Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Reinforcement Learning based Resource Allocation Mechanisms in Serverless Clouds

---

Author: Li Zhong (2688794)

1st supervisor: Dr. Adam Belloum

daily supervisor: Jamila Alsayed Kassem

second reader: Dr. Ana Oprescu

Abstract

Serverless computing has been introduced by Google with the release of Google AppEngine, which is equipped with impressive automatic scaling and high-availability mechanisms. There are various serverless platforms after AppEngine, attracting an increasing number of research works in serverless clouds. To improve the quality of service and minimize resource utilization, research in automatic resource allocation has been recently focused on learning-based approaches such as Reinforcement Learning algorithms. Compared to traditional rule-based resource scaling methods, RL-based solutions eliminate human intervention and the generation of rules. The goal of this project is to explore the applicability of RL-based agents and conduct detailed comparisons of RL-based methods and traditional rule-based auto-scalers. In this thesis, we propose one Q-Learning agent to interact with the Kubernetes cluster through discrete states of the system in resource utilization and QoS, actions for setting auto-scaling threshold values, and rewards of state-action pairs. The RL-based agent is trained and tested with simulated workloads and is compared against the Horizontal Pod Autoscaler provided by Kubernetes. The experimental results show that our proposed auto-scaler utilizes fewer resources than the baseline HPA while ensuring the quality of service.

Key words - Serverless Computing, Reinforcement Learning, Auto-scaling, Q-Learning, Kubernetes

Contents

1 Introduction **3**

1.1 Problem Description and Goal 3

1.2 Research Questions 4

1.3 Content of The Report 4

2 Background **6**

2.1 Background of RL algorithm 6

2.2 Background of EPI framework 6

2.3 Related Tools and Concepts 7

2.3.1 Kubernetes and Horizontal Pod Autoscaler 7

2.3.2 OpenAI Gym 7

2.3.3 NumPy 8

3 Related Work **9**

3.1 Rule-based Mechanisms in Serverless Platforms 9

3.2 RL-based Mechanisms in Serverless Platforms 10

3.2.1 Adjust Number of Machines 10

3.2.2 Adjust Resource Usage Thresholds 10

4 Design of Algorithm **12**

4.1 Action Space 12

4.2 State Space 13

4.3 Reward Function 14

4.4 Q-learning Algorithm 15

5 Implementation **17**

5.1 Design of Kubernetes Cluster 17

5.1.1 Architecture of Bridging Function 17
5.1.2 Implementation on Kubernetes Cluster 18
5.2 Implementation of RL agents 20
5.2.1 Implementation of Gym Environment 21
5.2.2 Implementation of Q-learning Allocation Planner 23
5.3 Implementation of Middleware Services 25
5.3.1 Workload Generation 25
5.3.2 Latency Collector 26

6 Experiment 27
6.1 Design of Experiment 27
6.1.1 Workload Design 28
6.1.2 Hyperparameters 30
6.1.3 Evaluation Metrics 31
6.2 Experiment Results 31
6.2.1 Training Process 32
6.2.2 Repetitive Testings and Results 35

7 Discussion 39
7.1 Discussion of Experimental Results 39
7.2 Future Work 40

8 Conclusion 42

Chapter 1

Introduction

1.1 Problem Description and Goal

In recent times, serverless cloud computing or Function-as-a-Service (FaaS) has been proposed to improve elasticity, flexibility, and scalability in cloud computing environments. In contrast to traditional serverful clouds, where resources are allocated and managed by application developers in advance, serverless cloud frameworks implement dynamic resource allocation and management. Infrastructure provisioning and maintenance are handled by the serverless platform and are completely hidden from developers and users [1]. In FaaS frameworks, complex applications are divided into multiple microservices or functions, where client code can be packaged and executed in lightweight containers. The deployment and resource management of such containers can be automated without human intervention.

Most serverless platforms provide default tools to scale the microservices, such as the threshold-based Horizontal Pod Autoscaler (HPA) [2] provided by Kubernetes and the workload-based AWS Auto Scaling [3]. However, those rule-based tools manage resources according to scaling parameters, which require a good understanding of domain knowledge to set accordingly. With the increasing popularity of Artificial Intelligence (AI) in science and industry, a lot of work in automatic resource allocation has concentrated on machine learning based mechanisms such as Reinforcement Learning (RL) in recent times.

RL is especially well-suited for resource management problems due to the sequential nature of the decision-making process [4]. This approach has been applied successfully in serverful cloud environments, where interest in the applicability of RL to the auto-scaling mechanism of serverless clouds is growing. Therefore, this thesis aims to explore and evaluate the application of RL algorithms for recommending scaling decisions in serverless

platforms. We adopt the secure data sharing network, the EPI framework [5], as our testbed. Based on this use case, we design and implement one simple Q-learning based auto-scaling agent to allocate resources according to resource utilization and QoS. Subsequently, we conduct experiments to compare the developed RL-based method with the threshold-based approach in Kubernetes and demonstrate the advantages of the learning-based method over Kubernetes' default scaling policy.

1.2 Research Questions

The exploration and evaluation of RL algorithms for automatic resource allocation on serverless clouds raise several issues when we attempt to answer the following research questions:

- Q1-How to implement and integrate Reinforcement Learning algorithms to recommend scaling within a defined scenario setup?
  - What metrics should be used to monitor the status of the infrastructure and application?
  - What actions should be defined to scale resources?
  - How to design the reward function to make the trade-off between resource cost and performance?
  - What type of the RL algorithm should be adopted in the integration?
- Q2-According to defined metrics, is it efficient to apply the RL algorithms in Kubernetes VFN container scaling, compared to using more conventional rule-based tools?
  - How to design and set up a reasonable experiment environment?
  - How to define metrics to quantify the difference between RL-based agents and more conventional tools?
  - What conventional techniques should be selected as a reference?

1.3 Content of The Report

The rest of the thesis is organized as follows: Chapter 2 provides some background knowledge, which first explains the workflow of RL algorithms, then introduces the EPI framework and

tools used in this project. Chapter 3 presents previous works of applying RL algorithms in automatic resource allocation on serverless clouds. Chapter 4 demonstrates the design of the Q-learning algorithm used in this project, followed by Chapter 5, which illustrates implementation details. Chapter 6 describes the settings of experiments and explains experimental results. Chapter 7 discusses the obtained results and future works and Chapter 8 concludes this thesis.

Chapter 2

Background

In this Chapter, we provide some background knowledge to help understand. We first introduce some basics about Reinforcement Learning algorithms, followed by the introduction of the EPI framework and related tools and concepts mentioned in this thesis.

2.1 Background of RL algorithm

RL implies the development of an agent learning an optimal policy through a certain number of trial-and-error interactions with the environment [6]. In resource allocation scenarios, the agent monitors the current status of the system and performs scaling actions accordingly. After executing actions, it will receive the reward that measures the result of the action, which needs to be as high as possible for the auto-scaler to decide the optimal plan. The agent updates and improves the policy mapping from states to actions through iteratively interacting with the environment. The process of interactions between the RL agent and the environment is illustrated in Figure 2.1.

2.2 Background of EPI framework

The EPI framework [5] is designed to enable secure data sharing within the healthcare context. This framework addresses multiple concerns across different levels: policy level, data level, application level, and network level and we mainly focus on the last network level in this project. To implement security requirements at the network level, developers of the EPI framework design the Virtualised Network Functionalities (VNF), which is responsible for bridging existing security gaps among all the end nodes of the data-sharing session.

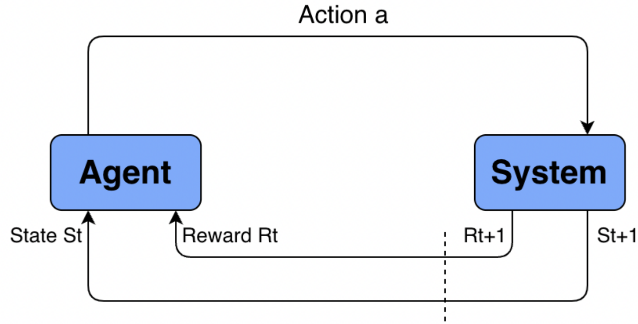

Figure 2.1: The workflow of RL algorithm

The VNF contains chaining of Bridging Functions (BFs), which can be containerized in lightweight containers for easier deployment and execution.

2.3 Related Tools and Concepts

2.3.1 Kubernetes and Horizontal Pod Autoscaler

Kubernetes [7] is an open-source system designed by Google to automate the deployment, expansion, and management of containerized microservices. It supports a variety of different container tools such as Docker [8]. The chaining of Bridging Functions in the EPI framework is containerized by Docker and deployed on the Kubernetes cluster in this project. The Horizontal Pod Autoscaler (HPA) [2] is a default resource-based auto-scaling mechanism provided by Kubernetes to perform horizontal resource scaling. The HPA performs dynamic scaling actions according to certain resource indicators, obtained from the metric server in Kubernetes or third-party services.

2.3.2 OpenAI Gym

In Reinforcement Learning, the agent needs to be operated in one environment and trained by interacting with the environment. However, it is a time-consuming job to manually define and create the environment from scratch. Therefore, it can save a lot of time if there are tools to help compile a new environment. One such module is the OpenAI Gym [9], which equips several sample environments to train various RL algorithms and provides interfaces for users to build custom environments. OpenAI Gym is used in this project to define the

environment that is responsible for all operations of the Kubernetes cluster and interacts with the RL agent.

2.3.3 NumPy

NumPy [10] is an open-source data processing tool, which is mainly used for multi-dimensional data analysis and scientific computing in Python. Data structures in the NumPy library contain multi-dimensional arrays and matrices, and methods to efficiently perform various mathematical operations on an n-dimensional array object are provided. Therefore, the NumPy library is suitable to store the high dimensional state space of the RL auto-scaling agent in this project.

Chapter 3

Related Work

In this Chapter, we discuss the relevant research work on automatic resource allocation in serverless computing environments. Auto-scaling mechanisms in related papers can be divided into rule-based and RL-based.

3.1 Rule-based Mechanisms in Serverless Platforms

Some investigations that compare automatic resource allocation solutions in serverless environments can be found. Palade et al [11] perform comparison experiments with auto-scaling approaches for monitoring various metrics such as the response time, throughput, the ratio of successful responses, and the time cost for deploying machines. Li et al [12] conduct a more detailed performance comparison of auto-scaling solutions, considering both resource-based and workload-based. Performance evaluations in this paper are performed based on the throughput and latency of applications. According to evaluation results, they demonstrate characteristics of different scaling approaches and conclude that there is a need to go beyond the resource utilization based scaling services [12].

Nonetheless, these works do not include learning-based solutions such as RL in comparison to the traditional rule-based approaches provided by serverless platforms. As the number of RL-based auto-scaling techniques in serverless clouds increases, there is a need to analyze the benefit of RL over conventional resource-based or workload-based approaches.

3.2 RL-based Mechanisms in Serverless Platforms

In this Section, we present research on resource auto-scaling with RL algorithms. There are two common types of RL-based mechanisms on serverless clouds: 1) scale by adjusting the number of assigned machines; 2) scale by changing the resource usage threshold values for triggering scaling actions.

3.2.1 Adjust Number of Machines

Zhang et al [13] implement one simple Q-learning algorithm, which combines horizontal scaling and vertical scaling with the consideration that vertical scaling cannot deal with sudden bursts of client requests. They use two open-source ruled-based scaling methods as the baseline and define throughput as the evaluation metric. The experimental results show that the Q-learning agent has a lower resource cost while ensuring the quality of service.

Similar to the work by Zhang et al [13], the auto-scaling mechanism proposed by Qiu et al [14] also combines horizontal scaling and vertical scaling but it is based on the Proximal Policy Optimization algorithm. The RL-based method and the benchmark, open-source rule-based FaaS suites, are tested with both real-world and synthetic workloads. This paper focuses on describing the convergence time and failures of the RL algorithm.

3.2.2 Adjust Resource Usage Thresholds

Instead of directly adjusting the number of assigned machines, some auto-scalers also manage resources by changing the resource usage threshold values of rule-based approaches that trigger scaling actions. Benedetti et al [6] propose one Q-learning based agent to adjust the CPU usage thresholds of the Horizontal Pod Autoscaler (HPA) by Kubernetes and conduct a series of load testing on the system with different fixed CPU thresholds as the base experiment. They compare the Q-learning auto-scaler and the resource-based HPA with fixed thresholds in the latency of applications. The experimental results show that the latency of the Q-learning agent is slightly greater than the minimum latency in the baseline case.

Rossi et al [15] implement one Deep Q-learning network auto-scaler that controls both the CPU and the Memory thresholds. They evaluate the performance from three perspectives, including the CPU and Memory utilization of the infrastructure and response time of applications. The Deep Q-learning agent is compared to the HPA with different fixed threshold values and one model-free RL-based method. Based on the comparison results,

this paper analyzes and demonstrates the merits and weaknesses of different approaches. In contrast to work by Rossi et al in [15], Khaleq et al [16] consider only RL-based solutions and conduct a detailed comparison of different types of RL algorithms, which are implemented to change the resource usage threshold of the HPA in kubeless environments. The experiments are conducted with both Memory-intensive and CPU-intensive applications to depict the differences among various RL algorithms in different types of applications.

To conduct more detailed comparisons between RL-based methods and rule-based approaches in resource scaling on serverless clouds, we design a simple Q-learning agent based on specific characteristics of the use case to set the thresholds of rule-based auto-scalers. We use the Kubernetes HPA as the baseline and find the optimal configuration to compare it against the Q-learning agent in both resource utilization and quality of service.

Chapter 4

Design of Algorithm

In the previous chapters, we present the auto-scaling problem, give an overview of different types of mechanisms to deal with it, and focus on RL-based methods for automatic resource allocation. We also provide some context about the task of auto-scaling, with special emphasis on serverless environments.

In this Chapter, based on the context provided previously we design one Reinforcement Learning based approach to recommend scaling decisions in the EPI scenario. As discussed in Section 3.2, we want to automatically allocate resources in the chaining functions to minimize resource consumption while satisfying the Service Level Agreement. In this project, we design scaling actions and conduct experiments on one of those Bridging Functions. After several rounds of load testings in advance, we found that the Firewall function utilizes most resources, therefore we choose the Firewall function as the target function to perform horizontal scaling actions on.

In the next Sections, we define basic elements of Reinforcement Learning algorithms according to characteristics of the EPI framework and Firewall function and choose one specific type of RL algorithm to implement.

4.1 Action Space

As discussed in Chapter 2, there are two types of resource allocation strategies, including horizontal scaling and vertical scaling. In this thesis, we adopt the horizontal method and scale out or in by adjusting the number of assigned machines for the target service.

Considering that the Firewall function is CPU intensive, we want to monitor and control the CPU usage of assigned machines at runtime to make more effective scaling decisions.

Hence, we define the scaling action space to change the threshold of CPU usage for triggering a scaling event and utilize the Horizontal Pod Autoscaler(HPA) to perform scaling actions according to the threshold. In this way, we can control the CPU usage of each machine and the number of machines at the same time.

The threshold of CPU utilization is represented by a percentage from 1% to 100%. From [17], large state and action spaces pose a problem in RL tabular methods, not only because of the memory required for large tables, but also because of the time and data required to fill them accurately. Therefore, to manage a simpler state and action space, we discretize the range of the threshold of CPU usage to five values on the basis of Five-Point Likert scale [18]. Discrete thresholds of CPU usage are {20%, 40%, 60%, 80%, 100%} and the action space is to decrease the threshold by 20%, no action, or increase it by 20%, which changes the threshold with one step duration.

4.2 State Space

The state space of a Reinforcement Learning algorithm consists of metrics about the environment it wants to interact with. In auto-scaling scenarios, the RL algorithm needs to make satisfactory trade-off between resource cost and SLA. Therefore, states of one RL auto-scaling agent should contain information about both resource utilization of the infrastructure in CPU utilization and the number of machines, and performance of applications in response time of requests. In this Section, we will define four discrete states as the basics of the auto-scaling strategy for the Firewall function in the EPI framework, which are listed in Table 4.1.

The first two states, CPU utilization and the number of active pods, represent resource cost of the infrastructure in two perspectives. As discussed before, the Firewall function is CPU intensive, therefore we focus on the CPU usage of this function and ignore memory resources in the thesis. The CPU utilization metric quantifies the average percentage of CPU usage of all active Firewall pods and presents the actual cost in CPU of Firewall function. We also monitor the number of deployed Firewall pods and include it in the state space. This metric is necessary for horizontal scaling, where actions of scaling in or out are implemented by increasing or decreasing the number of containers.

We define the response time of requests to measure the performance of applications. From the literature, latency and throughput are the most common metrics to quantify the performance of client response applications [19]. We conducted a series of load testings to

Table 4.1: Description of discrete metrics in the state space of RL algorithm

Name	Description	Unit	Type	Discrete space
CPU utilization	Average percentage of the CPU usage of active Firewall pods	Percentage	Resource cost	0%–20%, 20%–40%, 40%–60%, 60%–80%, 80%–100%, 100%-150%, greater than 150%
Number of active target pods	The number of active Firewall pods	Piece	Resource cost	1-2, 3-4, 5-6, 7-8, 9-10
Average Latency / SLA latency	The ratio of average response time to the SLA response time	Percentage	QoS	0%–20%, 20%–40%, 40%–60%, 60%–80%, 80%–100%, 100%-150%, greater than 150%
HPA threshold	Current threshold of the Horizontal Pod Autoscaler for Firewall function	Percentage	None	0%–20%, 20%–40%, 40%–60%, 60%–80%, 80%–100%

observe changes in these two metrics and found that there is little fluctuation in throughput under different loads in this scenario. Therefore, we use latency to quantify the performance, where influences of changing loads are more obvious.

In Section 4.1, we define the action space of the auto-scaling agent to change the threshold of HPA provided by Kubernetes and the agent must monitor this metric to make scaling decisions. We design the discrete state space according to the Five-Point Likert Scale, where the percentage value between 0% and 100% is divided into five parts. For values beyond this range, we divide them into the range from 100% to 150% and above 150%.

4.3 Reward Function

The reward function of RL algorithms in resource allocation defines the relationship between SLA parameters of applications and resource utilization of the infrastructure. The immediate reward measures the performance of the entire system on the resulting new configuration. One effective definition of reward function steers the RL agent towards better performance with higher utilization of resources.

The entire reward(R_{all}) at time t is calculated with the weighted sum of reward in resource cost(R_{res}) and reward in performance(R_{perf}), which is shown in Eq 4.1:

$$R_{all} = \alpha * R_{res} + \beta * R_{perf} \tag{4.1}$$

Hyperparameters α and β are used to control the importance of these two factors in the

entire reward. The sum of these two parameters is fixed to be 2.5 and the whole reward ranges from 0 to 25 based on the setting in the research by Grunitzki et al [20].

To compute the reward, R_{res} is estimated using Eq 4.2:

$$R_{res} = \frac{Pod_{maximum} - Pod_{deployed}}{Pod_{maximum} - 1} * 10 \tag{4.2}$$

From this equation, rewards in resource cost decrease as the number of assigned pods increase to punish growing resource utilization. The maximum number of pods is greater than one by default to avoid the error of dividing by zero.

R_{perf} is estimated based on the latency of applications using Eq 4.3:

$$R_{perf} = e^{-p*W_{LAT}*\frac{LAT_{AVG}}{LAT_{SLA}}} * 10 \tag{4.3}$$

$$W_{LAT} = \begin{cases} 0.3, & LAT_{AVG} < LAT_{SLA} \\ 10, & LAT_{AVG} >= LAT_{SLA} \end{cases} \tag{4.4}$$

In Eq 4.3, W_{LAT} is the weight to control the influence of SLA-Violation in latency on the reward computation process. According to the settings in [21], the weight parameter for the ratio of average latency to SLA latency is set to 10, which incurs a huge penalty for SLA violation. The exponential function is to make values on the same scale as the resource cost reward, i.e. 0 to 10. The p factor is a constant number to control the sharpness of e function.

4.4 Q-learning Algorithm

Temporal-Difference (TD) learning is one of the central ideas in Reinforcement Learning [17] and has the capability of being able to make predictions incrementally by bootstrapping the current estimate onto previous estimates [22]. From literature works in Chapter 2, Q-learning is a TD method widely used in RL, which has been utilized in the proposed model of this work. Algorithm 1 explains the Q-learning method, where the updated rule is defined as:

$$Q(s, a) = Q(s, a) + \alpha * [r + \gamma * max_{a'}(Q(s', a')) - Q(s, a)] \tag{4.5}$$

Where the discount factor γ controls the weights of the largest future reward after a transition to state s' in the whole reward, and the learning rate α determines the speed at which the model overrides old information with the newly acquired information.

Algorithm 1 - The process of Q-learning algorithm.

Input: State $S = \{s_1, s_2, ..., s_n\}$, Action $A = \{\text{scale_in, scale_out, no_change}\}$, the Q-Value table (s, a)

Output: Updation of Q-Value table (s, a)

Require: Learning rate (α) and discount rate (γ)

Initialize $Q(s, a)$ arbitrarily;

Observe the current state s;

Repeat

Choose partial action a_i from state s (ε-greedy policy);

Take action a, observe reward r and next state s';

$Q(s, a) = Q(s, a) + \alpha * [r + \gamma * max_{a'}(Q(s', a')) - Q(s, a)]$;

$s \leftarrow s'$;

Until converged.

The Q-learning algorithm takes the state space, the action space and one empty look-up form table as inputs and outputs one policy mapping state-action pairs to Q-values. The algorithm is explained as follows:

1) Initialize the Q-values: the RL approach captures historical information of the system into a lookup Q-values table. Each entry of this Q-table describes the benefit of taking action a while in state s and is updated during the learning process.

2) Choose an action: the Q-learning agent needs to explore states to learn from the system environment and exploit knowledge obtained to take better actions at the same time. Therefore, Q-learning utilizes the ε-greedy policy to choose actions, which selects the action with the greatest reward with probability 1-ε and chooses random or non-greedy actions with probability ε to explore non-visited states.

3) Calculate reward value: The agent receives the current state s of the system and calculates reward r based on the reward function defined in Section 4.3.

4)Update the Q-value: The algorithm obtains a reward, calculates a new $Q(s, a)$ value for this state-action pair after taking action a based on formula 4.5, and switches to state s'.

5) The process of stages 2-4 will continue iteratively until the algorithm converges, where the Q-table will map all states to actions with the best Q-value.

Chapter 5

Implementation

In this Chapter, we first introduce the structure and workflow of Bridging Functions of the EPI framework and the implementation of the proposed scenario in Section 2.2 on the Kubernetes cluster. Next, we illustrate the design of RL agents, including: (1) interactions with Kubernetes to collect states; (2) change of the Horizontal Pod Autoscaler (HPA) threshold; (3) modifications of the model in an iterative manner to learn scaling strategies. Finally, we introduce the implementation of some middleware services in the entire architecture.

5.1 Design of Kubernetes Cluster

In order to implement and integrate the Q-learning algorithm to recommend scaling, we need to define and set up experimental scenarios in advance. The experimental scenario is based on bridging functions(BFs) and domain servers in the EPI framework and deployed on the Kubernetes cluster. In this section, we introduce the implementation details of the experiment testbed:

- Structure and flowchart of BFs
- Installation and configuration on Kubernetes

5.1.1 Architecture of Bridging Function

Bridging functions contain chaining of three services, including the encryption and decryption of requests and firewall service that allows and redirects traffic. These three chaining functions

are managed by the proxy service that receives requests from clients and leads requests to go through BFs. The pipeline of bridging functions is illustrated in Figure 5.1.

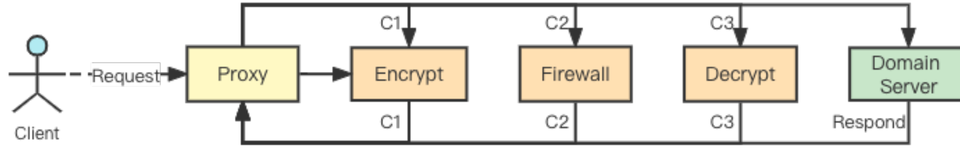

Figure 5.1: Workflow of Bridging Functions

Clients send requests to the proxy server, which runs a chaining of Bridging functions as described in Figure 5.1. Each request sent to the proxy server goes through the following function pipeline: (1) the encrypt service to encrypt data; (2) it is redirected to the firewall server to determine if the request is going to the legal address defined by the privacy policy; (3) it is sent to be decrypted if the destination of the request is valid. After the request has passed all three bridging functions, it can connect to the domain server and start transferring data. All functions are packaged as Docker images and published on Docker Hub[1], therefore can be easily deployed on the k8s cluster by filling the container URL in the pod configuration file.

5.1.2 Implementation on Kubernetes Cluster

Architecture of Kubernetes Cluster

Architecture design of the Kubernetes cluster is presented in Figure 5.2, where the basic component is the node. In this project, we deployed three VMs into the cluster. Each VM has 2 CPUs, 2 GB of RAM, and a 20 GB disk with Debian installed. Therefore, the cluster consists of one master node and two worker nodes. Considering that the Q-learning agent needs to interact with and reconfigure the cluster, it is more convenient to deploy it on the master node to access the configuration file.

Both the Q-learning agent and the threshold-based method need metrics about resource utilization of target pods to make scaling decisions. In this infrastructure, we utilize the metric server[2] provided by Kubernetes to monitor the states of pods and deploy it on the

[1] `https://hub.docker.com/`

[2] `https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/`

master node. The metric server monitors and provides the CPU usage of each pod, therefore extra calculations are needed to obtain average CPU utilization. For the sake of simplicity, we directly obtain this metric from the HPA interfaces, which obtain the CPU usage of every pod from the metric server and automatically calculate the average CPU utilization.

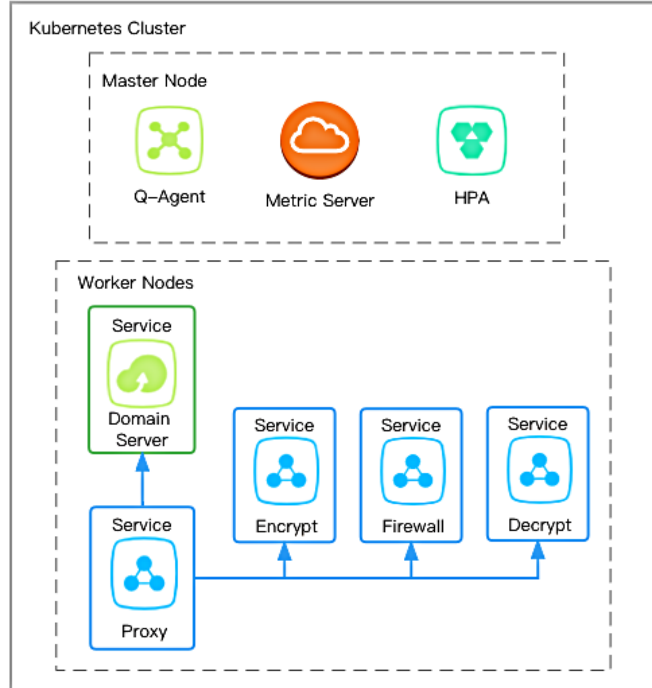

Figure 5.2: Architecture of Kubernetes Cluster

As stated in Section 4, we make scaling decisions on firewall function in this experiment and deploy one Horizontal Pod Autoscaler for the firewall pod. The HPA monitors average the CPU utilization of all target pods through the metric server and performs scaling actions through its default formula defined by Kubernetes according to the threshold configured by the Q-learning agent.

Configuration of Bridging Functions

To save resources on the master node and improve the efficiency of training Q-learning algorithm, all custom pods are assigned to worker nodes during runtime. Service components are deployed for each function to map Pod ports to Node ports, exposing ports to external networks and accepting traffic from outside of the infrastructure. Clients can access the proxy function and connect to domain servers through Node Port mapped by services.

Functions except for the target one, i.e. firewall function, are supposed to be allocated sufficient resources to ensure that the target function is the only factor affecting the performance of applications. After rounds of stress testings with different request rates, we found that because of the limits of the Kubernetes cluster, all functions utilize less than 100 millicores CPU, therefore we set the request CPU parameter of functions except for the target one to 100 millicores and deploy one pod for each function.

Because of the limits of Kubernetes, the maximum CPU utilization of the firewall function is around 100 millicores. As the maximum number of firewall replicas defined in section 4 is 10, we set the requested CPU of one firewall pod to 18 millicores and the CPU limit to 20 millicores to ensure that when the number of replicas reaches the maximum value, this function still has around 50% idle CPU to handle unexpected situations.

5.2 Implementation of RL agents

In this section, we will introduce the pipeline of RL agents and implementation details,

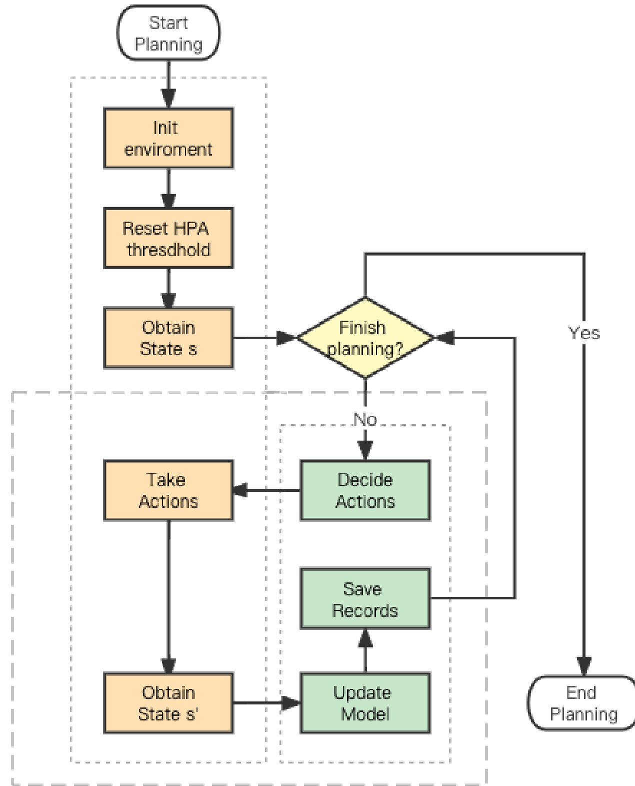

Figure 5.3: Complete workflow of the Q-learning agent

based on one open-source framework[1]. The Q-learning scaling agent consists of two components:

- Gym environment
- Q-learning scaling planner

The Gym environment component is responsible for interacting with and reconfiguring the Kubernetes cluster and exposes interfaces to the planner. The Q-learning allocation planner monitors the status of the infrastructure and applications and performs scaling actions via calling environment interfaces. The complete workflow of the Q-learning agent is depicted in Figure 5.3.

5.2.1 Implementation of Gym Environment

In the infrastructure with pre-deployed Kubernetes cluster and custom functions, all functions interacting with the cluster are encapsulated in one Python[2] class file and registered as one Gym environment[3]. The Gym environment is built as one custom package and can be installed with the pip[4] package installer.

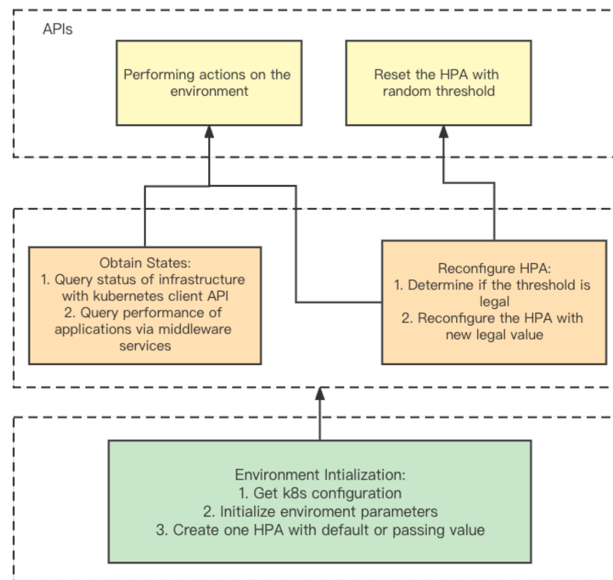

Figure 5.4: Architecture of Gym environment

[1] https://gitlab.com/netmode/k8s-rl-autoscaler
[2] https://www.python.org/
[3] https://www.gymlibrary.ml/
[4] https://pypi.org/project/pip/

From the architecture described in Figure 5.4, the environment exposes two APIs to the allocation planner, including the execution of scaling actions and reset action on the environment. These two API functions are based on two private functions in this environment class: 1) obtain the status of the infrastructure and performance of the application; 2) reconfigure the threshold of HPA. Details about the workflow of the Gym environment are described in next sections, which start with the initialization.

Initialize Environment

During the initialization of the environment, four types of variables need to be provided:

- Variables to define one base DiscreteEnv[1]: dimensions of the state space and the action space
- Variables about the target HPA: the pod name it scales on, initial threshold, the range of the number of pod replicas
- Variables about the status of applications: the SLA latency, URL to query the latency
- Variables about training: waited time duration to collect states after taking actions

The environment can be successfully created when all variables are defined correctly.

Execute Scaling Actions

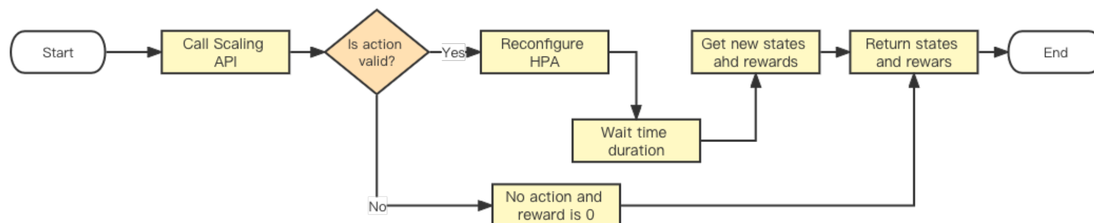

Figure 5.5: Pipeline of performing scaling action by Gym environment

The environment provides one API for the scaling planner to reconfigure the threshold of HPA and obtain the reward of this scaling action. The execution of scaling is presented in Figure 5.5 and consists of three stages:

[1]`https://github.com/openai/gym/blob/master/gym/spaces/discrete.py`

- Decide if the scaling action is valid: one valid threshold is within the range from 20% to 100% based on the definition in Chapter 4, therefore actions that set the threshold out of this range will not be executed, and the function returns the current system status and zero rewards
- Reconfigure the Horizontal Pod Autoscaler: if the action is valid, the environment will change the HPA to the new threshold by modifying the configuration file of HPA with Kubernetes client API
- Calculate Reward: after waiting the predefined time duration, the environment will collect new states of the infrastructure and application, calculate rewards and response the scaling planner

The Q-learning agent updates its Q-values table based on new reward and make next scaling decisions according to the latest state. It is safer and simpler for the Q-learning agent to execute scaling via calling APIs instead of directly interacting with the cluster.

Reset Environment

The reset of the environment is encapsulated in one public interface, which is executed at the beginning of each training epoch and helps the Q-learning agent to explore more states. The reset function will reconfigure the HPA with the random threshold from [20%, 40%, 60%, 80%, 100%].

5.2.2 Implementation of Q-learning Allocation Planner

The Q-learning scaling planner consists of one Q-values table that stores all Q-values and functions that control scaling actions and the training process. The Q-table is implemented with one NumPy[1] array. NumPy is one Python package that is responsible for manipulating high dimensional data and is suitable for storing Q-values with high dimensional discrete state space in this Q-learning algorithm. Historical records are stored in CSV files to monitor the performance of the agent and provide guidelines for convergence or improvement.

Two functions of the Q-learning planner contains one that is responsible for choosing and performing scaling actions and another one that is for updating the model and recording historical data. Pipeline of the allocation planner is illustrated in Figure 5.6.

[1] `https://numpy.org/`

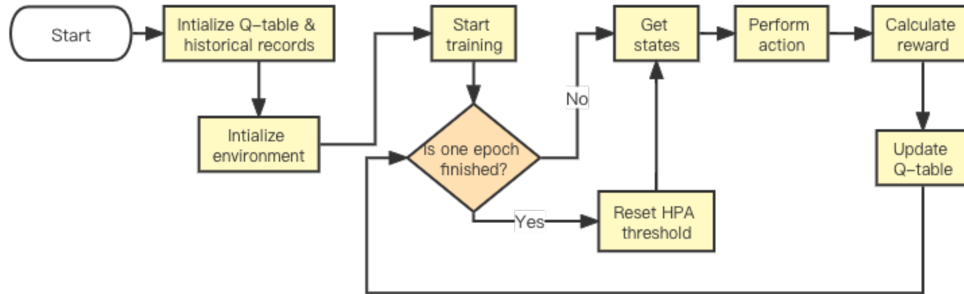

Figure 5.6: Workflow of the Q-learning planner

The planner first creates one NumPy array as the Q-table and one CSV file to store historical records. Next, it initializes the Gym environment to interact with the Kubernetes cluster or the infrastructure. After finishing preparation work, it enters multiple training epochs. One epoch contains several rounds of learning, where the agent will perform one action and update the Q-table once within one round. The number of training epochs, number of learning rounds within one epoch and time duration between each round are hyperparameters and will be discussed in section 6.1.3.

Within one learning round, the workflow of the agent is as follows:

- The agent determines if the current round is the beginning of one new training epoch and resets the threshold of HPA at the first round.
- The agent obtains the status of the infrastructure and the application and chooses one action to perform; the action is either random or the one with maximum Q-value, where the probability is controlled by the ϵ parameter
- The agent executes the scaling action and receives the corresponding reward via interfaces provided by the Gym environment
- The agent updates the Q-value and saves historical record, where the formula of calculating Q-value is discussed in Chapter 4

The end of Q-learning is controlled by humans, therefore it continues the training process in an iterative manner until interrupted.

5.3 Implementation of Middleware Services

5.3.1 Workload Generation

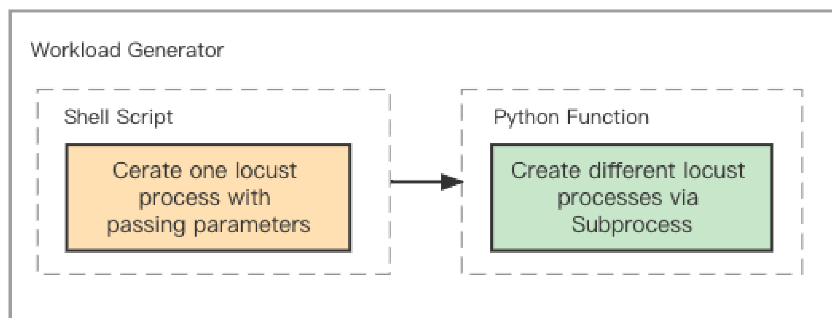

Figure 5.7: Architecture of workload generator

To train the Q-learning agent and compare the performance of two scaling mechanisms, we need to generate requests to produce traffic in the entire system. Requests that pass through all chaining services are the basics of generating traffic. The request is based on the client Python file in SOCKS toolkit[1] for the EPI framework with modifications on the volume of transferring data. The connection path of one client request is illustrated in Figure 5.1 in Section 5.1.1. After it is connected, it starts transmitting data to generate traffic in the network.

Besides the definition of one request, the workload generator contains two components, one Shell script and one Python function, as presented in Figure 5.7. The Shell script is responsible for creating one Locust[2] process to generate requests. Locust is one load testing tool for sending requests to the specific host with given transmission rates and other parameters. The Shell script accepts parameters and runs Locust commands to start one Locust process.

Executing the script once only transmits requests with one rate, therefore we need to design one function to automatically generate changing workloads with different rates. We implement one Python function with the help of Subprocess[3] package, which is used to create processes in Operating System. We utilize Subprocess interfaces to create and manage different processes to execute the Locust script with different request rates in an iterative

[1] `https://github.com/epi-project/socksx`

[2] `https://locust.io/`

[3] `https://docs.python.org/3/library/subprocess.html`

manner to produce changing workloads.

5.3.2 Latency Collector

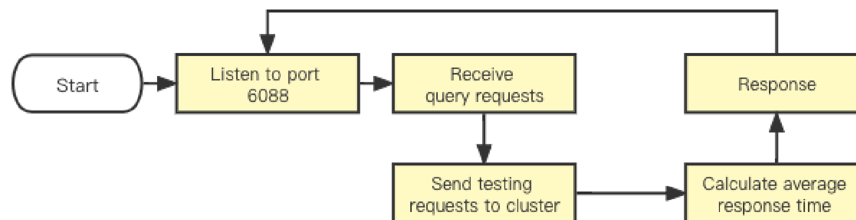

Figure 5.8: Workflow of the latency collector

The latency or response time of client requests is defined as one state in the Q-learning algorithm to measure the performance of applications. This metric cannot be obtained through interfaces provided by Kubernetes and needs to be collected by custom functions. We implement one latency collector function with Python and deploy it on one cloud VM. The structure of the latency collector is illustrated in Figure 5.8 and the pipeline is as follows:

- Run command to execute the Python file and starts the Web server that listens to port 6088 of the machine
- The Gym environment sends requests to IP:6088/latency triggers the collector server to handle requests
- The collector calls private handling function and sends two requests to EPI framework on Kubernetes cluster
- After getting all responses, the server calculates average response time and responds to the Gym environment

The latency collector server listens to requests forever after getting started and can only be halted with interruptions.

Chapter 6

Experiment

The experiment consists of two periods, a training period and a testing period. During the training period, this Q-learner will learn actions to take under different conditions and save the corresponding Q-value. In order to train the Q agent and test its performance, we need to design simulated workloads, both for testing period and training period. We also need to set reasonable hyperparameters of the Q-learning agent before starting the learning. The evaluation of the algorithm is designed to compare and contrast the behaviors between auto-agent and threshold-based methods. Therefore, we need to come up with effective metrics to quantify their performance.

Before we test the Q-learner and obtain experimental results, we first need to train the Q-learning agent to make it converge. The training process is illustrated in Section 6.2.1. After finishing the training, we run multiple repetitive rounds of testing to collect the performance of the Q-learning algorithm. According to those results, we choose one fixed threshold-based method as the reference to compare with Q-agent and run the same repetitive testing on it. The comparison results of two types of strategies are presented in Section 6.2.2.

6.1 Design of Experiment

In this Section, we provide detailed discussion about different components of the experiment, which include three main components:

- Simulated workload
- Hyperparameter setting

- Evaluation metrics

6.1.1 Workload Design

In the real world, the arrival rate of client requests to servers remains changing, thus it is reasonable and necessary to use changing workloads within a time duration to investigate the performance of auto-scalers in realistic scenarios.

There are two types of workloads: 1) historical workload collected from realistic environments; 2) simulated workload generated from some patterns. Undoubtedly, the history of realistic workloads is more accurate in performing realistic patterns than simulated workloads, however, it is more stable and contains less changing patterns in the same time period compared to artificial workloads. Therefore, training and testing by simulated workloads is able to save a lot of time, whereas realistic one can cost a long time to include all patterns. Besides, through collecting historical workloads and extracting all kinds of patterns from them, the accuracy of simulated workloads is improved to a large extent.

During this experiment, we utilized two simulated workloads, one is for training the auto-scaling agent and the other is for testing the performance of agent and threshold-based methods.

Training Workload

We designed two scenarios to train the Q-learning agent, the first period is performed on several fixed workloads and the second training is on multiple rounds of changing workloads. First of all, we conducted some experiments in advance to choose a set of number of requests per second to consist of the fixed workload series. Selected values are typical ones that correspond to specific numbers of pods when CPU utilization is 100%, the corresponding between arrival rates and resource costs is shown in Table 6.1.

Table 6.1: Relationship between request rate and required resource

Request rate per second	Number of pods
3	[2,3]
8	[3,4]
10	[5,6]
25	[9,10]

After the agent has explored all possible situations under each fixed arrival rate and

saved corresponding patterns, we began to train the agent under one changing workload. The basis of this workload is one stable workload with sin function tendency [23], where a ratio of 0.2 bursts are inserted into the workload randomly. The tendency of this workload is presented in Figure 6.1, where the left one is the original workload and the right one is inserted with random bursts.

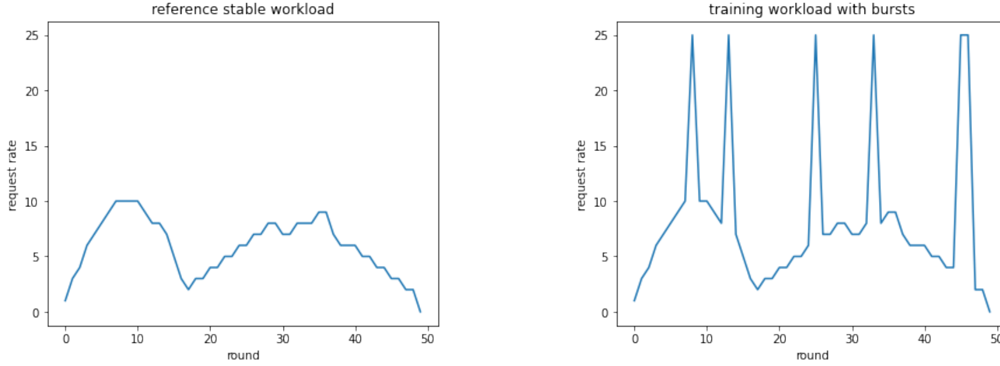

Figure 6.1: Changing workload during training

Testing Workload

After the convergence of the RL algorithm, it is necessary to perform both the auto-scaler and threshold methods on one different changing workload to compare with each other and observe the generalizability of the Q-learning model. We designed this testing workload with the same stable tendency as in the training workload, but is inserted with different random bursts and sudden drops of request rates, which is shown in Figure 6.2.

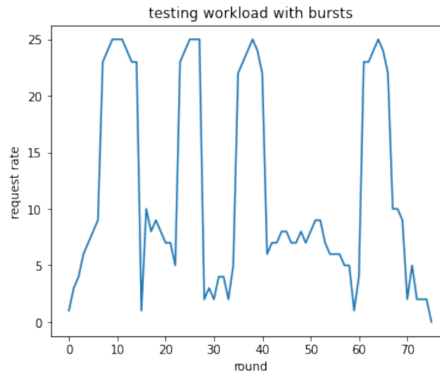

Figure 6.2: Changing workload during testing

6.1.2 Hyperparameters

Hyperparameters that control the training process of RL algorithms are important factors in the experiment and need to be determined with careful consideration in advance. Hyperparameters and their values are listed in Table 6.2.

The first parameter we need to set is the time interval between two scale decisions made by the auto-scaling agent. After the scaling agent takes action and changes the threshold of the horizontal scaler, it will wait for the horizontal scaler to adjust the number of machines and obtain the latest states to calculate rewards, where the entire work costs around 1.2 minute. We set the request rate of training workload to 2 minutes to smooth the change rate and decrease the time duration of each arrival rate to increase change rate and the agent can learn two different types of changing rates. The time duration of one complete workload is determined by the predefined number of request rates in the workload and time duration of one rate.

In the beginning of each training epoch, the horizontal scaler will be reset to a random number to remain random and explore as many states as possible. We set the time duration of one training round as 100 minutes to make sure the reset behavior will not be performed within one workload and influence the training performance.

We refer from other papers to set the epsilon parameters. As stated before, epsilon controls the probability of selecting an action randomly rather than the one with maximum Q-table value. We set the initial value to 0.97 for exploring a large range of states and actions and the minimum value to 0.1 to keep random exploration but utilize the obtained patterns at the same time.

Table 6.2: Descriptive statistics of thresholds during repetitive testing

Hyperparameter	Value
time interval between two agent actions	1.2 minutes
time duration of each request rate	2 minutes for training; 1 minutes for testing
time duration of complete workload	100 minutes for training; 76 minutes for testing.
number of rounds within each epoch during training	87
initial epsilon	0.97 [24]
minimum epsilon	0.1 [24]

6.1.3 Evaluation Metrics

In order to compare and contrast the performance of Q-learning agent and threshold-based methods, we need to define useful metrics and statistics to quantify their performance from different perspectives. Measurements are defined in Table 6.3.

First of all, quantifying resource utilization is necessary, considering that the goal of applying scaling strategies is to save resource cost, and we defined two corresponding metrics. The first one is the number of deployed pods, which demonstrates differences in scaling behaviors between two methods in an intuitive manner. The second one is wasted CPU, which quantifies the resource utilization in a more fine grained aspect. This metric is calculated by summing up the difference between requested CPU and utilized CPU in all pods when the CPU utilization is below 100 percent.

Besides, scaling mechanisms are supposed to satisfy service level agreement while minimizing resource cost, thus metrics about performance are indispensable. As discussed in Section 4.1, we defined latency of requests as one element in the state space of RL agents and we reuse it as one metric to quantify the performance of scaling methods.

Historical data cannot reveal deeper tendencies or clues, therefore we adopt some statistics to describe experimental data and analyze evaluation metrics. We calculate mean values of all metrics within one training round and among three rounds. Standard deviations of mean values among three rounds are calculated to present the stability of these two methods.

Table 6.3: Evaluation metrics and descriptive statistics

Metric	Unit	Descriptive statistic
number of pods	piece	mean, standard deviation
wasted CPU	milli	mean, standard deviation
latency	second	mean, standard deviation

6.2 Experiment Results

Evaluation results in the training period of Q-learning agent and in the stress test scenario of RL agent and threshold-based methods are presented in this section. Historical results during training or testing period are illustrated with figures and contain five perspectives:

- Number of pods
- Wasted CPU

- Latency
- Rewards during training
- HPA thresholds of Q-learning agent

Those results show changes in details and give guidelines about the convergence of the Q-learning agent. Statistical analysis of results are collected and calculated based on the definition of evaluation metrics in Section 6.1.3 and will be presented with tables.

6.2.1 Training Process

As discussed before, we designed two simulated environments to train the Q-learning agent, consisting of several fixed request rates and multiple rounds of one changing workload. The Q-learning agent is based on elements defined in Section 4 and trained with hyperparameters stated in Section 6.1.2.

Several Fixed Rates Figure 6.3 depicts the results produced during the training phase in the simulated scenarios at different request rates per second. In case of all request rates, in the initial set of epochs, higher latency 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. For example, higher rewards are observed after round 80 than rounds before are observed with request rate as 10. The steady change in rewards and average CPU utilization from Figure 6.3 illustrates the convergence of the Q-learning algorithm.

One Changing Workload After the convergence of the Q-learning agent, we are supposed to continue training on the changing workload. In Figure 6.4, we depict the produced results during the training period under multiple rounds of the changing workload defined in Section 6.1.1. We generated five rounds of workloads in total, where each round lasts for 100 minutes.

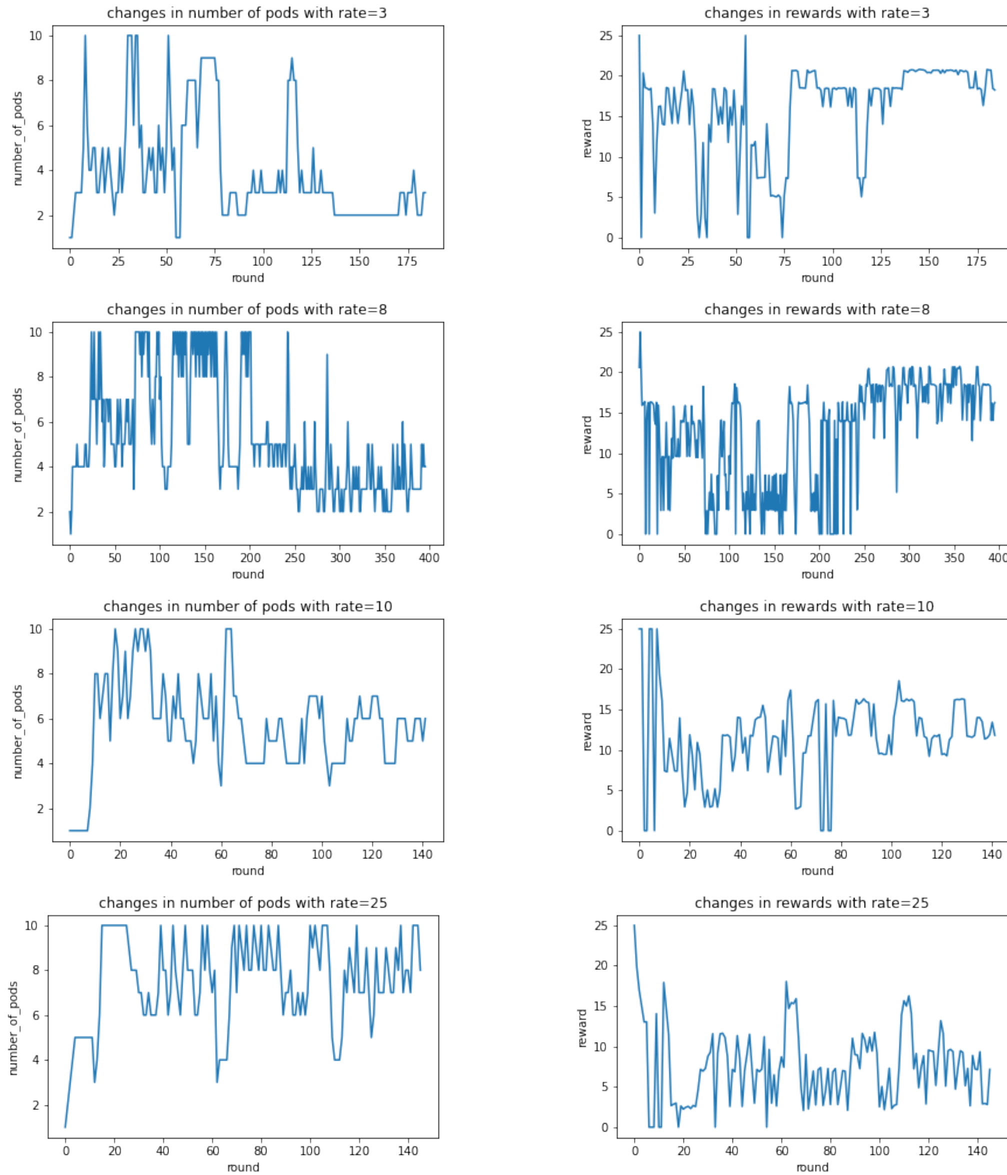

Figure 6.3: Changes of Q-learning agent during training with fixed rates

From Figure 6.4, in the beginning of training, actions are random and changes in rewards or threshold setting are fluctuant. After the first three rounds, changes between each round become similar to each other and display some patterns. It can be seen that the threshold of HPA decreases when the request rate increases and rises when the request drops. The reason is that the Q-learning agent is making decisions on the trade-off between resource costs and performance. Therefore, when the request rate becomes high, it decreases the threshold to force HPA to increase the number of pods in order to decrease the latency. When the arrival rate starts dropping, it increases the threshold to decrease the number

of pods and save more resources.

Considering that there is 20% probability of random selections and real world influences, the historical changes during each round will not be the same even when the agent does not update anymore. Therefore, the similarity in patterns of rewards and thresholds between last three rounds demonstrates the convergence of this agent on changing workload.

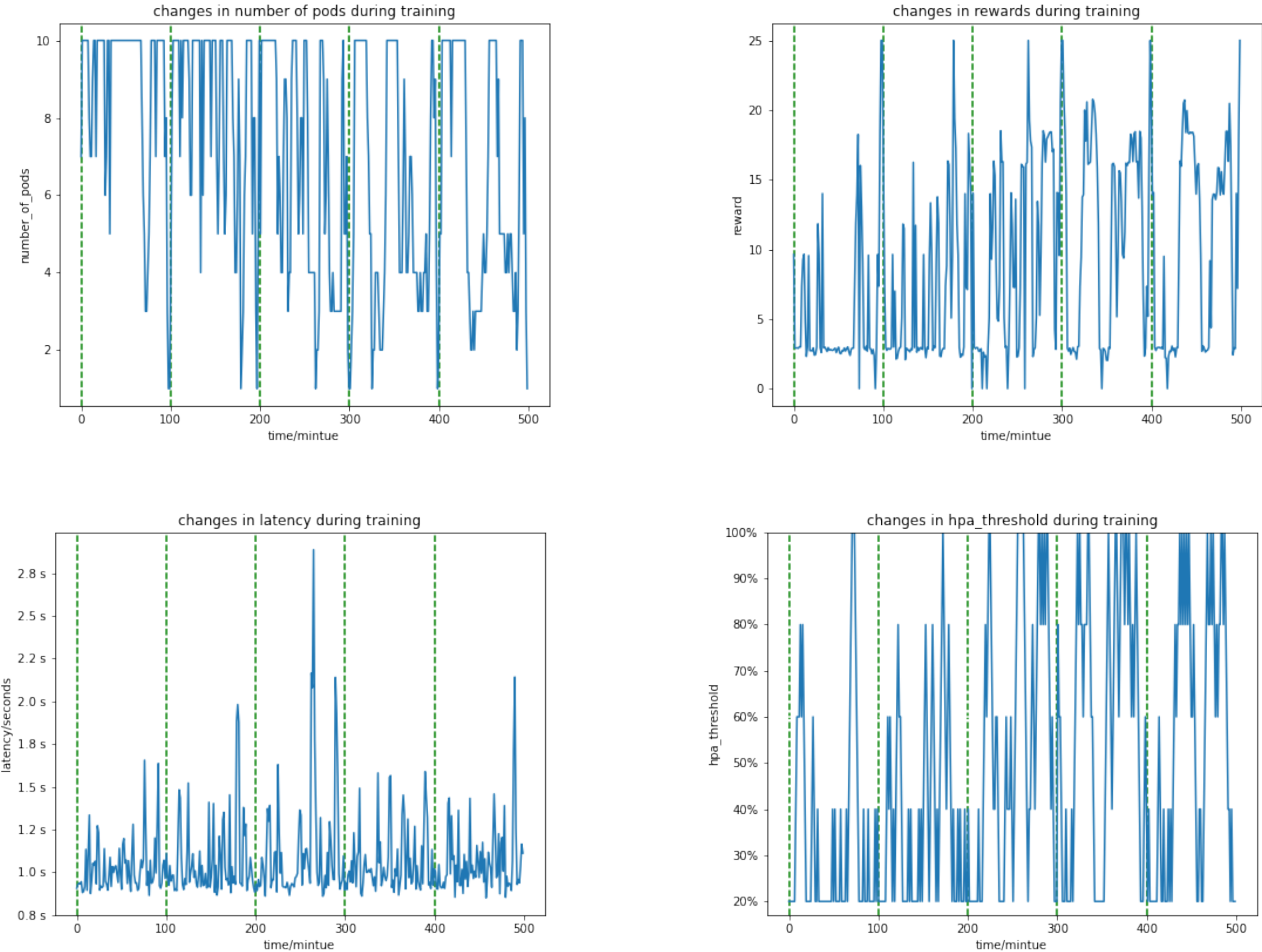

Figure 6.4: Changes during training period with changing workload

6.2.2 Repetitive Testings and Results

Experiments about testing the performance of the Q-learning agent are conducted after it is converged on training workloads. We applied both the auto-scaling agent and one reference threshold method respectively on the simulated testing workload defined in Section 6.1.1 to compare and contrast each other. In order to reduce random effects and improve the robustness of experiments, we conducted three repetitive rounds of testing. Every round of testing starts with the same Q-learning agent, (i.e. with the same Q-table), and the online learning will occur within one round.

In order to choose one reference threshold to compare with the Q-learning agent, we collected historical records in the threshold set by the agent. Information about the threshold is illustrated in Figure 6.5 with one color corresponding to one round. From the plot, we can see that decision patterns are similar to each round with the consideration of 20% probability of random selection.

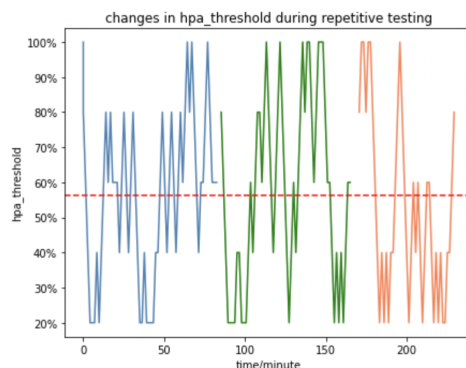

Figure 6.5: Thresholds of HPA during repetitive testing

Table 6.4: Descriptive statistics of thresholds during repetitive testing

mean	median	mode
56%	60%	60%

Descriptive statistics of thresholds are shown in Table 6.4. According to these statistics, we conduct stress test experiments with thresholds as 56% and 60%, and found that the HPA with threshold as 60% has more similar performance with the Q-learning agent. Therefore, we started with 60% threshold and make more attempts with thresholds close to 60%. Finally, after rounds of experiments, we found that 59% threshold is the most similar to the

Q-learning agent, both in resource cost and performance, thus we set 59% as the reference threshold to compare with the Q-learning agent.

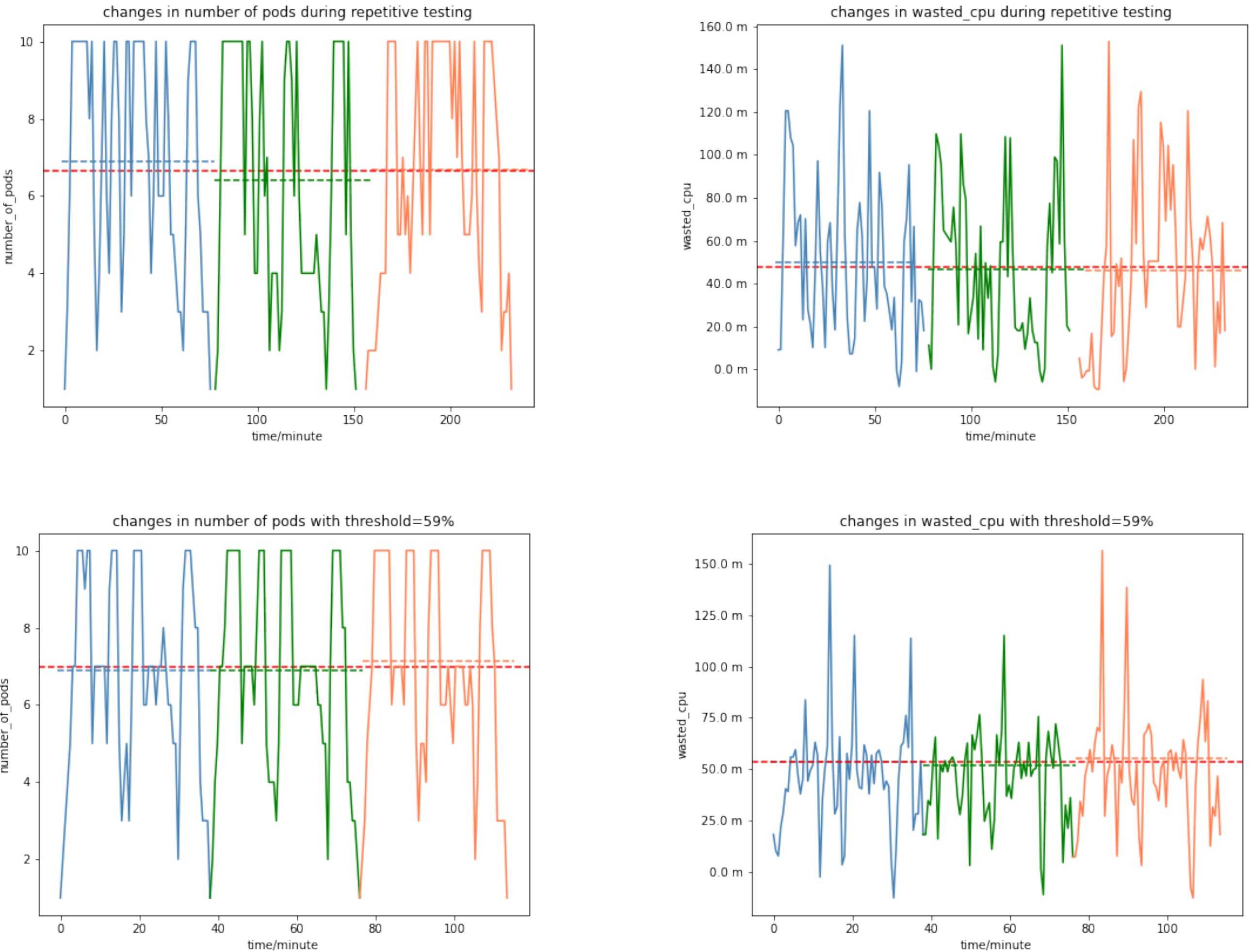

Figure 6.6: Changes of two mechanisms in resource utilization during repetitive testing

Figure 6.6 presents the comparison in resource cost between the Q-learning agent and threshold method, including the number of pods and wasted CPU, where the red horizontal line in each subfigure represents mean values over three rounds and other horizontal lines depict mean values within each round. From changing patterns of these two metrics, we can tell that the threshold-based method is more stable than the Q-learning agent. For example, patterns in the number of pods are more similar among three rounds in threshold-based scenario than in the case of the Q-learning agent.

Descriptive statistics about resource utilization are shown in Table 6.5 and Table 6.6. The Q-learning agent has a lower average number of pods and wasted CPU in each round compared to the threshold method, which demonstrates that the Q-learning agent is able to save more resources than traditional threshold-based methods. Besides, higher standard deviations in both the number of pods and wasted CPU is observed in the Q-learning agent among different rounds, which proves that threshold-based methods are more stable than Q-learning agents in resource cost as well.

Table 6.5: Descriptive statistics in number of pods during repetitive testing

Method	R1-Mean	R2-Mean	R3-Mean	All-Mean	Std
Q-learning	6.90	6.41	6.67	6.66	0.20
Threshold	6.68	6.73	6.80	6.74	0.05

Table 6.6: Descriptive statistics in wasted CPU during repetitive testing

Method	R1-Mean	R2-Mean	R3-Mean	All-Mean	Std
Q-learning	50.01	46.56	46.08	47.55	1.75
Threshold	45.80	45.05	48.13	46.33	0.02

In Figure 6.7, we depict historical results of two methods in performance of applications, i.e. latency, where three outliers can be observed in the Q-learning agent scenario. In spite of those outliers, these two methods have similar changing patterns, and most request time changes from 0.9 seconds to 1.7 seconds.

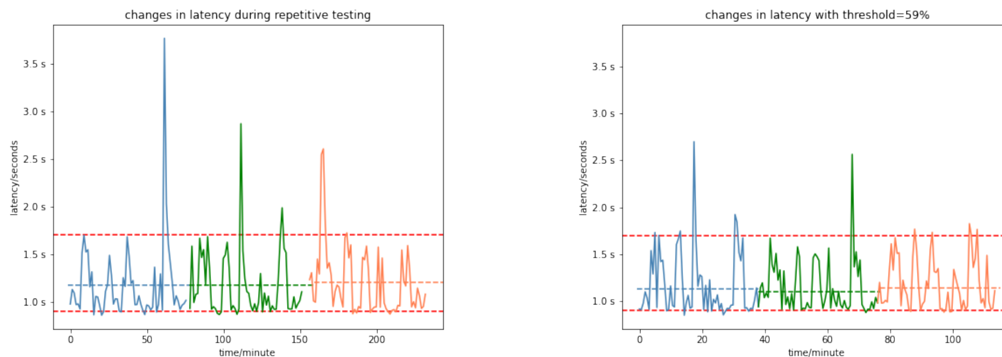

Figure 6.7: Changes of two mechanisms in latency during repetitive testing

Table 6.7: Descriptive statistics of latency during repetitive testing

Method	R1-Mean	R2-Mean	R3-Mean	All-Mean	Std
Q-learning	1.170	1.170	1.174	1.172	0.003
Threshold	1.173	1.147	1.190	1.170	0.018

Descriptive statistics about performance of these two methods are shown in Table 6.7. From the table, the threshold-based method and the Q-learning agent achieves similar average latency. However, the Q-learning agent has a lower standard deviation of average latency among different rounds and is more stale than the threshold method in performance of applications.

Chapter 7

Discussion

In this Chapter, we summarize comparative experimental results and present some reflections. Based on the results and discussion, we propose future directions to enhance the whole work.

7.1 Discussion of Experimental Results

We summarize comparisons between the Q-learning agent and the threshold-based method in resource utilization and performance of applications. Both descriptive statistics and historical patterns are discussed.

From average values in the number of pods and latency during repetitive testing, the Q-learning agent utilizes around 0.1 fewer pods compared to the threshold-based method, while having the same latency as the rule-based one. The lower number of assigned machines illustrates the effectiveness of the Q-learning auto-scaler in making the trade-off between resource cost and SLA. Since we spend a lot of time in designing the Q-learning algorithm according to the characteristics of the EPI framework and the training of RL agents is a time-consuming job, parameters of the Q-learning algorithm in our experiment scenario are reference values from literature works [24] and are not fine-tuned. There is a huge possibility that the performance of the Q-learning agent can be improved by adjusting those parameters.

However, according to historical records in the number of pods and wasted CPU during repetitive testing in Section 6.2.2, oscillations in resource utilization can be observed in both mechanisms, especially when the input load oscillates in a short time. These oscillations are caused by the combination of request rate oscillations, frequent scaling operations, and

the poorly configured auto-scaler. When the load is too unbalanced (has significant rate differences) and the time duration between scaling operations is short, states of the system will have significant differences, leading to oscillations in the system. Meanwhile, from comparisons between these two methods, we can see that the Q-learning algorithm is more sensitive to working load oscillations and is noisier compared to the threshold-based method.

7.2 Future Work

In Section 7.1, we discuss that both the Q-learning agent and the threshold-based method present oscillations in the number of assigned machines, and the Q-learning agent is more sensitive to very short bursts of traffic. Fluctuations lead to inefficient resource usage, therefore we want to reduce the possibility of oscillations in provisioning machines in our future work.

From literature works, there are three commonly adopted methods to reduce the possibility of causing oscillation for rule-based methods:

- Cooling Time: the scaling agent conservatively waits a fixed minimum amount of time between each scaling operation. The time is set by users and is known as the cooling time [25].
- Dynamic Parameters: scaling operations are triggered and controlled by predefined parameters, which are dynamically tuned during runtime. For instance, Lim et al [26] defined dynamically tuning triggering thresholds to control scale-down operations in their work.
- Anomaly Detection: rule-based auto-scalers are presented with models or rules to identify abnormal conditions that might cause oscillations and pose restrictions on such settings. The Anomaly-Aware Deep RL-based scaling method proposed by Moghaddam et al [27] integrates one data analyzer module into the system to decide if the system is abnormal compared to the previous observation of VM.

The above strategies can be applied in RL-based auto-scalers as well, meanwhile, the RL-based mechanism is able to reduce fluctuations by including costs of reconfiguring the infrastructure in the reward function. Ghobaei et al [28] consider the price of initializing one VM and the start-up time of VMs when defining VM costs in the total cost function.

Rossi et al [23] include the penalty of machine adaptation in the cost function, where the adaptation cost measures the application unavailability following a vertical scaling action.

As a future work, we plan to first conduct experiments to collect the costs of initializing or destroying one machine in our experimental infrastructure. Then, we will change the current action space from reconfiguring the threshold of HPA to directly setting the number of machines. With the new action space, the costs of adjusting the number of machines can be calculated in the reward function to reduce oscillations. Meanwhile, more simulations will be performed to better capture the behavior or pattern of requests from clients and generate a more realistic working load. Parameters of the Q-learning algorithm will be tuned correctly on the refined simulated working load to improve the performance.

Chapter 8

Conclusion

In this project, we propose and give answers to two research questions:

- Q1-How to implement and integrate Reinforcement Learning methods to recommend scaling within a defined scenario setup?
  Based on knowledge from literature works and the characteristics of the use case, the EPI framework, we define two types of metrics to quantify the status of the entire system. Resource utilization is represented by the number of assigned machines and average CPU utilization, and quality of service is measured by the response time of requests. The Q-learning agent allocates resources by setting the thresholds of triggering scaling actions in the rule-based auto-scaler. One reward function is defined to make the trade-off between the number of machines and the latency of applications.
- Q2-According to defined metrics, is it efficient to apply Reinforcement Learning methods in Kubernetes VFN container scaling, compared to using more conventional rule-based tools?
  We first find the optimal configuration of the system through a series of load testings, then train the RL agent on multiple fixed request rates and one changing workload. Subsequently, we design one changing workload with sudden bursts and drop-offs of request rates to test the RL agent and the threshold-based auto-scaler. These two approaches are compared in the number of deployed machines, wasted CPU, and response time with descriptive statistics and visualization of historical records.

The purpose of this thesis is to provide a detailed comparison between the RL-based agent and the rule-based method for automatic resource allocation in the EPI framework. One Q-learning agent is designed and implemented to scale resources through the Kubernetes

HPA and experiments are conducted to evaluate the RL agent. Experimental results show that the Q-learning based auto-scaler can be applied in the infrastructure with a slightly lower resource cost compared to the traditional threshold-based method. In the future, we plan to design one RL agent to directly adjust the number of machines and consider the cost of creating or destroying machines to avoid frequent reconfiguration of the system and reduce oscillations.

Bibliography

[1] Cncf wg-serverless whitepaper v1.0.

[2] Kubernetes horizontal pod autoscaler. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod- autoscale/. [Online].

[3] AWS Auto Scaling. Available: https://aws.amazon.com/cn/autoscaling/. [Online].

[4] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*, pages 50–56, 2016.

[5] Enabling personalized intervention. `https://enablingpersonalizedinterventions.nl/index.html`.

[6] Priscilla Benedetti, Mauro Femminella, Gianluca Reali, and Kris Steenhaut. Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 674–679. IEEE, 2022.

[7] Kubernetes. `https://kubernetes.io`.

[8] Docker. `https://www.docker.com`.

[9] Openai gym. `https://www.gymlibrary.ml`.

[10] Numpy. `https://numpy.org`.

[11] Andrei Palade, Aqeel Kazmi, and Siobhán Clarke. An evaluation of open source serverless computing frameworks support at the edge. In *2019 IEEE World Congress on Services (SERVICES)*, volume 2642, pages 206–211. IEEE, 2019.

[12] Junfeng Li, Sameer G Kulkarni, KK Ramakrishnan, and Dan Li. Understanding open source serverless platforms: Design considerations and performance. In *Proceedings of the 5th international workshop on serverless computing*, pages 37–42, 2019.

[13] Zhiyu Zhang, Tao Wang, An Li, and Wenbo Zhang. Adaptive auto-scaling of delay-sensitive serverless services with reinforcement learning. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 866–871. IEEE, 2022.

[14] Haoran Qiu, Weichao Mao, Archit Patke, Chen Wang, Hubertus Franke, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. Reinforcement learning for resource management in multi-tenant serverless platforms. In *Proceedings of the 2nd European Workshop on Machine Learning and Systems*, pages 20–28, 2022.

[15] Fabiana Rossi, Valeria Cardellini, Francesco Lo PRESTI, and Matteo Nardelli. Dynamic multi-metric thresholds for scaling applications using reinforcement learning. *IEEE Transactions on Cloud Computing*, 2022.

[16] Abeer Abdel Khaleq and Ilkyeun Ra. Intelligent autoscaling of microservices in the cloud for real-time applications. *IEEE Access*, 9:35464–35476, 2021.

[17] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[18] Ankur Joshi, Saket Kale, Satish Chandel, and D Kumar Pal. Likert scale: Explored and explained. *British journal of applied science & technology*, 7(4):396, 2015.

[19] Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang. Best of both latency and throughput. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.*, pages 236–243. IEEE, 2004.

[20] Ricardo Grunitzki. A flexible approach for optimal rewards in multi-agent reinforcement learning problems. 2018.

[21] JV Bibal Benifa and D Dejey. Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment. *Mobile Networks and Applications*, 24(4):1348–1363, 2019.

[22] Enda Barrett, Enda Howley, and Jim Duggan. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and computation: practice and experience*, 25(12):1656–1674, 2013.

[23] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 329–338. IEEE, 2019.

[24] Shubo Zhang, Tianyang Wu, Maolin Pan, Chaomeng Zhang, and Yang Yu. A-sarsa: A predictive container auto-scaling algorithm based on reinforcement learning. In *2020 IEEE International Conference on Web Services (ICWS)*, pages 489–497. IEEE, 2020.

[25] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, 51(4):1–33, 2018.

[26] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. Automated control for elastic storage. In *Proceedings of the 7th international conference on Autonomic computing*, pages 1–10, 2010.

[27] Sara Kardani-Moghaddam, Rajkumar Buyya, and Kotagiri Ramamohanarao. Adrl: A hybrid anomaly-aware deep reinforcement learning-based resource scaling in clouds. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):514–526, 2020.

[28] Mostafa Ghobaei-Arani, Sam Jabbehdari, and Mohammad Ali Pourmina. An autonomic resource provisioning approach for service-based cloud applications: A hybrid approach. *Future Generation Computer Systems*, 78:191–210, 2018.