing serverful computing and thereby bringing closure to the client–server era [1]. According to the Cloud Native Computing Foundation, “serverless computing refers to a new model of cloud native computing, enabled by architectures that do not require server management to build and run applications" [3]. Developers focus on writing the business logic of their applications, packaging their code in the form of containers and following microservices-based software development approaches. Management operations such as server provisioning and scaling are tackled by a serverless platform under the supervision of a cloud provider.

One of the main characteristics of the serverless computing paradigm is the support of scalability with minimal engineering effort [2]. Decisions regarding horizontal scaling actions are taken by the serverless platform in real-time, usually following the varying workload to be served by each part of the application at each point of time [3]. This approach is mainly targeted

* Corresponding author.
*E-mail address:* tzafeir@cn.ntua.gr (A. Zafeiropoulos).

https://doi.org/10.1016/j.simpat.2021.102461

Available online 4 January 2022

to applications parts that serve bursty workloads, however also applicable to other application types. Numerous autoscaling mechanisms are proposed for managing elasticity actions by serverless platforms. These mechanisms do not necessitate any action on behalf of the developer, however, they pose a management overhead on the cloud provider part, since there is still a need for proper configuration of the serverless platform.

A set of challenges are identified that have to be tackled by cloud providers. Given that manually-driven solutions for scaling management impose administration overhead, lead to waste of resources and to frequent violations to specified resource limitations [4], agility and automation in the management of dynamic and bursty workloads has to be introduced, enabling the on-demand scaling of resources without advanced application-level knowledge. Reaction time in terms of deployment of new instances and decision-making regarding the number of instances to be provisioned or deprovisioned are crucial parameters to be considered. Another challenge regards the identification of the equilibrium conditions that lead to elasticity actions with a two-fold objective; reduction of the violations of the Service Level Agreement (SLA) between the cloud provider and the application provider, and efficient usage of the computing resources to minimize the cost for the deployment and operation of the serverless application. These two objectives can be conflicting and their convergence can boost the benefits of the adoption of the serverless computing paradigm compared to other computing approaches. For instance, in case of serverless query processing platforms, it is shown that 50% of the requests can be served with fewer resources than the reserved ones without any impact on the performance, while in case of accepting a 5% performance loss, the relevant percentage goes up to 92 % [5]. Proper scaling of resources has to take place based on the comprehensive understanding of environmental changes and dynamic factors that can affect the performance of the system, considering the specification of performance and cost models for serverless applications [6,7].

In the current work, we try to address these challenges based on the development of autoscaling mechanisms for serverless applications, powered by Reinforcement Learning (RL) techniques. RL has been evolved as a generic framework for representing and solving control tasks, where decisions must be made or some behaviour must be enacted. In RL, the learning algorithm decides which actions to take for a control task, based on the definition of an ultimate goal to achieve. Upon proper training, RL agents can be applied for driving orchestration actions, injecting intelligence and automation characteristics [8]. It should be noted that the applicability of RL-based technology to optimize auto-scaling capabilities in serverless environments has not been adequately investigated [9]. A few relevant works are available in the literature, focusing mainly on the theoretical formulation of RL problems for tackling elasticity management aspects. Practical considerations and development of solutions integrated into emerging serverless platforms are partially examined.

Under this perspective, the main motivation for our work in this paper is the development of open-source RL-assisted autoscaling mechanisms for serverless applications, including the associated RL environments. These environments are integrated into emerging open-source serverless platforms and namely the Kubeless Kubernetes-native serverless platform. The proposed autoscaling mechanisms can manage serverless workloads with Quality of Service (QoS) guarantees, while opting for efficient resources usage. Scaling decisions are not based purely on the workload characteristics, but considering metrics such as latency, throughput, SLA violations, resources usage and associated costs. Continuous learning of the environment, development and adjustment of scaling policies that maximize the provided reward is taking place, exploiting the knowledge gained through the continuous interaction between the RL agent and the environment. Evaluation results are made available, validating the suitability of the proposed RL-assisted autoscaling mechanisms to efficiently manage scaling actions for serverless applications.

## 2. Background

### 2.1. Serverless computing and applications

As already stated, serverless computing aims to provide ease of use and efficiency for applications deployed over cloud resources [2]. A serverless computing platform may support different deployment models related to provisioning Functions-as-a-Service (FaaS), Backend-as-a-Service (BaaS) or both [10]. In FaaS, developers can deploy small units of code and scaling can take place without any need for management of the underlying infrastructure. In BaaS, a set of backend services (e.g., file storage, messaging, databases, streaming) are made available to developers through Application Programming Interfaces (APIs). These backend services can be used as part of their application and auto-scale based on management rules applied by the cloud provider [10]. A pay-as-you-go model is applied on serverless computing, while, in part of the implementations, no cost appears in case of zero requests for the offered service (a scale to zero option is supported).

A serverless approach is considered a good choice in case of stateless and ephemeral applications. Such applications are distributed in nature and can support workloads that are asynchronous and easy to parallelize into independent tasks [10]. Serverless applications are most commonly used for short-running tasks with low data volume and bursty workloads but are also frequently used for latency-critical, high-volume core functionality. Based on the analysis of 89 serverless applications at [2], it is shown that 84% of the serverless applications serve bursty workloads, 82% consist of five functions or less and 93% consist of 10 functions or fewer [2].

### 2.2. Scaling mechanisms for serverless applications

Automation of elasticity management is one of the main characteristics of the serverless computing paradigm. An autoscaling strategy defines how to adapt the number and type of cloud resources that are reserved to serve the application's demand. Three main types of scaling patterns are applied, namely scale per request, concurrency value scaling and metrics-based scaling [9,11].

Scale per request patterns manage scaling based on the number of incoming serving requests for a function. In this case, the scaling mechanism has to be completely reactive and able to tackle issues related to the cold start of the functions. No queuing is applied for new requests, while an upper limit exists to the latency that each request may face. In concurrency value scaling, each function instance can receive multiple requests concurrently. A maximum number of concurrent requests is defined. When this number is reached, a scaling action is taking place to serve the new requests. Metrics-based scaling tries to keep metrics like CPU usage, memory usage, throughput or latency within a predefined range [11]. This approach is adopted by the majority of the open-source serverless computing platforms, including the Kubeless platform, where the resource-based Kubernetes Horizontal Pod Autoscaler (HPA) is adopted to drive scaling decisions. Metrics-based scaling is not the best choice in terms of performance for reacting to bursty workloads, however it considers performance, resource usage and cost indicators. Thus, it is considered attractive to cloud providers for managing the scaling actions in a way that it serves their clients' needs, while in parallel does not lead to underprovisioning of the cloud resources and to increased cost.

Considering the three different types of scaling patterns, different mechanisms have been proposed to support autoscaling. Such mechanisms include static approaches that support scaling based on the specification of thresholds and usually take advantage of time-series based analysis for decision making [12,13], as well as more dynamic approaches based on machine learning techniques. In the static approaches, the main disadvantage is the need to define scaling rules that can be updated only through the intervention of the system administrator. This imposes a significant overhead, since each set of rules is associated with a specific function. A microservices-based application with numerous functions is going to need the specification of a large set of rules, making hard their definition as well as their update in case of failures. The exploitation of machine learning techniques seems promising to increase automation in this process [13,14].

In our work, to tackle the challenges detailed in Section 1, we follow a metrics-based scaling approach and focus on the support of automation features through the exploitation of RL techniques. The objective is to combine both high performance of the applications with reasonable usage of the available cloud resources, managing the trade-off between limited budget and the desired Quality of Service (QoS) of serverless applications [6].

### 2.3. Reinforcement Learning-based autoscaling

Reinforcement Learning (RL) regards a branch of AI algorithms that is composed of an environment, agents and rewards [15]. The agent is responsible to take actions that change its own state and the state of the environment to achieve a goal. According to the effect of each action towards the achievement of a goal, the agent is rewarded or penalized for this action. By considering the collected feedback over a set of actions, the agent is able to develop policies that maximize the reward.

RL has demonstrated a great potential for automatically solving decision-making problems in complex, uncertain environments. It appears as a promising approach for managing autoscaling in the cloud [13] compared to existing — more static- rule-based approaches combined with time-series data analysis. RL offers the capability to adapt the scaling policies to ensure QoS satisfaction in the presence of various performance-related problems [7]. Since learning of policies is achieved based on the interaction with the environment, no human intervention is required. Dynamicity and adaptivity is supported, since the learning process is continuous and the produced policies may be changed according to changes that occur in the cloud environment [7,13].

Various RL-assisted autoscaling approaches have been proposed in the literature [7,13], targeted to cloud environments. In most of them, model-free approaches are followed, while Q-learning, SARSA and Deep Q-learning are the most widely used RL algorithms [13]. Among the existing approaches, few of them are considering experimentation over real infrastructure, while very few of them refer to serverless environments. Such an approach is investigated at [9], where an RL environment based on Q-learning -for learning effective request-based autoscaling policies — has been designed and evaluated based on the usage of the Knative serverless framework. It is shown that the proposed approach can adapt the concurrency appropriately without prior knowledge within limited time and outperforms the average throughput compared to the default setting of Knative [9]. Autopilot is provided by Google as an autoscaler for cloud computing environments, including support for serverless applications [4]. Autopilot considers both horizontal and vertical scaling of application components, while reinforcement learning techniques are exploited for the vertical scaling part. A deep reinforcement learning resource scaling framework for serverless applications is proposed at [7]. The proposed solution utilizes an anomaly detection module to detect the persistent performance problems in the system as a trigger for the decision-making module of RL to perform a scaling action for correcting the problem.

In the work presented in this paper, we consider the identified gaps in the literature, especially related to the need for the development of open solutions and the existence of few approaches that are targeted to serverless computing platforms. We focus on the specification and development of open and interoperable RL environments for horizontal scaling of serverless applications that are applicable in both simulated and real serverless computing environments.

### 2.4. Reinforcement Learning algorithms

Reinforcement learning (RL) deals with learning through interaction and feedback. As already stated, an agent is developed that can perceive and interpret the environment in which it is placed, take actions and interact with it.

There are different ways to classify RL algorithms [16]. One classification regards model-free and model-based algorithms. Model-free algorithms are easier to implement and use since they do not require knowledge of the environment, while model-based algorithms can select actions by knowing their effect on the applied environment. Another way to classify RL algorithms is based on the means considered to find an optimal policy. In this case, the algorithms are classified as value-based or policy-based. Value-based algorithms aim to learn the value function or the action-value function to generate the optimal policy. The value function is a function that measures how good a state is based on the prediction of future reward. The action-value function measures the same considering both the state and the upcoming action. Policy-based algorithms aim to learn the policy directly using a parameterized function. Value-based algorithms are not suitable in cases that the action space is very high-dimensional or continuous, while policy-based algorithms show better convergence properties but can converge to a local than a global optimum. RL algorithms are also classified as on-policy or off-policy algorithms. On-policy algorithms are learning the policy based on the experience collected through the application of this policy, while off-policy algorithms are learning based on experience collected based on other policies.

In our work in this paper, we focus on model-free RL techniques and the development of RL agents based on the Q-learning and Deep Q-learning (DQL) RL algorithms, while we also consider a model-based RL technique and the development of the relevant agent based on the DynaQ+ RL algorithm.

Q-learning is a model-free, value-based, off policy RL algorithm that seeks to find the best action to take given the current state, considering the calculation of action-value functions. It seeks to learn a policy that maximizes the total reward. Q-learning uses a table – called as Q-table – to store all state–action pairs, thus it can be used for small and discrete environments (discrete state space, discrete action space). In a continuous environment, Q-learning can still be applied by discretizing the states.

DynaQ is an RL algorithm that illustrates how real and simulated experience can be combined in building a policy. Simulation experience is generated based on a model of the environment, while learning of a policy is taking place based on both the real interaction with the environment and the simulation experience. DynaQ is considered as a model-based, value-based, off-policy RL algorithm. DynaQ+ regards an extension of DynaQ, where a bonus reward is provided for actions that have not been tried for a long time, since there is a greater chance that the dynamics for that actions might have changed.

DQL has been developed by enhancing Q-Learning with deep neural networks (DNN). DQL is a model-free, value-based, off policy RL algorithm. Since, in most cases, it is not practical to maintain a table containing values for each combination of state and action, a deep neural network – known as Deep Q-network (DQN) – is trained over a continuous state space to approximate the Q-table, making DQL applicable in environments with large state space. DQL is applied over a continuous state space and discrete action space.

### 2.5. Main contribution

The main contribution of this paper with regards to the state of the art is twofold. On one hand, we provide the modelling of a set of RL environments that can be applied for managing autoscaling mechanisms in serverless computing platforms, considering the existence of few works targeted to such platforms. On the other hand, we provide an implementation of these environments based on the adoption of existing RL algorithms and the development of RL agents that are interoperable with monitoring and orchestration mechanisms of the Kubeless Kubernetes-native serverless computing platform, as well as a simulation environment. The developed RL environments, including the RL agents and their porting into Kubeless, are made available as open-source, aiming at their adoption and extension by any interested party and their potential integration in further serverless computing platforms.

## 3. Reinforcement Learning assisted autoscaling environments

To conceptualize and evaluate the performance of the RL-assisted autoscaling mechanisms, we have developed a series of RL environments and the corresponding agents. Each RL environment encompasses the orchestration ecosystem, including the mechanisms for applying scaling actions and the monitoring infrastructure for collecting data related with resources usage, Quality of Service (QoS) indicators and information regarding the deployment status of the serverless application. In each setting, the RL agent takes actions that are applied in the RL environment. Each action is associated with the specification of resources-usage thresholds that trigger scaling events. Actually, when the specified threshold is surpassed, the orchestrator proceeds to the creation of new instances of the considered serverless function. The objective is to serve the dynamic workload, while guaranteeing the provision of the required QoS level in accordance with the Service Level Agreement (SLA) that may be established between the application provider and the cloud provider. Similarly, deprovisioning of the deployed instances take place in case of low resources usage level – compared to the configured scaling thresholds – to avoid unnecessary consumption of resources. Each RL action is interpreted into a reward and a representation of the state, which are fed back into the agent. The objective is to provide autonomous adjustment of the scaling thresholds in the orchestration ecosystem, aiming to achieve high QoS in the provision of the serverless application, while in parallel keeping the resources' usage level (e.g., the number of active instances of the application in terms of containers) as small as possible. To achieve so, in the reward function we penalize the SLA violations, as well as the increased usage of compute resources in terms of the deployed instances. The overall RL process for supporting autoscaling mechanisms is depicted in Fig. 1.

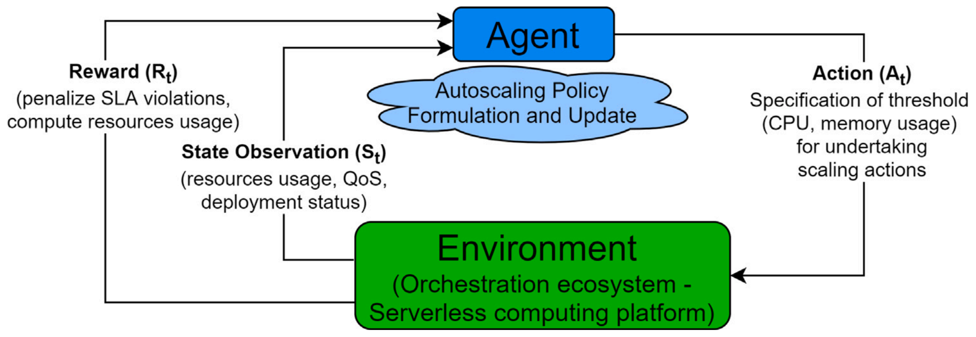
Agent
Autoscaling Policy Formulation and Update
Reward ($R_t$) (penalize SLA violations, compute resources usage)
State Observation ($S_t$) (resources usage, QoS, deployment status)
Action ($A_t$) Specification of threshold (CPU, memory usage) for undertaking scaling actions
Environment (Orchestration ecosystem - Serverless computing platform)

**Fig. 1.** Reinforcement Learning Mechanism for Autoscaling.

The developed RL environments can be classified in environments with discrete state space and environments with continuous state space. The objective for this classification is to examine the autoscaling performance achieved in each environment, based on the application of the selected RL algorithm. Each environment is targeted to a different set of algorithms. For instance, in the case of environments with discrete state space and discrete action space, the Q-learning and DynaQ+ algorithms are applied, while in the case of environments with continuous state space and discrete action space, the DQL algorithm is applied.

To define the state, we have considered a set of metrics that can affect the decision making in the autoscaling policy. As already mentioned, such metrics include resource usage metrics, QoS metrics and metrics related to configuration aspects or the status of the deployed containers. The resource-usage metrics refer to CPU and memory usage of the deployed containers, while the QoS metrics refer to the latency (response time) and the throughput (requests served per second) of the offered service. Furthermore, information regarding the active number of the deployed containers is always taken under consideration while, in some cases, the number of containers that have been terminated (e.g., due to faults during their instantiation or operation) is included in the state space. A complete list of the metrics that are used for the composition of the state in the defined RL environments is listed in Table 1.

A list of the developed environments is provided in Table 2. For each environment, we provide details for the type of the state and the actions, the set of metrics that compose the state, the set of metrics used for the definition of the reward function, the total number of states and actions that may appear, the set of applied RL agents and the deployment testbeds that the RL agents interact with.

## *3.1. RL environment with discrete state and action space*

The first RL environment has been developed based on the specification of a discrete state and action space. Two agents are applied to the environment k8s-env-discrete for evaluating the achieved performance based on the Q-learning and DynaQ+ RL algorithms.

In the k8s-env-discrete environment, the state consists of four metrics, namely CPU usage, the applied CPU threshold for autoscaling, the ratio of the number of active instances of the serverless function with the maximum allowed instances, and the ratio of the measured latency with the latency defined in the SLA. To manage to keep the overall state space small, we considered a set of discretized values per metric. For the ratio of the CPU usage, seven states are defined. These discrete states correspond to values that range 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%. Similar classification applies (seven states) in the case of the ratio with the latency metric. Regarding the ratio with the active instances and the CPU threshold value, since the real number cannot overcome the maximum number (100%), five states are defined that correspond to values that range from 0%–20%, 20%–40%, 40%–60%, 60%–80% and 80%–100%. The total number of states for this environment is 1225 (7x7x5x5). The actions space consists of three potential actions. Based on these actions, the applied threshold in the CPU usage for triggering a scaling event may be reduced by 20%, remain at the same value, or be increased by 20%.

Upon each action, a reward is provided, as detailed in Listing 1. The main parameters considered in the definition of the reward function are the number of active instances of the serverless function and the average monitored latency. These parameters are considered based on different weights, where a larger weight (0.7) is given to the latency part and a smaller weight (0.3) to the active instances part. With regards to the number of active instances, a linear function is applied with maximum value (equal to 100) in case of one active instance and minimum value (equal to 0) in case of the maximum number of active instances. With regards to the latency, the maximum reward is given in cases where the latency is as close as possible to 0.8*SLA_latency. This value is selected having in mind that in bursty workloads, a scaling action has to be applied in a short reaction time upon the identification of high latency to avoid SLA violations. The agent can be trained accordingly to tackle such situations. On the other hand, in case of small latency, the desired QoS level for the application is guaranteed, providing some flexibility to the agent to reduce the number of allocated resources to avoid high costs. In values that are smaller or larger than 0.8*SLA_latency, the provided reward is reduced in an exponential way. The reduction is more steep in case of values that are larger than 0.8*SLA_latency. In case that the current latency is greater that the SLA_latency, this is reported as an SLA violation and the provided reward is penalized to zero.

**Table 1**
Metrics composing the RL Environment State Space.

| Metric name | Metric description | Measurement unit | Metric type |
|---|---|---|---|
| CPU Usage | Average percentage of the CPU usage of the active instances (containers/pods) | Percentage (%) | Resource usage |
| Memory Usage | Average percentage of the memory usage of the active instances (containers/pods) | Percentage (%) | Resource usage |
| CPU Usage Threshold | Threshold of the average percentage of the CPU usage of the active instances (containers/pods) for triggering a scaling action (as defined in Kubernetes Horizontal Pod Autoscaler) | Percentage (%) | Resource usage |
| Memory Usage Threshold | Threshold of the average percentage of the memory usage of the active instances (containers/pods) for triggering a scaling action (as defined in Kubernetes Horizontal Pod Autoscaler) | Percentage (%) | Resource usage |
| Active Serverless Function Instances | Percentage of the active instances (containers/pods) of the serverless function with regards to the maximum allowed instances per function | Percentage (%) | Configuration - Status |
| Terminated Serverless Function Instances | Percentage of the terminated instances (containers/pods) of the serverless function with regards to the maximum allowed instances per function | Percentage (%) | Configuration - Status |
| Latency | Average time for serving a request by the deployed instances (containers/pods) | Time (ms) | Quality of Service |
| SLA Latency | Average time for serving a request by the deployed instances (containers/pods) based on the applied SLA | Time (ms) | Quality of Service |
| Throughput Rate | The number of successfully served requests per second by all the deployed instances (container/pods) | Number/Time | Quality of Service |
| Success Ratio | The percentage of successfully served requests compared to the total submitted requests by all the deployed instances (container/pods) | Percentage (%) | Quality of Service |

```
reward = 0
Max Reward = 100
Min Reward = 0
pod weight = 0.3
latency weight = 0.7

#smooth operation with low throughput level and successful throughput rate
if num of pod = 1 and current_latency <= sla_latency:
   Reward = max
   return Reward
else if current_latency > sla_latency:
  Reward = 0
  return Reward

pod reward = -100 / (max_pod - 1) * num_pods + 100 * max_pod / (max_pod - 1)
reward += pod weight * pod reward

# if current_latency < sla_latency
if current_latecy / sla_latency < 0.8:
  latency reward = 100 * e^(-0.3 * d * (0.8 - current_latency / sla_latency)^2)
# if current_latency > sla
else if current_latecy / sla_latency > 0.8:
   latency reward = 100 * e^(-10 * d * (0.8 - current_latency / sla_latency)^2)
reward += latency weight * latency reward

# where: d defines the sharpness of the reward function
```

Listing 1: Discretised Environment Reward Definition.

**Table 2**
Autoscaling RL Environments for Serverless Computing Platforms.

| Env. name | Env. description | State definition | Reward metrics | States - actions | RL agent | RL agent deployment |
|---|---|---|---|---|---|---|
| k8s-env-discrete | Discrete State, Discrete Action | CPU, CPU Threshold, Active Instances/Maximum Instances, Latency/SLA Latency | Active Instances, Latency | 1225 - 3 | Q-learning, DynaQ+ | Kubeless, Simulation |
| k8s-env-cont-v0 | Continuous State, Discrete Action | CPU, CPU Threshold, Memory, Memory Threshold, Active Instances, Terminated Instances, Throughput Rate, Success Ratio, Latency | Active Instances, Latency, Throughput | N/A - 9 | Deep Q-learning (DQL) | Kubeless |
| k8s-env-cont-v1 | Continuous State, Discrete Action | CPU, CPU Threshold, Active Instances, Terminated Instances, Latency | Active Instances, Latency | N/A - 3 | Deep Q-learning (DQL) | Simulation |

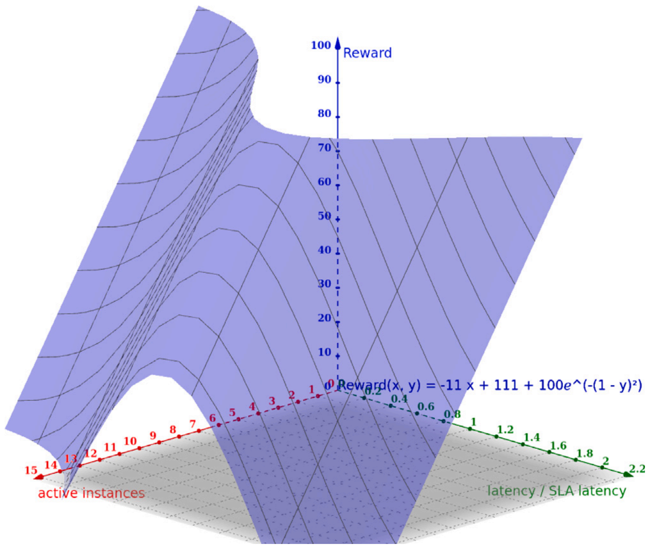

**Fig. 2.** Reward function visualization.

Fig. 2 provides a graphical representation of the reward function. As depicted, an increase in the number of the active instances of the serverless function leads to smaller reward. Regarding the latency, the reward is maximized in case that the monitored latency is close enough (somehow smaller) to the agreed latency in the SLA, as already explained.

### 3.2. RL environments with continuous state space and discrete action space

Two more RL environments have been developed according to Table 2 based on the specification of a continuous state and discrete action space. In both cases, an agent based on the Deep Q-learning (DQL) algorithm has been applied.

In the k8s-env-cont-v0 environment, the continuous state space consists of nine metrics in a percentage format. These metrics are the CPU usage, the applied CPU threshold, memory usage, the applied memory threshold, the ratio of the number of active instances of the serverless function with the maximum allowed instances, the ratio of the number of terminated instances of the serverless function with the maximum allowed instances, the successfully served requests, the ratio of the measured throughput with the throughput defined in the SLA and the ratio of the measured latency with the latency defined in the SLA.

The actions space consists of nine potential actions that correspond to changes in the scaling thresholds based on the CPU and memory usage values. Specifically, three actions are defined, where action 0 dictates the decrease of the threshold by 20%, at action 1 no change is applied, while action 2 dictates the increase of the threshold by 20%. Therefore, to manage both CPU and memory usage thresholds, the action space is composed by two numbers $[\alpha, \beta]$ where $\alpha$ and $\beta$ refer to numbers in the set $\{0, 1, 2\}$, $\alpha$ represents the action for the CPU threshold and $\beta$ the action for the memory threshold.

Upon each action, a reward is provided. The reward function is similar to the function defined in Section 3.1. A minor difference regards a slight modification in the weights, where a larger weight is given to the latency (0.4) and throughput (0.4) part and a smaller weight (0.2) to the active instances part. Furthermore, in addition to the latency, the throughput is also considered in the reward function, in a way similar to the latency. In this case, the maximum reward is given in cases where more than 95% or the requests are served successfully or the achieved throughput does not overpass the value 2*SLA_throughput.

In the k8s-env-cont-v1 environment, the main difference is the selection of a smaller state space. Compared to the k8s-env-cont-v0, the memory and throughput-related metrics are not included in the state space, while the action space consists of three actions targeted to the thresholds for the CPU usage, similar to the k8s-env-discrete environment.

It should be noted that the main advantage of the continuous state environments compared to discrete state environments is the absence of information loss due to discretization, since the state is kept intact. This can potentially lead to better results in terms of efficiency of the RL agent to get greater reward, however with a cost in terms of the time required for the training phase.

## 4. Testbed setup and RL agents implementation

### 4.1. Testbed setup

#### 4.1.1. Cloud computing infrastructure

To realize experiments for evaluating the performance of the developed RL agents over the specified environments, a cloud computing testbed has been deployed, as depicted in Fig. 3. The testbed combines a real and a simulated environment. For the real environment, a Kubernetes cluster is created over three nodes, namely three Intel NUC PCs (two of them with Intel Core i5 and 16 GB of memory and one of them with Intel Core i7 and 32 GB of memory). The Kubeless serveless computing platform has been installed for managing function deployments over the cluster. Scaling mechanisms are applied based on the Kubernetes Horizontal Pod Autoscaler (HPA) [17] (short description of the HPA is provided in Section 4.2). The Kubernetes Metrics Server is used for the collection of resources usage and QoS metrics. These metrics are provided to the Prometheus Monitoring Engine that acts as a time series database. Regarding the serverless functions that are used for the experimentation part, a microservice that calculates a Fibonacci number has been developed. NGINX Ingress Controller is being used as the load balancer for this microservice. Stress testing of the serverless functions takes place through the specification of HTTP traffic profiles in the Vegeta HTTP load testing tool. Each instance of the serverless function is deployed within a Pod.

Modelling of the RL environments (Table 2) has been realized with the OpenAI Gym [18] toolkit. OpenAI Gym facilitates the definition of test problems called environments that can be used to train an agent using reinforcement learning. These environments are considered as the "glue" between the RL agents and the testbed itself. As depicted in Fig. 3, the environment collects the resource and SLA metrics from the HPA, the Metrics server and the Prometheus monitoring engine and offers them to the agent in the form of an observed state. Similarly, it gets the selected action by the RL agent and applies the new thresholds in the Kubernetes cluster HPA mechanism. Furthermore, depending on the observed state of the environment and the provided action, the environment calculates the reward and returns it as a feedback to the agent. To reduce the overall training time for the RL agents, distributed training techniques are applied. Parallel deployment of the developed environments and agents is taking place, enabling the collection of a wider amount of experience that is stored at the same Q function (Q-table or deep neural network).

The set of developed RL environments and agents are made available as open-source in a Gitlab repository [19]. The objective is to promote their adoption and extension by the scientific community, as well as their integration in open-source serverless computing platforms (e.g., in the current work, integration is supported with the Kubeless platform).

#### 4.1.2. Simulation environment

A simulation environment has been developed to support extensive training of RL agents to overcome time-related limitations of real environments. Each action applied to the real environment needs 6 min to get enforced and affect the function metrics in a stable way, while at the simulation environment this happens in some nanoseconds. In the simulation environment, the simulated state, reward and actions have been produced based on probability distributions constructed by the collected data (historical traces) in the real environment. In this way, evaluation results over a large period can be produced, taking advantage of the produced knowledge in the real environment.

Specifically, a sampling distribution technique is applied over a set of metrics to calculate their probability distributions based on the collected sample data. For each metric, the associated sampling distribution is produced through repeated sampling from the available data. The mean values are calculated and made available in an array, while the probability distribution of the mean is considered to closely approximate a normal distribution in accordance with the central limit theorem. The calculated mean values and the relevant standard deviations are stored in a dictionary that is offered to the "Metrics Generation" component of the simulation environment.

Based on the provided distributions, the "Metrics Generation" component is able to predict the values of specific metrics based on their relevance with the rest of metrics in the dataset. We actually take advantage of this component to predict the number of

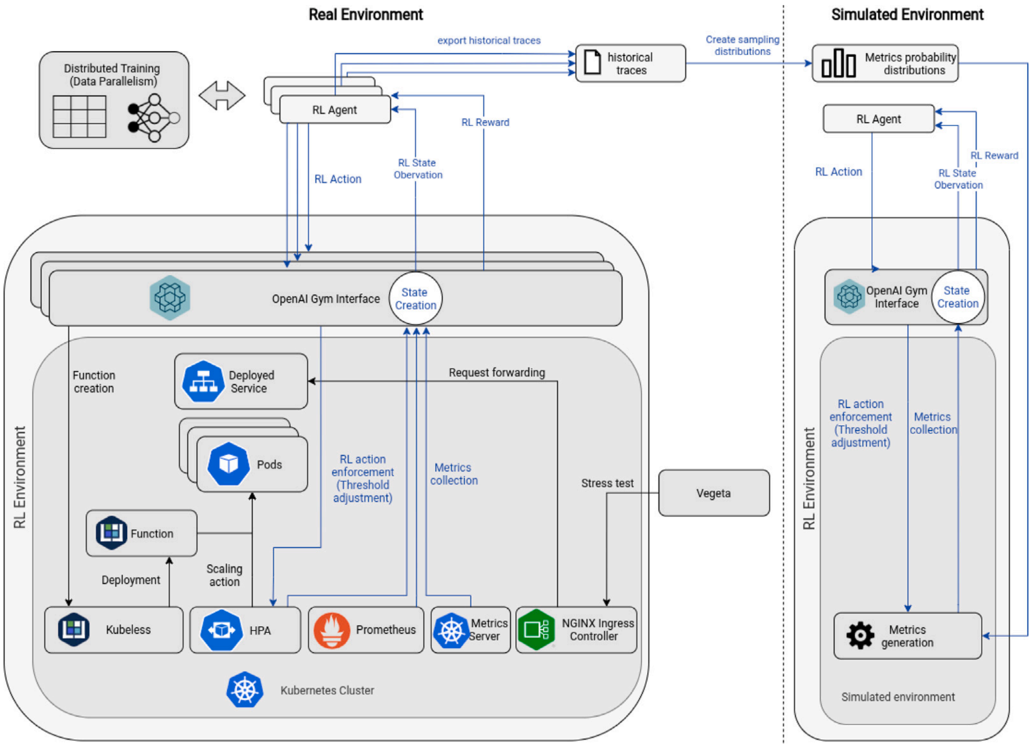

**Fig. 3.** Testbed setup for real and simulation environments.

pods (replicas or instances) that are deployed, the current CPU utilization and the observed latency. To calculate the number of pods, we consider the distributions of the HPA threshold and the current stress load on the service. The current CPU utilization depends on the current stress load and the number of deployed pods, while the latency of the deployed function depends on the current stress load and the number of deployed pods.

## 4.2. Horizontal Pod Autoscaling Mechanism

The Horizontal Pod Autoscaler (HPA) is a module of the Kubernetes orchestrator that periodically adjusts the number of replicas (instances) to match the observed metrics, such as average CPU utilization, average memory utilization or any other custom metric, to the target specified by the system administrator. This target is considered as the scaling threshold and is statically defined by the system administrator. The Kubeless serverless platform leverages Kubernets HPA to automatically scale functions based on the defined workload metrics. In the work presented in this paper, the definition of the targets is made by the RL agents, automating the overall scaling process.

Following, we provide short description on the way that the HPA works. A control loop is implemented where, at each time period, the HPA gets information regarding the metrics that are considered in the scaling decision. The HPA operates on the ratio between the desired metric value and the current metric value based on Eq. (1):

$$desiredReplicas = ceil[currentReplicas * (currentMetricValue/desiredMetricValue)] \tag{1}$$

In case of multiple instances of a function, the currentMetricValue is computed as the average of the given metric across all the active instances. In case that an instance fails to provide a value for the considered metric, the calculation takes place based on the rest instances. To avoid fluctuations in case of continuous scaling in and out of the instances, a cooldown period is introduced upon a downscale action. During the cooldown period, no further downscale action can be applied. This value is set to 5 min. In case of consideration of multiple metrics for scaling (e.g., CPU and memory usage), the HPA evaluates each metric and considers the largest measured scale as the proposed one.

## 4.3. RL agents implementation

### 4.3.1. Q-learning agent

A Q-learning agent is developed for the k8s-env-discrete environment. The Q-learning agent observes a set of 1225 states that are composed of four metrics and performs three types of actions. The values in the Q-table are initialized to the value 50, while the

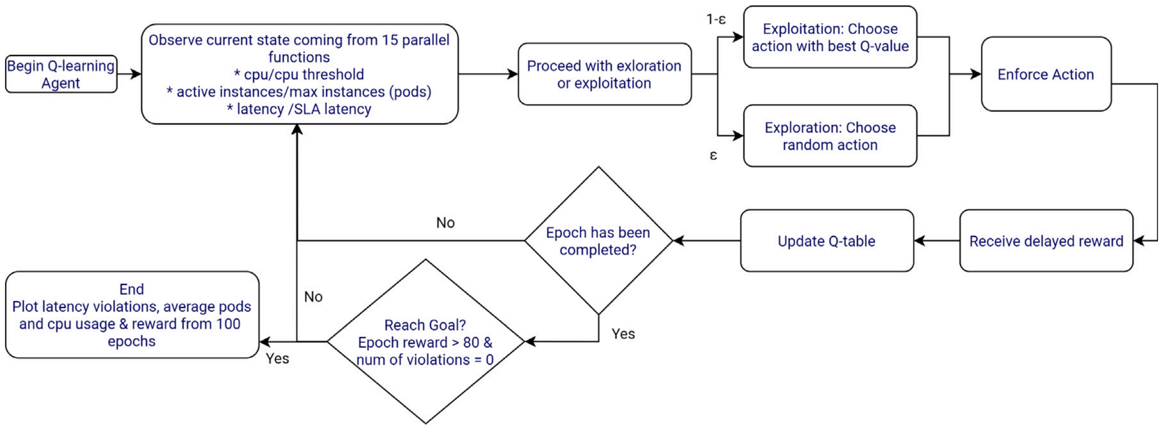
Begin Q-learning Agent
Observe current state coming from 15 parallel functions
* cpu/cpu threshold
* active instances/max instances (pods)
* latency /SLA latency
Proceed with exloration or exploitation
1-ε
ε
Exploitation: Choose action with best Q-value
Exploration: Choose random action
Enforce Action
Receive delayed reward
Update Q-table
Epoch has been completed?
No
Yes
Reach Goal? Epoch reward > 80 & num of violations = 0
No
Yes
End
Plot latency violations, average pods and cpu usage & reward from 100 epochs

**Fig. 4.** Q-learning Agent flow diagram.

Q-table is formulated gradually during the training phase. Details for the state and the supported actions are provided in Table 2. A flow diagram with the business logic supported by the agent is depicted in Fig. 4.

Initially, the Q-learning agent is trained by interacting directly with the Kubeless serverless platform. To examine the behaviour of the agent in long-term, it is also applied over a simulation environment, as detailed in Section 4.1. The first step of the overall flow regards the observation of the current state of the environment. To reduce the required time for the training process, we support the collection of observations in a distributed way, based on the deployment of parallel instances of the serverless function over the available infrastructure. In our case, the number of the parallel instances is 15. Each function operates at its own environment and a dedicated agent, however, all of them provide updates to the same Q-table. In this way, an important amount of data can be fed to the Q-table within a limited number of epochs.

Upon the state observation, an exploration or exploitation phase follows based on the application of a decayed epsilon value to combine both cases. In the exploration phase, the agent chooses a random action to manage to visit most of the candidate states and detect the most effective state–action combinations. In the exploitation phase, the agent chooses the action with the best Q value. Following, the action is enforced leading to the receipt of a reward based on the defined reward function. The Q-table is updated accordingly.

The agent is executed during a set of epochs (up to 100 epochs). The execution goes on until the agent achieves the goal of getting reward values greater than 80, while no SLA violations are present. For the first ten epochs, the agent is executed with an epsilon = 1 that favours the full exploration of the environment. Epsilon is gradually reduced, while at the last ten epochs (90th–100nd epochs), it is set to zero to depict the agent performance upon choosing the best actions (exploitation) based on the values in the Q-table. Each epoch execution is taking place over sixteen timesteps, where each timestep has a duration of 6 min. At each timestep, a specific workload is posed to the serverless function. The workload follows a distribution where there is a gradual increase and decrease in the load each eight timesteps. The load in the time steps follows the pattern $\lambda \rightarrow \lambda \rightarrow 3\lambda \rightarrow 3\lambda \rightarrow 5\lambda \rightarrow 5\lambda \rightarrow 7\lambda \rightarrow 7\lambda \rightarrow 9\lambda \rightarrow 9\lambda \rightarrow 7\lambda \rightarrow 7\lambda \rightarrow 5\lambda \rightarrow 5\lambda \rightarrow 3\lambda \rightarrow 3\lambda$ where $\lambda$ is the basic request rate per second.

### 4.3.2. *DynaQ+ agent*

A DynaQ+ agent is developed for the k8s-env-discrete environment. A flow diagram with the business logic supported by the agent is depicted in Fig. 5. Similarly to the Q-learning agent, the DynaQ+ agent observes a set of 1225 states, performs three types of actions and its Q-table is initialized to the value 50 (Table 2). The overall configuration for the execution of the DynaQ+ agent is the same with the configuration already described for the Q-learning agent.

DynaQ+ involves three basic phases, namely the direct RL, the model learning and the planning phases. At the direct RL phase, training is taking place in the real environment in a way similar to Q-learning. Based on the state observation, an action is selected using the $\epsilon$-greedy method. The action is enforced, leading to the next observed state and reward, while the Q-table action values are updated accordingly.

The model learning phase is used to improve the specified model. Using the observed next state and reward, the agent updates both the model and the tau table. The motivation behind the use of the tau table is to give a bonus reward for actions that have not been tried for a long time, since there is a greater chance that the dynamics for that actions might have changed. In particular, if the modelled reward for a transition is $r$, and the transition has not been tried in $\tau(s,a)$ time steps (where $s$ represents a specific state and $a$ a specific action), then planning updates are done as if that transition produced a reward of $r+\kappa\sqrt{\tau(s,a)}$, for some small $\kappa$.

In the planning phase (indirect RL), the agent keeps updating the Q-table based on the model experience and not by direct interaction with the k8s-env-discrete environment. At the planning phase, the action values are calculated by $n$ (150 in our case) simulated experiences using random starting states and actions from the model table. The process of choosing the state and action to simulate an experience is known as 'search control'.

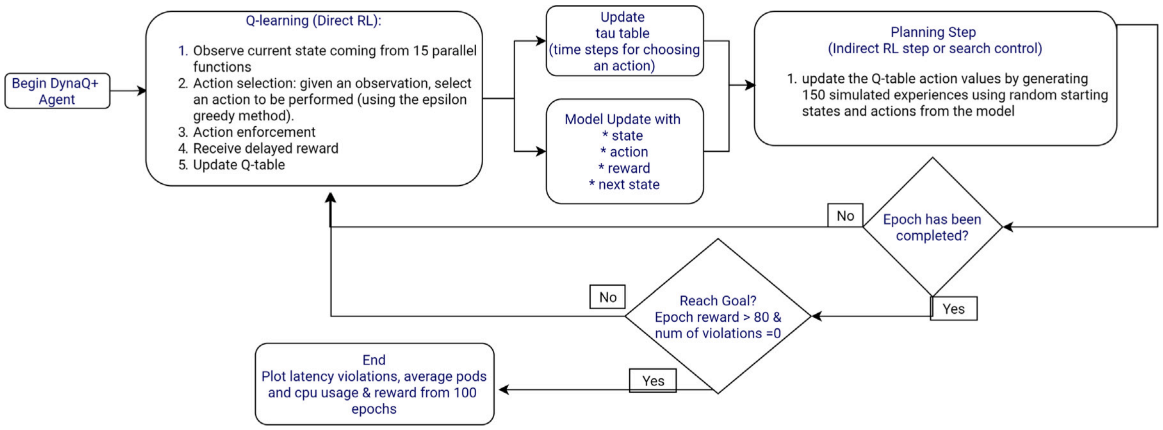
Begin DynaQ+ Agent
Q-learning (Direct RL):
1. Observe current state coming from 15 parallel functions
2. Action selection: given an observation, select an action to be performed (using the epsilon greedy method).
3. Action enforcement
4. Receive delayed reward
5. Update Q-table
Update tau table (time steps for choosing an action)
Model Update with * state * action * reward * next state
Planning Step (Indirect RL step or search control)
1. update the Q-table action values by generating 150 simulated experiences using random starting states and actions from the model
Epoch has been completed?
No
Yes
Reach Goal? Epoch reward > 80 & num of violations =0
No
Yes
End Plot latency violations, average pods and cpu usage & reward from 100 epochs

**Fig. 5.** DynaQ+ Agent flow diagram.

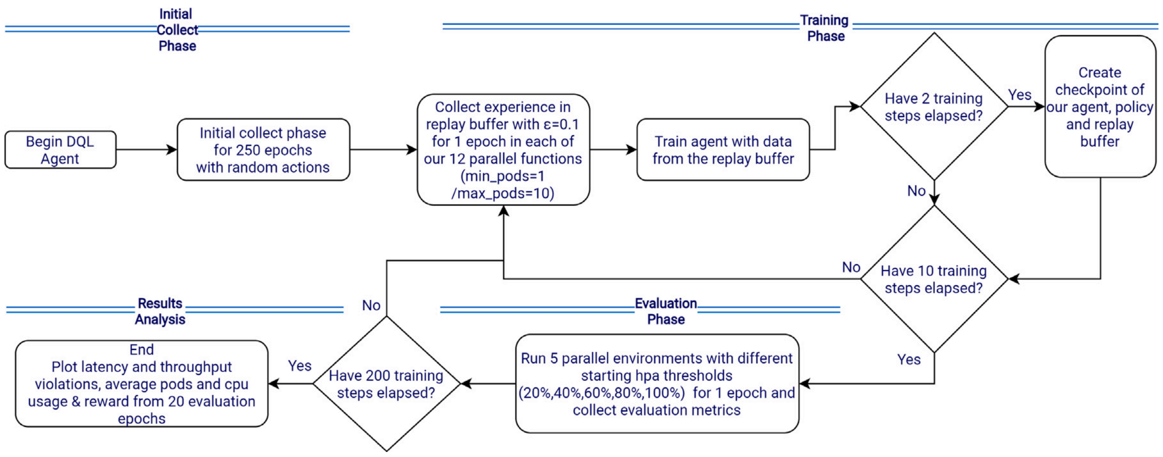
Initial Collect Phase
Training Phase
Begin DQL Agent
Initial collect phase for 250 epochs with random actions
Collect experience in replay buffer with ε=0.1 for 1 epoch in each of our 12 parallel functions (min_pods=1 /max_pods=10)
Train agent with data from the replay buffer
Have 2 training steps elapsed?
Yes
Create checkpoint of our agent, policy and replay buffer
No
Have 10 training steps elapsed?
No
Yes
Evaluation Phase
Run 5 parallel environments with different starting hpa thresholds (20%,40%,60%,80%,100%) for 1 epoch and collect evaluation metrics
Have 200 training steps elapsed?
No
Yes
Results Analysis
End Plot latency and throughput violations, average pods and cpu usage & reward from 20 evaluation epochs

**Fig. 6.** Deep Q-learning (DQL) Agent flow diagram.

The learning and planning phases are accomplished by exactly the same algorithm that is applied over real experience for learning and over simulated experience for planning.

### 4.3.3. Deep Q-learning agent

A Deep Q-learning (DQL) agent is developed for the k8s-env-cont-v0 and k8s-env-cont-v1 environments. A flow diagram with the business logic supported by the agent is depicted in Fig. 6. The DQL agent is trained both in the Kubeless serverless platform (via the k8s-env-cont-v0) and in the simulated environment (k8s-env-cont-v1), as detailed in Section 4.1.

As stated before, DQL is an extension of the Q-learning algorithm where the table with Q values is replaced by a deep neural network. RL is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function [20]. This instability is due to the existence of high correlation in the sequence of observations, the fact that small updates to Q values may significantly change the policy and therefore change the data distribution, and the existence of high correlations between the action-values (Q values) and the target values $r+\gamma \max_{a'} \hat{Q}(s', a')$ [20]. The DQL agent addresses these instabilities with the use of a replay buffer that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution. Furthermore, an update that adjusts the action-values (Q values) towards target values is done periodically to reduce correlations with the target [20].

The process of loading the replay buffer with historical traces is the initial collect phase of the agent execution. The agent interacts in an random way with the environment k8s-env-cont-v0 during 250 epochs. Each epoch execution is taking place over four timesteps, where each timestep has a duration of 6 min. Thus, the time required for the completion of the agent's initial collect phase is 4 days. At each timestep, a specific workload is posed to the serverless function. The workload follows a distribution where there is a gradual increase and decrease in the load per 2 timesteps. The workload follows the pattern $\lambda \rightarrow 7\lambda \rightarrow 7\lambda \rightarrow \lambda$, where $\lambda$ is the basic request rate per second. The workload has been simplified in comparison with the discretized environment to boost the convergence of the agent into an optimal policy formulation sooner.

Similarly with the previous agents, the collection of observations is realized in a distributed way, based on the parallel deployment of 12 serverless function instances. During the replay buffer preparation, a set of 12.000 historical traces are added to it before the start of the DQN training phase (250 epochs * 4 historical traces/epoch * 12 functions). At the training phase, the agent collects data with an $\epsilon$-greedy policy and $\epsilon$ equal to 0.1. After each epoch, the new data enter to the buffer and the agent's DQN is trained with data from the replay buffer. Gradually, the random historical traces of the replay buffer are replaced by the historical traces coming from the tracing of the optimal actions, making the agent to progressively pass from the exploration to the exploitation phase. Every 2 training steps, checkpoints of the agent, policy and replay buffer are saved and used as a basis for the ongoing training. Each training phase is completed every 10 epochs where the agent enters at the evaluation phase. Each training phase has a duration of 4 h (10 epochs * 4 timesteps/epoch * 6 minutes/timestep) and enriches the replay buffer with 480 historical states (10 epochs * 4 historical traces/epoch * 12 parallel functions). During the evaluation phase, 5 different serverless functions are initialized. For each function, a different CPU HPA threshold (values from the set: 20, 40, 60, 80, 100) is applied and the agent checks its performance during an epoch by enforcing a greedy policy over the full set of the weights the DQN has learned so far. Each evaluation epoch has a duration of 24 min (4 timesteps/epoch * 6minute/timestep). The switch between the training and evaluation phases is repeated 20 times, leading to a total duration of 7,5 days (4 days initial collection + 20 times * 4 h + 20 times * 24 min) and the generation of 21600 historical states (12000 historical states from initial collection + 480 historical states * 20 times). Finally, the agent produces a plot with the latency violations, the average pods and CPU usage and the reward from 20 evaluation epochs. The process is repeated on k8s-env-cont-v1 taking advantage of the big amount of historical traces that have been collected and their underlying distributions. Since the interaction with the simulated environment is faster, the process is repeated for 700 evaluation epochs instead of 20 with a total duration of 2 min revealing how the agent will reach a good convergence level at the future.

## 5. Evaluation results and discussion

### 5.1. Evaluation results

A set of performance evaluation results have been produced based on the deployment of the developed RL agents over the created RL environments. In all the cases, we have aggregated the collected data based on the posed workload and examined the performance of the various agents.

In the case of the environment k8s-env-discrete and the Q-learning agent, in Fig. 7 we depict the produced results from the stress test in the real environment at a low and a high request rate. The reason for making this classification is to show the different learning rates that are achieved. In case of high request rates, in the initial set of epochs, more SLA violations are noticed, leading to very small reward levels. As the number of epochs increases, the agent learns to better adjust the threshold to avoid violations while in parallel slightly reducing the number of pods and increasing their average CPU usage, leading to a relevant increase in the reward (e.g., higher reward in the range of epochs from 55 to 80). In case of low request rates, the learning curve in the agent follows a more upward trend, as depicted in the increase in the received reward from levels close to 0 to levels close to 80. The SLA violations are significantly reduced, the number of the used pods remain in low levels and their CPU usage is slightly increased.

Following, in Fig. 8, the Q-learning agent is applied over the k8s-env-discrete simulated environment. To manage to get simulation results, the distributions of the metrics (number of pods that are deployed, the current stress workload, the respective HPA thresholds, the current CPU utilization and the observed latency) that are used for the observation of the state and the calculation of the reward are produced. The Q-learning agent interacts with the simulated environment during 1000 training epochs. Two types of workload are applied to examine the performance of the Q-learning agent in dynamic scenarios. The first workload is similar to the workload used for training purposes, as detailed in Section 4.3.1. The second workload follows a bursty profile that corresponds to the pattern $\lambda \rightarrow 9\lambda \rightarrow \lambda \rightarrow 9\lambda$. In both cases, a 10-point simple moving average (SMA) of the performance evaluation results is produced.

Based on the simulation results, the agent converges to the collection of a high reward (values close to 90) while reducing significantly the number of SLA violations. The number of used pods seem to be slightly increased with a parallel slight decrease in the average CPU usage. Such conditions are considered by the Q-agent optimal, since it is able to serve the posed workload with the desired QoS and without over-consuming the available resources. Similar behaviour is noticed in both types of workload, validating the ability of the agent to efficiently manage dynamic scenarios, where the workload characteristics are changing during the evaluation period.

In the case of the environment k8s-env-discrete and the DynaQ+ learning agent, in Fig. 9 we depict the produced results from the stress test in the real environment at a low and a high request rate. In case of low request rates, DynaQ+ manages to coverage faster to an optimal policy with high reward and low SLA violations, while having small fluctuations in the number of the reserved pods and their average CPU usage. This stabilization potential is not present in case of the high request rates, however, also in this case, the reward is following an increasing trend upon the twelfth epoch, the SLA violations are following a declining trend and the average CPU utilization seems to slightly increase with a parallel decrease in the number of pods. As expected from the theory, the indirect RL phase of the DynaQ+ agent leads to an early convergence (around the 25th epoch), while the same behaviour was observed much later (after the 80th epoch) at the Q-learning agent.

By applying the same agent over the simulation environment, in Fig. 10 we can observe the overall convergence trends of DynaQ+ in 1000 epochs. Similarly to the case of Q-learning agent, two types of workload are applied, while a 10-point simple moving average (SMA) of the performance evaluation results is depicted. The DynaQ+ agent presents earlier convergence but more oscillation in

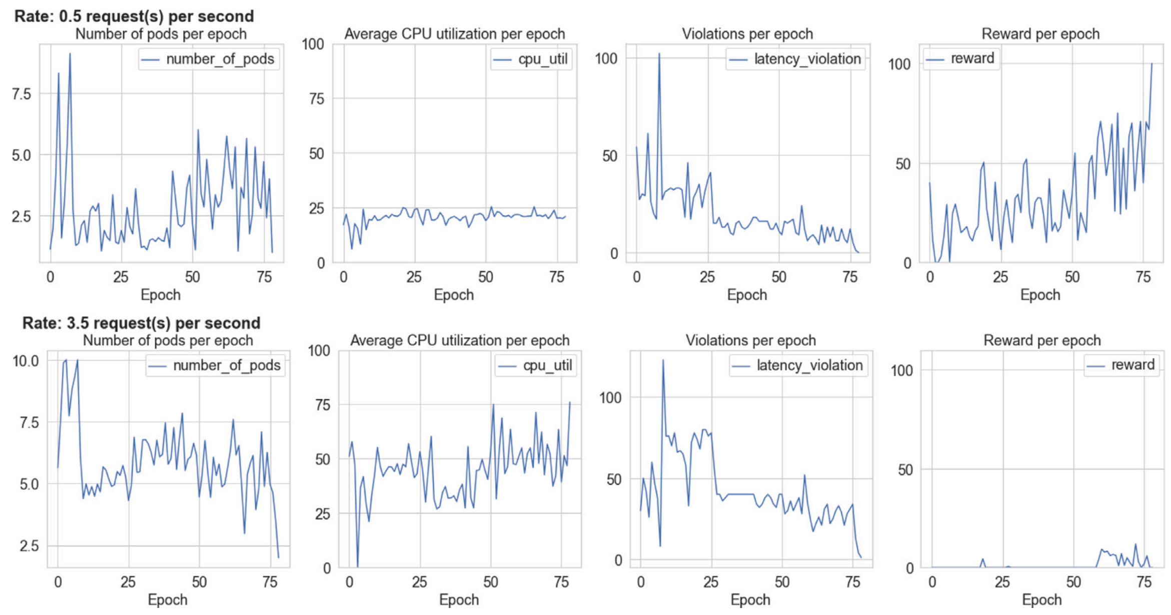

**Fig. 7.** Q-learning Agent Performance Evaluation Results — Real environment (k8s-env-discrete).

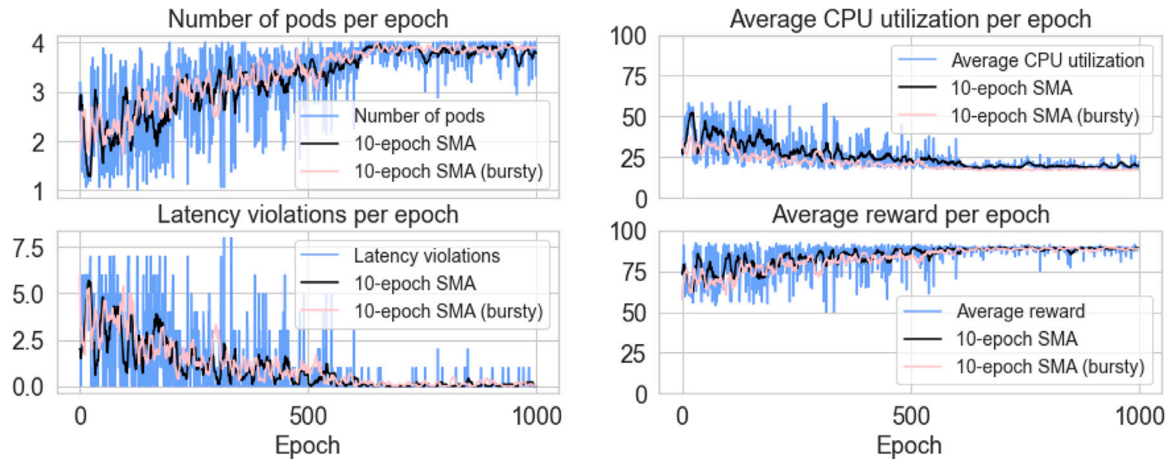

**Fig. 8.** Q-learning Agent Performance Evaluation Results — Simulation environment (k8s-env-discrete).

the aggregated evaluation metrics. This may due to the effort provided by the agent to get accustomed with the dynamicity that is present in the serverless computing environment (e.g., continuously changing workload, enforcement of cooldown periods upon scaling). Once again, similar behaviour is noticed in both types of workload (smooth and bursty scenario).

In the case of the continuous environments and the DQL agent, in Fig. 11 we depict the produced results from the stress test in the real environment (k8s-env-cont-v0) at a low and a high request rate. In both cases, no clear convergence pattern is shown within the first 20 evaluation epochs. Even though the training time was around a week, it seems that the produced data are not adequate to improve the learning status of the agent. The number of pods is reduced and their average CPU usage is increased. Such results are desirable, however they are not combined with a relevant increase in the collected reward and a reduction in the number of SLA violations. For the latter, we consider both latency and throughput rate violations, as they are defined in the relevant environment in Section 3.2.

The improvement in the learning behaviour of the DQL agent upon extended training is confirmed based on the simulation results (k8s-env-cont-v1 environment), as depicted in Fig. 12. Similarly to the Q-learning and the DynaQ+ agents, two types of workload are applied. The first workload is similar to the workload used for training purposes, as detailed in Section 4.3.3. The second workload follows a bursty profile that corresponds to the pattern $\lambda \rightarrow 7\lambda \rightarrow \lambda \rightarrow 7\lambda$. In both cases, a 10-point simple moving average (SMA) of the performance evaluation results is produced. Upon the training of the agent for 300 evaluation epochs, high convergence rates are achieved in all the evaluation metrics. The average reward is increased and stabilized close to 80 for the

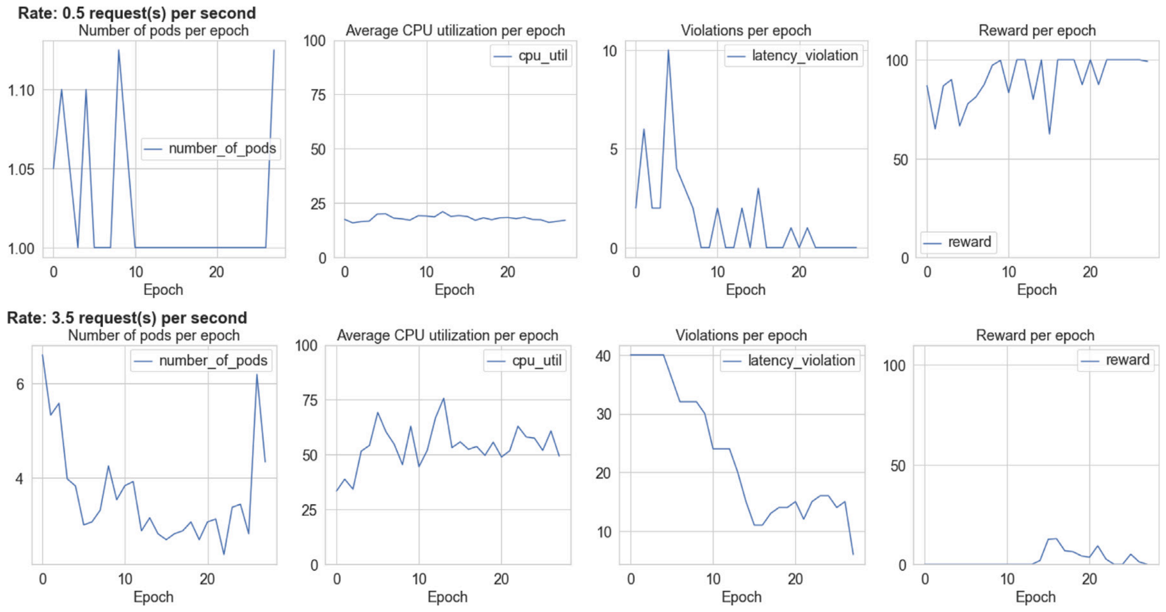

**Fig. 9.** DynaQ+ Agent Performance Evaluation Results — Real environment (k8s-env-discrete).

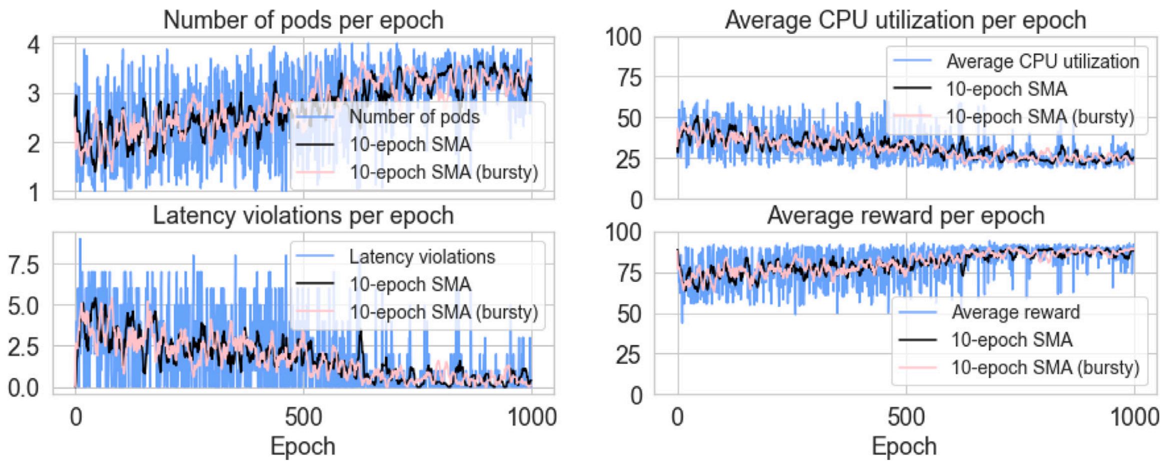

**Fig. 10.** DynaQ+ Agent Performance Evaluation Results — Simulation environment (k8s-env-discrete).

smooth workload and close to 90 for the bursty workload. The SLA violations are significantly decreased, especially in the case of the bursty workload, while the number of the used pods and the average CPU utilization is close enough for both types of workload.

## *5.2. Discussion*

Taking into account the evaluation results that are produced in real and simulation environments, it can be claimed that RL-driven autoscaling can be efficiently applied in serverless computing, upon proper training of the RL agents. The trade-off between the need for conformance to SLA contracts and the reduction of the overall cost for the provisioning of the cloud computing applications based on conservative usage of compute resources can be autonomously managed by the RL agents.

Based on the simulation results, faster learning times are noticed in case of environments with discrete state and action space. It seems that in case of continuous environments, the more accurate representation of the state space leads to much bigger exploration needs and, thus, more strict requirements in terms of training. Achievement of fast learning times is important to enable cloud application providers to apply – in large scale – such solutions for autoscaling of various microservices-based serverless applications that may operate under dynamic and bursty workloads. For instance, the Q-learning and DynaQ+ agents achieve a high reward and close to zero SLA violations in less that 100 epochs, while in a continuous environment, similar learning convergence is observed upon 300 evaluation epochs. Furthermore, in the case of the Q-learning and DynaQ+ agents, a positive learning trend is noticed in

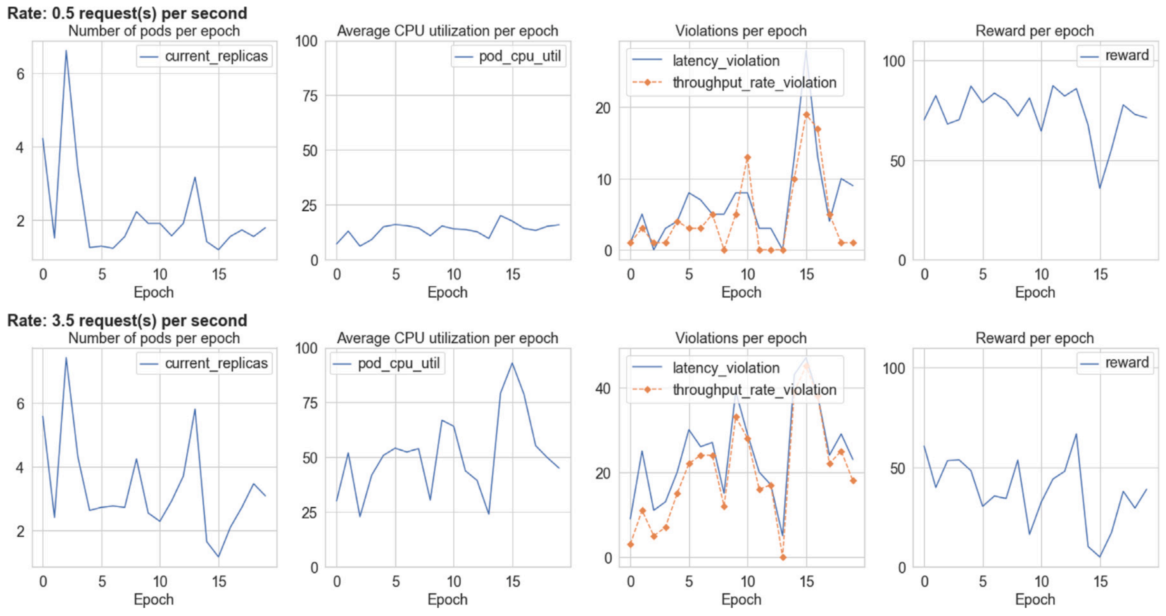

**Fig. 11.** DQL Agent Performance Evaluation Results — Real environment (k8s-env-cont-v0) (e=0.1).

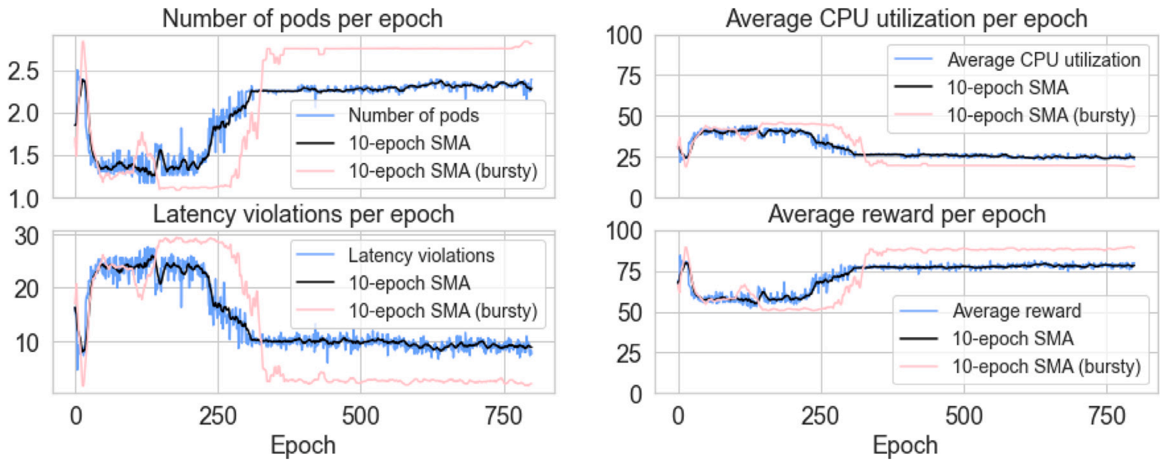

**Fig. 12.** DQL Agent Performance Evaluation Results — Simulation environment (k8s-env-cont-v1) (e=0.1).

both real and simulation environments, while in the case of the DQL agent such a trend was only revealed within the simulation environment. In all cases, similar performance evaluation results were noticed in case of stress testing with smooth and bursty workload profiles.

Regarding the usage of resources and their average CPU usage, various oscillations are present in the diagrams without a clear trend. This is possibly due to the small-scale infrastructure that has been used in the testbed for training purposes. Due to limitations in the available resources, limitations in the maximum number of instances are posed (10 max instances), while the posed workload is also in accordance with these limitations. Realization of extended experiments in larger infrastructure could better reveal the positive learning trend in parallel with the conservative usage of resources. The aforementioned limitations are also transferred in the simulation environment, given that the creation of the distributions per metric is based on the data collected through the real infrastructure.

Differences are also noted in the performance of the RL agents under low and high request rates. This classification is applied in case of the real environments to better depict the peculiarities noticed per RL agent, given the constraints in the number of epochs that we could consider in the experiments. The learning rate is increased in a higher pace in the case of low request rates, while the SLA violations are smaller. These results sound reasonable since the dynamicity of the serverless computing environment is smaller in this case (e.g., less oscillations upon scaling actions and less SLA violations due to the small workload). On the other hand, in case of high request rates, the average CPU utilization seems to be higher. This is also a positive outcome, since the agent learns to better distribute the workload without creating new instances, as the workload increases. This effect could be potentially shown better in case of experimentation in a larger infrastructure.

It could be claimed that the outcomes of this work can constitute the basis for further performance evaluation studies in the future. Towards this direction, it can be considered the application of the proposed solution over further open-source serverless computing platforms and the comparison of their performance, as well as the comparison of the produced results with any relevant datasets that can be made openly available in the future based on the application of autoscaling solutions on open-source or commercial serverless computing platforms.

## 6. Conclusions and open research areas

In the current manuscript we have detailed an approach for tackling autoscaling of functions in serverless computing platforms based on decisions made by RL agents. The approach has been conceptualized and implemented upon the identification of a relevant gap in the existing bibliography towards the development of RL environments and agents for assisting autoscaling mechanisms over real testbeds of serverless computing platforms. A set of RL environments have been designed and implemented, combining discrete and continuous state space environments, while RL agents based on the Q-learning, DynaQ+ and DQL algorithms are developed and applied over these environments. Both the developed environments and the RL agents are made available as open-source code in a Gitlab repository [19]. The objective is to promote the adoption and the extension of the available RL environments and agents by interested researchers, as well as to facilitate extended evaluations and comparisons.

In the current work, a set of evaluation results are presented in real and simulation environments. These results validate the effectiveness of the proposed approach for tackling autoscaling in serverless computing platforms. The advantages of adopting environments with discrete state space are highlighted, given that they require less training time and can be more easily applicable in real scenarios. Simulation environments are also very helpful, since they can be applied over the trained datasets to examine the future behaviour of the RL agents and provide insights regarding their learning behaviour.

A set of open research areas have been also identified. The development and examination of the performance of further RL agents based on algorithms such as actor critic or double Q-learning can be examined. Such an activity can be combined with the creation of pre-trained openAI Gym simulation environments that can be easily adoptable by any interested third party. In this way, a pool of RL agents and environments can be made available for adoption by serverless computing platforms, tackling a diverse set of requirements. Another promising research area for improving the convergence times of the RL agents regards the blending of federating learning and reinforcement learning approaches. By taking advantage of federated learning techniques, serverless computing applications with similar resources usage profiles can shape jointly their optimal autoscaling policy, relaxing significantly the requirements in terms of training times. Finally, the examination of autoscaling patterns in serverless computing application graphs consisted of a number of serverless functions is also a promising research field. Each serverless function can shape its own autoscaling policy considering in parallel the impact of scaling in the rest parts of the application graph, taking advantage of multi-agent based reinforcement learning techniques.

## CRediT authorship contribution statement

**Anastasios Zafeiropoulos:** Conceptualization, Methodology, Validation, Formal analysis, Writing – review & editing. **Eleni Fotopoulou:** Conceptualization, Software, Validation, Formal analysis, Visualization. **Nikos Filinis:** Conceptualization, Software, Validation, Formal analysis, Visualization. **Symeon Papavassiliou:** Conceptualization, Validation, Writing – review & editing, Supervision.

## References

[1] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J.E. Gonzalez, R.A. Popa, I. Stoica, D.A. Patterson, Cloud programming simplified: A Berkeley view on serverless computing, 2019, arXiv:1902.03383.

[2] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C.L. Abad, A. Iosup, Serverless applications: Why, when, and how? IEEE Softw. 38 (01) (2021) 32–39, http://dx.doi.org/10.1109/MS.2020.3023302.

[3] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski, The rise of serverless computing, Commun. ACM 62 (12) (2019) 44–54, http://dx.doi.org/10.1145/3368454.

[4] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, J. Wilkes, Autopilot: Workload autoscaling at google, in: Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20, Association for Computing Machinery, New York, NY, USA, 2020, http://dx.doi.org/10.1145/3342195.3387524.

[5] A. Pimpley, S. Li, A. Srivastava, V. Rohra, Y. Zhu, S. Srinivasan, A. Jindal, H. Patel, S. Qiao, R. Sen, Optimal resource allocation for serverless queries, 2021, arXiv:2107.08594.

[6] C. Lin, H. Khazaei, Modeling and optimization of performance and cost of serverless applications, IEEE Trans. Parallel Distrib. Syst. 32 (3) (2021) 615–632, http://dx.doi.org/10.1109/TPDS.2020.3028841.

[7] S. Kardani-Moghaddam, R. Buyya, K. Ramamohanarao, ADRL: A hybrid anomaly-aware deep reinforcement learning-based resource scaling in clouds, IEEE Trans. Parallel Distrib. Syst. 32 (3) (2021) 514–526, http://dx.doi.org/10.1109/TPDS.2020.3025914.

[8] J.S. Pujol Roig, D.M. Gutierrez-Estevez, D. Gündüz, Management and orchestration of virtual network functions via deep reinforcement learning, IEEE J. Sel. Areas Commun. 38 (2) (2020) 304–317, http://dx.doi.org/10.1109/JSAC.2019.2959263.

[9] L. Schuler, S. Jamil, N. Kühl, Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments, 2020, CoRR abs/2005.14410 arXiv:2005.14410.

[10] CNCF WG-Serverless Whitepaper v1.0, Cloud Native Computing Foundation (CNCF), 2021, URL https://github.com/cncf/wg-serverless.

[11] N. Mahmoudi, H. Khazaei, Performance modeling of serverless computing platforms, IEEE Trans. Cloud Comput. (2020) 1, http://dx.doi.org/10.1109/TCC.2020.3033373.

[12] P. Gouvas, E. Fotopoulou, A. Zafeiropoulos, C. Vassilakis, A context model and policies management framework for reconfigurable-by-design distributed applications, Procedia Comput. Sci. 97 (2016) 122–125, http://dx.doi.org/10.1016/j.procs.2016.08.288, URL https://www.sciencedirect.com/science/article/pii/S1877050916321044.

[13] Y. Garí, D.A. Monge, E. Pacini, C. Mateos, C.G. Garino, Reinforcement learning-based application autoscaling in the cloud: A survey, Eng. Appl. Artif. Intell. 102 (2021) 104288, http://dx.doi.org/10.1016/j.engappai.2021.104288.

[14] M. Imdoukh, I. Ahmad, M.G. Alfailakawi, Machine learning-based auto-scaling for containerized applications, Neural Comput. Appl. 32 (13) (2020) 9745–9760, http://dx.doi.org/10.1007/s00521-019-04507-z.

[15] R. Sutton, A. Barto, Reinforcement Learning | An Introduction, The MIT Press, 1998, URL https://mitpress.mit.edu/books/reinforcement-learning.

[16] H. Zhang, T. Yu, Taxonomy of reinforcement learning algorithms, in: Deep Reinforcement Learning: Fundamentals, Research and Applications, Springer Singapore, Singapore, 2020, pp. 125–133, http://dx.doi.org/10.1007/978-981-15-4095-0_3.

[17] Horizontal Pod Autoscale. URL https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[18] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, 2016, arXiv:arXiv:1606.01540.

[19] E. Fotopoulou, N. Filinis, A. Zafeiropoulos, Repository for the development of the RL agents for autoscaling in serveless computing platforms, 2021, URL https://gitlab.com/netmode/k8s-rl-autoscaler.

[20] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis, Human-level control through deep reinforcement learning, Nature 518 (7540) (2015) 529–533, http://dx.doi.org/10.1038/nature14236.