Deep Learning and Feedback Control based Container Auto-scaling for Cloud Native Micro-services

Zhicheng Cai, *IEEE Member* and Hang Wu, and Xu Jiang, *IEEE Student Member* and Xiaoping Li, *IEEE Senior Member* and Rajkumar Buyya, *IEEE Fellow*

Abstract—In kubernetes-based Cloud Native platforms, allocating containers to micro-services elastically according to workload changes is benefical to minimizing resource cost while stabling response times. However, inaccurate performance models for multi-container systems, along with coarse-grained container-based allocation, cause performance fluctuations. In this paper, deep learning, traditional Jackson Queuing Network (JQN) and feedback control are integrated to devise a container provisioning algorithm which leverages the neural networks' ability to fit nonlinear performance models, the real-time responsiveness of feedback control, and the precise prediction of micro-service interactions offered by the JQN. The proposal is evaluated on a real Kubernetes based Cloud Native cluster. Experimental results illustrate that the container cost is decreased by 10.94%∼11.36% while satifisfying Service Level Agreements (SLA) in terms of 95^{th} accessing-path response times.

Index Terms—Deep neural network, Micro-service, Container provisioning, Cloud computing, Cloud native

1 Introduction

More and more applications are designed or revised to use the Cloud Native architecture, because of light-weight deployment, easy auto-scaling and recovery of containers. In Cloud Native, applications are composed of tens or even thousands of micro-services, and each micro-service has multiple back-end container replicas to support handling of high concurrent requests [1]. Auto-scaling containers allocated to each micro-service appropriately is crucial for guaranteeing Quality of Service (QoS) in terms of response times. Performance models describing the relationship among response times, request arrival rates and given container resources are the basis of container auto-scaling algorithms. However, there is no exact performance models which could be used to describe multi-container systems accurately. Meanwhile, request transferring among different micro-services has a great impact on container auto-scaling too. JackSon Queuing Network (JQN) is an effective method to describe the impact among micro-services, but JQN is only designed based on traditional queuing models. Model-free Deep Reinforcement Learning (DRL) is another kind of container auto-scaling method which has self-studying ability, but needs long period of time to collect training samples before it could be put into practice. The main goal of this paper is to design a container auto-scaling algorithm for Cloud Native applications with multiple micro-services which is accurate, but needs very short training times. The main challenges include real-time changes of non-linear performance models, impact among multiple services, and coarse granularity of container allocating.

The relationship among response time, arrival rate and allocated number of containers for a micro-service is non-linear and changes in real time which makes the container auto-scaling complex. Linear, inverse-proportional and queuing models have been widely used to describe such relationship. However, there are great deviations between these time invariant mathematical models and the real-time changed systems. Although feedback control is an effective way to make forward performance models suitable for real time changes of physical systems, such deviations make feedback control take much effort to become stable whenever the workload changes.

Complex invoking relationships among micro-services make the change of allocated containers to one service have impacts on other services. Modeling such mesh structure among micro-services is helpful to auto-scaling more appropriately. JQN serves as an effective tool for predicting cascading effects among multiple micro-services. However, JQN is designed to cooperate with queuing models in existing works and queuing models are not able to describe multi-container systems accurately.

The coarse-grained horizontal resource provisioning approach results in system instability. Horizontal auto-scaling is more widely used by Cloud Native Application developers because existing containers are not influenced by newly added containers providing uninterrupted service. However, container based adjustment is not in fine granularity. When feedback control is utilized to ensure that the system explicitly follows a specified reference response time, the allocation or removal of a single additional container can

- *Zhicheng Cai and Hang Wu are with the School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China. (caizhicheng@njust.edu.cn)*
- *Xu Jiang is with the School of Cyber Science and Engineering, Nanjing University of Science and Technology, Nanjing, China.*
- *Xiaoping Li is with the School of Computer Science and Engineering, Guangdong University of Technology, Guangzhou, China.*
- *Rajkumar Buyya is with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, the University of Melbourne, Australia.*

sometimes cause the response time to fluctuate significantly around the reference time, rather than converging precisely onto it.

In this paper, a Deep Neural Network and collaborative Feedback Control based container provisioning method (DNN-FC) is developed for container auto-scaling which is a hybrid method of Deep Neural Network (DNN), JQN and feedback control. The main contributions include

1) DNN-based performance model is built to describe the multi-container systems more accurately. DNN-based feedback control is designed to cope with real-time changes of system performance.
2) JQN is integrated with DNN-based feedback control to collaboratively adjust containers across multiple services, taking into account estimated impacts among them. This integration is capable of shortening the adjustment cycle.
3) A strategy for suppressing fluctuations based on state tracing, along with a customized cooling-down method specifically designed for feedback control, have been developed to stabilize the control process.

The rest of the paper is organized as follows. Section 2 is the related work followed by the problem description in Section 3. Section 4 describes the proposed method and Section 5 evaluates the proposal in a real Kubernetes cluster.

2 Related Work

Container scheduling of Cloud Native includes auto-scaling and placement [2]. While container placement primarily involves selecting suitable nodes for containers considering resource efficiency [3], energy consumption [4] or network latency [5], [6], auto-scaling addresses the determination of the number of containers required for each service, which is the focus of this paper and is surveyed as follows. In certain existing work, such as the container scheduling method for serverless functions [3], the total quantity of required resources is able to be determined directly by considering the concurrency of requests and the resource consumption per request, as outlined in the service specification. However, for common micro-services, multiple requests are usually processed by a fixed set of shared threads simultaneously in each container, requests which can not be processed immediately are queued in containers leading to unignorable waiting times and response times vary under different queuing lengths [7]. Such nonlinear queuing systems complicate the process of container auto-scaling in terms of response time control.

2.1 Exact model based auto-scaling methods

For each micro-service in Cloud Native, the average response time is determined by the number of allocated containers, arrival rates of requests and the configuration of each container. Linear models are the simplest way to describe the performance of such queuing systems including linear time-invariant [8], adaptive parameter [9], or multi-model switching [10] models. Besides linear models, an inverse-proportional performance model is proposed by Baresi et al. [11]. Queuing models are more accurate than the linear and inverse-proportional models [2]. Given the arrival rate of requests and provisioned processors or containers, the average response time can be obtained based on queuing models. For instance, Salah et al. [12] developed a Markov queuing model to depict the probabilities of state transitions for linear two-tier services. However, there are still unignorable deviation between queuing models and real systems. Although feedback control is able to amend the inaccuracy of queuing models [13], [14], the deviation of the applied performance model and the real system increases the number of consumed steps of feedback control to follow the reference point.

For auto-scaling resources of applications with multiple services, complex invoking relationships among services should be considered [15]. Auto-scaling resources of one service usually lead to non-neglectable impact on other services [16]. For example, when the bottleneck of a service is eliminated by providing more resources to the service, the additionally passed requests of the service might generate more requests to subsequent services leading to new bottlenecks. JQN is an effective method to modeling the relationship of request transfer among meshed services which connect multiple diverse queuing models as a network [17], [18], [19]. However, JQN is still based on inaccurate queuing models.

2.2 Deep learning based auto-scaling methods

DNN has been used to fit the performance function of single services more accurately. For example, Rao et al. [16] and Yazdanov et al. [25] applied Deep Q-Network (DQN) to scale resource of VMs vertically. For single services with multiple parallel backend VMs or containers, DQN-based methods are proposed by Zong et al. [20] and Fang et al. [21] to allocate VMs horizontally. However, when the number of containers required by one service increases greatly in Kubernetes, the action space increases exponentially making the DQN hard to convergence. In order to relief the impact of large action space, the action space of DQN is limited to be increment, decrement of a fixed amount of resources or no operation [21], [22]. For applications with multiple services, the state and action spaces are more larger than single service applications making the deep reinforcement learning hard to convergence. Therefore, DQN is only applied to determine wheather to add or release resources from the whole system perspective without specifying which services to adjust, and the detailed service to be adjusted will be determined by another heuristic method [24]. However, the heuristic method is hard to cope with the complex service selecting and resource scheduling task.

2.3 Comparison to existing studies

For exact methods, although feedback control has made up for the inaccuracy of queuing models, finding more accurate performance model itself for multi-container system is still a promising method to decrease the drawback time. Additionally, achieving convergence in DRL for multi-service applications that possess extremely large state and action spaces necessitates a considerable amount of time. Therefore, in this paper, DNN has been first used to model the relationship among response time, arrival rate and the

TABLE 1: A comparison of existing resource provisioning methods for Cloud and Edge systems

Works	Application types	Platforms	Methods
[20], [21]	Single multi-instance services	CloudSim, simulation	DQN based horizontal or vertical auto-scaling
[22]	Two-tier multi-instance services	Amazon EC2	Accessing pattern extracting, DQN, limited action space
[12], [17]	Linear multi-instance services	Virtual machine based data center, discrete-event simulation	Queuing network, Embedded Markov chain, Scaling up all tiers proportionally
[23]	Meshed multi-instance services	Physical machines of PlanetLab	Ilog Cplex 9.0, ignoring queuing times of concurrent users
[15]	Meshed multi-instance services	Physical machine data center	Multi-tier negotiating
[24]	Meshed multi-instance services	OpenStack based Edge and Cloud data centers	DQN for high-level strategy and heuristic based action selection
[18], [19]	Meshed multi-instance services	CloudSim or Kubernetes based Cloud data center	Queuing network
[14]	Meshed multi-instance services	CloudSim based Cloud data center	Queuing network, feedback control
Our approach	Meshed multi-instance services	Kubernetes based Cloud data centers	Deep neural network, Queuing network and control theory

number of containers forming a more accurate and light-weight performance model. Then, the light-weight DNN-based model is used to replace queuing models of JQN. Finally, DNN-based feedback control is applied to follow the required performance closely.

3 Problem Description

The service architecture is widely used in Cloud Native applications which are usually composed of multiple micro-services with meshed interactions among them [1]. For example, Figure 1 shows a bookinfo application [26] which displays information about books and contains *product page*, *details*, *reviews* and *rating* micro-services. Each micro-service contains multiple parallel backend containers. Requests to the same micro-service are routed to different backend containers using diverse load-balancing algorithms. Kubernetes is an open source system which is usually used to organize these containerized micro-services. Micro-services usually need to invoke other micro-services synchronously or asynchronously. Different business types have diverse accessing paths. For example, in Figure 1, p_1=(S_1), p_2=$(S_1 \rightarrow S_2)$ and p_3=$(S_1 \rightarrow S_3 \rightarrow S_4)$ are three accessing paths of different business types. Each business type usually has a path-SLA which specifies that the percentage V_{p_l} of the 95th response time (or mean response time) of the corresponding accessing path p_l larger than a upper-limit T_{p_l} should be smaller than a threshold θ_{thr} (e.g., 5%). The main goal of this paper is to design a horizontal container auto-scaling algorithm to minimize the cost of deployed containers while satisfying path-SLAs. For simplification, the cost of each container (pod) is set to be unit 1 per control interval. Let $N_i(k)$ be the number of allocated pods to service S_i at control step k and L be the count of business paths. The formal description of the considered problem is

$$\min C = \sum_{S_i \in S} \sum_{k=1}^{K} N_i(k) \tag{1}$$

$$\text{st. } V_{p_l} < \theta_{thr}, l = 1, 2, ..., L \tag{2}$$

Equation (1) minimize the total cost of allocated pods for all micro-services where K is the number of considered control steps and $S = \{S_i | i = 1, ..., n\}$ is the set of all micro-services. Equation (2) is the constraint of fulfilling path-SLAs. In order to fulfill the path-SLAs, every micro-service is usually needed to be allocated a separate service-SLA (e.g., average processing time of the micro-service S_i is smaller than a reference time W_i^r). Service-SLAs should be set appropriately for each micro-service so that the SLA of each path will be fulfilled when service-SLAs of all micro-services are fulfilled. In this paper, it is assumed that service-SLAs of all micro-services have been given in advance based on characteristics of each microservice. Then, the main objective of the proposed algorithm is to allocate appropriate number of Pods to each micro-service to fulfill its service-SLA. Common notations are shown in Table 2.

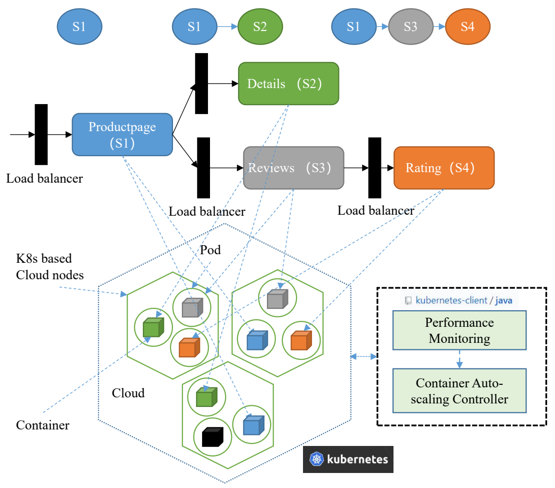

Figure 1: Architecture of container auto-scaling in Kubernetes

TABLE 2: Common notations

Notation	Description
k	the index of control step
S_i	micro-service i
p_j	the $j-$th accessing path of services
N_i	the number of allocated containers to S_i
W_i^r	the reference time of S_i (service-SLA)
λ_i	request arrival rate of S_i
$\omega_i(k)$	adjustment factor of λ_i for step k of S_i
$\varphi_i(k)$	cumulative arrival rate adjustment factor for S_i
q_i	the queuing length of requests in S_i
$y_i(k)$	the real time processing time of S_i at step k
$\hat{\mu_i}$	the average request processing rate per container of S_i

4 Proposed Provisioning Method

In this paper, as shown in Figure 2, a JQN-based scheduling framework is first proposed in which not only queuing

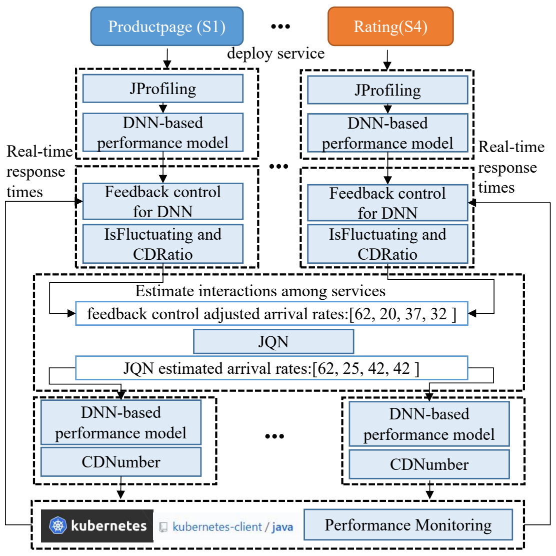
Productpage (S1)
Rating(S4)
deploy service
JProfiling
DNN-based performance model
Feedback control for DNN
IsFluctuating and CDRatio
Real-time response times
Estimate interactions among services
feedback control adjusted arrival rates:[62, 20, 37, 32]
JQN
JQN estimated arrival rates:[62, 25, 42, 42]
DNN-based performance model
CDNumber
kubernetes
kubernetes-client / java
Performance Monitoring

Figure 2: Architecture of the proposed algorithm

models, but also deep-learning based models can be used to describe the system performance. Then, DNN-based performance model is built for the scheduling framework based on off-line sampling (JProfiling) to describe the complex non-linear relationships among response times, resources and arrival rate of each micro-service accurately. Feedback control is utilized to adjust the parameters (arrival rate fixing factor) of the DNN-based performance model, ensuring its suitability for real-time changes. Initially, the original arrival rate of each service is updated using the arrival rate fixing factor. This factor is dynamically adjusted by DNN-based feedback control, which fine-tunes it based on real-time performance metrics. Subsequently, the updated arrival rates of all tiers are employed to derive the final arrival rates, utilizing the JQN method to account for request transfers between services. Finally, the number of allocated containers for each service is determined in accordance with the relationships outlined by the DNN-based performance model. To mitigate the fluctuation arising from coarse-grained container-based provisioning and variations in workload, a novel cooling down approach is introduced, which integrates container number cooling down (CDNumber) and feedback control ratio cooling down (CDRatio), along with a state-tracing based fluctuation detection strategy (Isfluctuating).

4.1 Queuing network based scheduling framework

When a micro-service does not have sufficient containers, many requests are blocked in its waiting queue. When sufficient number of containers are allocated to such kind of micro-services, there are additional passed requests from these micro-services to other connected micro-services leading to workload surge. In JQN, the additional passed-through request rate $\Delta\lambda_i$ can be estimated using queuing theory as follows [19].

$$\Delta\lambda_i = \begin{cases} \lambda_i - \hat{\mu}_i \times N_i + q_i & \hat{\mu}_i \times N_i < \lambda_i \\ q_i & otherwise \end{cases} \tag{3}$$

where q_i, λ_i, N_i and $\hat{\mu}_i$ are the queuing length, arrival rate, container count and request processing rate per container for S_i, respectively. The final arrival rate of each service is

$$\lambda_i^u = \lambda_i + q_i + \sum_{j=1}^{n} \Delta\lambda_j \times \psi_{ji}, i \in \{1, 2, \ldots, n\} \tag{4}$$

where ψ_{ji} is the request transition probability matrix. Adjusting containers based on estimation of such interactions is beneficial to decreasing the length of the resource adjusting period.

After estimating request arrival rates of services based on JQN, performance models are required to determine the required resource of each service. In existing JQN-based resource scheduling algorithms, the M/M/N or its variants are widely used, but they could be replaced by other more accurate models. In this paper, a JQN-based resource scheduling framework is proposed which does not specify applied performance models. A performance model $y_i = \phi(N_i, \lambda_i, \hat{\mu}_i)$ of S_i with multiple containers is used to describe the relationship among λ_i, N_i, $\hat{\mu}_i$ and y_i. Given λ_i, N_i and $\hat{\mu}_i$ of S_i, the response time y_i could be calculated by the performance model. On the contrary, given λ_i, $\hat{\mu}_i$ and a reference time W_i^r, the number of required containers for the micro-service could be obtained by

$$N_i = \min_{N_i \in Z^+} \{N_i | \phi(N_i, \lambda_i, \hat{\mu}_i) \leq W_i^r\} \tag{5}$$

If the performance model is not accurate, allocating N_i containers to S_i, the real response time y_i might deviate from W_i^r. Feedback control could be used to amend N_i to minimize the deviation through adjusting multiplication factors of λ_i [14] or W_i^r [13]. Different kinds of provisioning algorithms could be obtained by providing different performance models. In this paper, a DNN-based performance model is proposed in Section 4.3.

4.2 Service time extracting

During synchronize calls, the caller services do nothing but wait the return of remote calls. Therefore, the response time of a caller service includes the turnaround time of remote calls. Longer response times in caller services may not necessarily stem from a lack of resources within the caller service itself, but rather from insufficient resources in the called services. The existence of synchronize calls makes the estimation of real computation capacity requirement of each service complex. Therefore, the actual processing time of a service (called service time) is determined by excluding the turnaround time associated with invoking subsequent services. Taking the accessing path ($S_1 \to S_3 \to S_4$) as an example, when response times of S_1, S_3 and S_4 are 0.8, 0.5 and 0.2, the service time of them are 0.3 (0.8-0.5), 0.3 (0.5-0.2) and 0.2, respectively. In Traefik and Istio based systems, accessing trace of requests can be obtained and used to calculate service times.

4.3 Deep neural network based performance model

In this paper, a lightweight four-layer fully-connected DNN has been implemented for modeling multi-container systems, offering exceptional computational efficiency and rapid initialization capabilities. The input layer contains the arrival rate λ_i and the number of containers N_i. There are

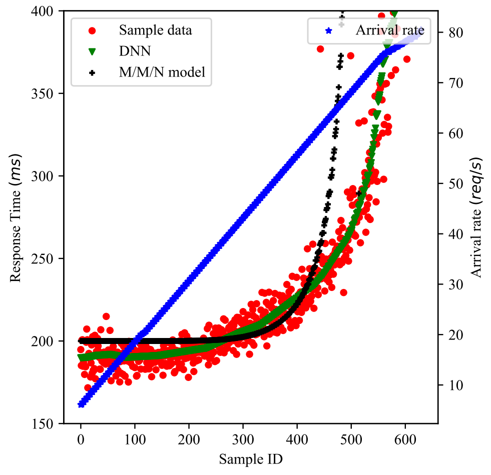

Figure 3: Response times of a service under different arrival rates

two hidden layers and an output layer which only contains the response time $y_i(k)$, i.e.,

$$y_i(k) = DNN_\theta(N_i, \lambda_i), \tag{6}$$

where θ is the parameters of the DNN. The activation function is Relu and the optimization algorithm is Adam. The DNN is trained offline based on JMeter profiling data. Figure 3 shows the estimated response times of different performance models for different arrival rates given a fixed number of containers. The blue five-pointed stars are arrival rates which increase as the sample id increases. As the arrival rate increases, the response times increase and the deviations between the M/M/N and the sample data increases too. As a whole, DNN is more accurate than the M/M/N. Given the trained DNN, the number of containers could be obtained for a given λ_i and W_r as follows

$$N_i = \min_{N_i \in Z^+} \{N_i | DNN(N_i, \lambda_i) \leq W_i^r\} \tag{7}$$

In order to automate the profiling process, a JMeter based profiling method (**JProfiling**) is developed which integrates seamlessly with Kubernetes. Upon each new service is deployed to Kubernetes, JProfiling automatically initiates data acquisition processes and constructs a dedicated DNN model. JProfiling systematically evaluates latency characteristics under different combinations of request arrival rates and allocated containers.

4.4 DNN-based feedback control

According to equation (7), y_i should be close to W_i^r given N_i containers under a real arrival rate λ_i, i.e.,

$$\lambda_i \approx DNN^{-1}(N_i, W_i^r) \tag{8}$$

However, in practice, $y_i(k)$ may differ from W_i^r resulting in an output error $e(k) = W_i^r - y_i(k)$. The theoretical arrival rate used to generate $y_i(k)$ based on DNN is

$$\lambda_i^{th} = DNN^{-1}(N_i, y_i(k)) \tag{9}$$

In other words, λ_i^{th} is the arrival rate experienced by the DNN. Let $\varphi(k) = \lambda_i^{th}/\lambda_i$ be a fixing coefficient, the DNN model could be fixed to be

$$y_i(k) = DNN(N_i, \lambda \times \varphi_i(k)), \tag{10}$$

which is more consistent with the real performance model. It is assumed that this model is accurate for different N_i. Therefore, allocating

$$N_i = \min_{N_i \in Z^+} \{N_i | DNN(N_i, \lambda_i \times \varphi_i(k)) \leq W_i^r\} \tag{11}$$

containters will lead to a response time more close to W_i^r. In other words, fixing the DNN by adding a coefficient $\varphi_i(k)$ to the arrival rate is able to adjust the real response time from $y_i(k)$ to W_r.

Feedback control is a widely used technique which allows to minimize the output error $e(k)$ step by step while observing real-time effect. As shown in Figure 4, the DNN-based performance model could be abstract to be a linear model

$$y_i(k+1) = y_i(k) + u_i(k) \tag{12}$$

by considering DNN and the inverse of it as a whole, like [27] to simplify the design of feedback controller where $u(k)$ is the control input. If a proportional control (PC) is applied, $u(k) = K_p e(k)$ where K_p is the control gain of proportional gain. PC tries to adjust $y_i(k)$ to $y_i(k+1)$ with a step size $u(k)$ each time. If we want to make the response time change from $y_i(k)$ to $y_i(k+1)$ rather than W_r, the DNN should be fixed according to Equation (10) by setting

$$\varphi_i(k) = \lambda_i^{th}/\lambda_i^{ref} \tag{13}$$

where $\lambda_i^{ref} = DNN^{-1}(N_i, y_i(k+1))$, and used to obtain a new N_i. The goal of PC is to minimize $e_i(k)$ through fixing the DNN model based on $u_i(k)$ and obtain new N_i using fixed DNN model $y_i(k) = DNN(N_i, \lambda \times \varphi_i(k))$. After resource is adjusted according to N_i, the new response time $y_i(k)$ is updated. Based on the fixed DNN model, the theoretical arrival rate for $y_i(k)$ is

$$\lambda_i^{th} = DNN^{-1}(N_i, y_i(k))/\varphi_i(k) \tag{14}$$

If there is still an output error, a new $y_i(k+1)$ will be selected by the PC. Let $\lambda_i^{ref} = DNN^{-1}(N_i, y_i(k+1))/\varphi_i(k)$, the coefficient for fixing the DNN model of Equation (10) again is

$$\varphi_i(k)' = \frac{\lambda_i^{th} \times \varphi_i(k)}{\lambda_i^{ref} \times \varphi_i(k)} = \frac{DNN^{-1}(N_i, y_i(k))}{DNN^{-1}(N_i, y_i(k+1))} \tag{15}$$

The cumulative fixing coefficient for S_i is

$$\varphi_i(k+1) = \varphi_i(k) \times \omega_i(k) \tag{16}$$

where $\omega_i(k) = \varphi_i(k)'$ is the adjustment factor of every step. The above process iterates to monitor and minimize the output error continuously. To avoid excessive adjustments in one step, $\omega_i(k)$ is limited within the interval $[\omega^{lower}, \omega^{upper}]$. Similarly, $e_i(k)$ is trimmed as

$$e_i(k) = \begin{cases} lower\_thr - y_i(k) & y_i(k) < lower\_thr \\ upper\_thr - y_i(k) & y_i(k) > upper\_thr \\ 0 & Otherwise \end{cases} \tag{17}$$

where $upper\_thr$ and $lower\_thr$ are two thresholds which define a tolerable interval $[lower\_thr, upper\_thr]$ for the real response time. By applying such threshold based error computing strategy, resources are not adjusted when the real response time is within the interval avoiding too

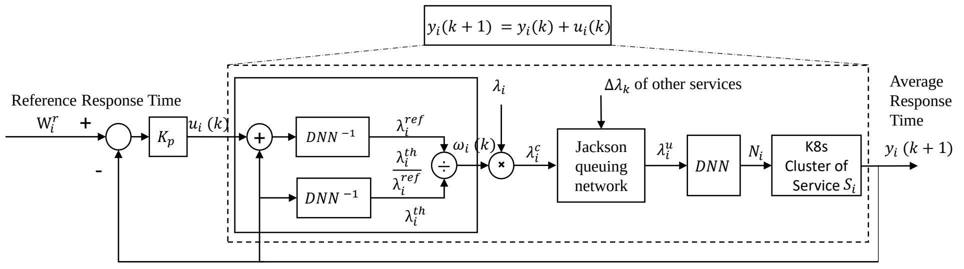

Figure 4: DNN-based feedback control

frequent adjustment. $\varphi(k)$ is limited within an interval $[\varphi^{lower}, \varphi^{upper}]$ which will be used to obtained a fixed arrival rate for each service by $\lambda_i^c = \lambda_i \times \varphi_i(k)$. After more accurate arrival rate λ_i^c of every S_i is obtained by the DNN-based feedback control as mentioned above, JQN will be used to calculate the final arrival rate λ_i^u of each service (Equations (3) and (4)) considering interactions among services. Finally, N_i is determined based on λ_i^u using the proposed DNN-based performance model according to Equation (7).

4.5 Cooling down for feedback control

When the arrival rate increases, additional containers are allocated to stabilize response times. When the arrival rate decreases, a part of containers are required to be released for saving cost. When the arrival rate increases again, new containers are required once more. However, it consumes several seconds for new containers to be ready leading to temporary high response times. Meanwhile, frequent adjusting of the number of allocated containers consumes additional overheads decreasing system performance further. The cooling down (CD) is a kind of strategy which is widely used to avoid resource fluctuations. However, in existing works, CD is usually only used at the last step to stabilize the container numbers directly (called **CDNumber**) without considering collaboration with feedback control. For instance, Algorithm 1 illuminates the cooling-down mechanism in HPA [28], specifying that the present number of allocated containers is capped at the maximum quantity allocated during the preceding cooling-down period.

Algorithm 1 Container number cooling down (CDNumber)

```
Input: $N_i$ container number of service $S_i$, $\nu^d$ length of the cooling down period.
Output: container number of service $S_i$ after cooling down
1: Append $N_i$ to the end of a queue $Q_i$;
2: if $|Q_i| > \nu^d$ then
3:     Remove the first element of $Q_i$;
4: end if
5: $N_i^{'} \leftarrow \max Q_i$;
6: return $N_i^{'}$
```

For the proposed feedback control method in Section 4.4, cooling-down strategy is also required to stabilize the control ratio $\omega(k)$. Unrestricted adjustments to $\omega(k)$ may lead to an undesirable proliferation of unnecessary control actions. When the CDNumber is applied, a control action of decreasing $\omega(k)$ will not take effect only after a cooling-down period ν^d. Therefore, during the cooling-down period, $\omega(k)$ should not be decreased again, i.e., $\omega(k)$ cannot be decreased again before the action actually takes effect to avoid excessive control. Meanwhile, $\omega(k)$ takes the maximum value of past L_q steps to avoid occasional changes. Since decreasing actions may only take effect after L_q steps, ν^d should be increased by L_q to compensate for the inherent latency. On the contrary, increasing $\omega(k)$ whenever necessary is helpful for dealing with workload surges, but it might consume several control steps before the control action of increasing containers takes effect. Therefore, $\omega(k)$ cannot be increased again only after an increase cooling-down period ν^u. The formal description of the control-ratio cooling-down strategy is shown in Algorithm 2.

Algorithm 2 Control-ratio cooling down (CDRatio)

```
Input: $\omega_i(k)$ control ratio, $k$ index of control steps, $K^u$ the index of last control step which allowed to increase $\varphi_i(k)$, $K^d$ the index of last control step which allowed to decrease $\varphi_i(k)$.
Output: $\omega_i(k)$ adjusted control ratio
1: if $\omega_i(k) > 1$ then
2:     if $k - K^u > \nu^u$ then
3:         $K^u \leftarrow k$ and return $\omega_i(k)$ ;
4:     end if
5: else
6:     Append $\omega_i(k)$ to the end of the queue $Q_i^\omega$;
7:     if $|Q_i^\omega| > L_q$ then
8:         Remove the first element of $Q_i^\omega$;
9:     end if
10:    $\omega_i(k) \leftarrow \max Q_i^\omega$;
11:    if $\omega_i(k) < 1$ and $k - K^d > \nu^d + L_q$ then
12:        $K^d \leftarrow k$ and return $\omega_i(k)$ ;
13:    end if
14: end if
15: return 1 ;
```

4.6 State tracing based fluctuation suppression

The main objective of control method is to make the real response time $y_i(k)$ within the interval $[lower\_thr, upper\_thr]$. However, containers are provisioned in a coarse-grained granularity. For an arrival rate, allocating a certain number of containers results in $y_i(k)$ falling below $lower\_thr$, however, releasing even a single container cause $y_i(k)$ to surge beyond $upper\_thr$. If the control method keeps the original objective unchanged, the system will fluctuate greatly.

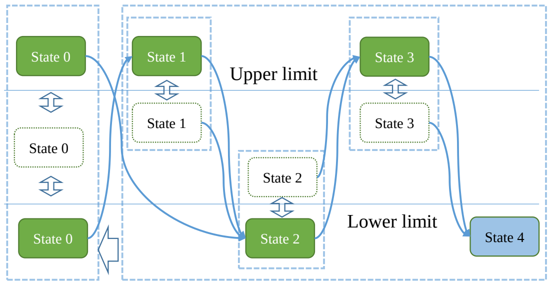

Figure 5: State changes of one micro-service

In this section, a state-monitoring based fluctuation suppression method **IsFluctuating** is proposed. The core concept is that when $y_i(k)$ experiences fluctuations around the range of $[lower\_thr, upper\_thr]$ due to the repeated allocation of an additional container or the release of one container, further alterations in the number of allocated containers are not permitted in order to prevent continued fluctuations. As shown in Figure 5, the micro-service starts in state 0 and transitions to 1 when $y_i(k)$ crosses from below $lower\_thr$ to above $upper\_thr$. From state 0 or 1, it moves to state 2 if $y_i(k)$ drops from above $upper\_thr$ to below $lower\_thr$. At state 2, it advances to state 3 when $y_i(k)$ rises from below $lower\_thr$ to above $upper\_thr$. From state 3, it proceeds to state 4 if $y_i(k)$ falls from above $upper\_thr$ to below $lower\_thr$. The state remains unchanged if, while in state 1 and 3, $y_i(k)$ shifts from above $upper\_thr$ into the $[lower\_thr, upper\_thr]$ interval, or while in state 2, $y_i(k)$ moves from below $lower\_thr$ into the same interval. In any state 1-4, the micro-service reverts to state 0 if the current arrival rate significantly deviates from the original arrival rate (e.g., $2 \times \hat{\mu_i}$) recorded upon entering that state. Upon reaching state 4, $\omega(k)$ becomes immutable and IsFluctuating returns true, otherwise, IsFluctuating returns false.

4.7 Formal description of DNN-FC

The formal description of the auto-scaling algorithm is show in Algorithm 3. At first, JProfiling is called to train a DNN-based performance model for each micro-service. For each control step, logs are collected from Kubernetes to calculate the average pure processing time of each micro-service by eliminating the times of remote calls. During the feedback control period, output errors are computed through Equation (17) and used to calculate the adjusted arrival rate λ_i^c based on Equation (15) and the cooling-down strategy CDRatio for $\omega_i(k)$. Next, the final arrival rate λ_i^u of each micro-service is calculated based on JQN via Equations (3) and (4) considering the impacts among micro-services. Finally, the container number N_i of each micro-service is obtained using DNN-based performance model according to Equation (7) and the container number cooling-down strategy CDNumber, and sent to Kubernetes for adjusting.

Algorithm 3 DNN-FC

```
Input: $W_i^r$ reference times, $[\omega^{lower}, \omega^{upper}]$, $[\varphi^{lower}, \varphi^{upper}]$
1:  Call JProfiling() for each service upon deployed.
2:  while True do
3:      for every control step $k$ do
4:          for $S_i \in S$ do
5:              Update service times based on trace logs;
6:              Calculate $e_i(k)$ using Equation (17);
7:              $u_i(k) = K_p e_i(k)$;
8:              if IsFluctuating() then
9:                  $\omega_i(k) \leftarrow 1$;
10:             else
11:                 $\omega_i(k) \leftarrow$ Equation (15);
12:             end if
13:             $\omega_i(k) \leftarrow$ CDRatio($\omega_i(k)$);
14:             $\varphi_i(k) \leftarrow \varphi_i(k-1) \times \omega_i(k)$;
15:             Limit $\varphi(k)$ within $[\varphi^{lower}, \varphi^{upper}]$;
16:             $\lambda_i^c = \lambda_i \times \varphi_i(k)$;
17:         end for
18:         Sort $S_i \in S$ in topological order;
19:         for $S_i \in S$ do
20:             $\lambda_i^u \leftarrow$ Equations (3) and (4), by inputting $\lambda_i^c$;
21:         end for
22:         for each $S_i$ do
23:             $N_i \leftarrow$ Equation (7), by inputting $\lambda_i^u$;
24:             $N_i \leftarrow$ CDNumber($N_i$);
25:             Adjust containers of Kubernetes based on $N_i$;
26:         end for
27:     end for
28: end while
```

5 Performance Evaluation

Our approaches are first compared with QFC [14] which is the state-of-art algorithm for meshed multi-instance micro-service applications. Then, the proposals are compared with the Kubernetes' built-in Horizontal Pod Autoscaler (HPA) [28] which dynamically adjusts Pod counts based on CPU utilization. Finally, our approaches are also compared with the C-DQN method [24], which integrates a centralized DQN and heuristic rules in a hybrid framework. Algorithms are evaluated on a real Kubernetes platform which is established on a cluster with one master virtual machine and 8 slave virtual machines. These virtual machines locate on a physical cluster including two machines with 40 CPU cores and 80 GB Memory, and one machine with 64 cores and 128 GB Memory.

The test book-info application consists of four micro-services. To simulate micro-services that consume varying amounts of computational and memory resources, the original four micro-services have been replaced with four micro-services that recursively compute Fibonacci numbers. The access paths among these micro-services remain unchanged. Each micro-service randomly computes between 10 and 70 Fibonacci numbers. The micro-services are deployed in Tomcat Web servers with connection time-out of 2 seconds. Each replica of the micro-service is deployed in a Pod with one CPU core and 500 Mi Memory. The Traefik is applied as the reverse proxy and load balancer of each micro-service using the Dynamic Round Robin load-balancing algorithm. The JMeter is used to generate high concurrency user requests which is deployed on a physical machine with 8 CPU cores and 16 GB Memory. To simulate the fluctuation of arrival rates, the accessing history of Wikipedia [29] and NASA-HTTP [30] websites are adopted to change the

TABLE 3: SVRs of algorithms with and without CD and FS operators

Service	Wiki						NASA					
	QFC		DNN-FC				QFC		DNN-FC			
	FFF	TTT	FFF	FFT	TTF	TTT	FFF	TTT	FFF	FFT	TTF	TTT
S_1	25.95%	26.66%	20.62%	23.73%	12.01%	7.9%	31.36%	18.36%	33.43%	28.63%	12.57%	9.04%
S_2	1.27%	0.0%	12.01%	10.58%	5.22%	2.82%	3.39%	0.0%	13.68%	8.46%	2.4%	1.27%
S_3	4.23%	0.0%	9.18%	7.48%	2.12%	3.39%	4.94%	0.0%	9.17%	2.4%	1.69%	0.85%
S_4	0.28%	0.0%	11.58%	4.51%	2.4%	2.12%	0.42%	0.14%	18.05%	2.12%	3.67%	2.68%
Total costs	27192	30142	27708	28025	29400	29453	34590	38675	35033	36757	37612	37496

number of requests per seconds over time.

Several combinations of reference response times for services are given in advance by hand. How to select an appropriate reference time combination for services is left as future work. Let W_i^r be the allocated reference response time of S_i. The service time **upper-limit** $upper\_thr_i$ of each micro-service is set to W_i^r and $lower\_thr_i = W_i^r - 0.02s$. $[\omega^{lower}, \omega^{upper}]$ and $[\varphi^{lower}, \varphi^{upper}]$ are set to be $[0.95, 1.1]$ and $[0.7, 1.3]$ based on experimental comparison. L_q is set to 3, balancing the trade-off between quick response and stability. K_p is set to 1 for quick response. The increase cooling-down period should be set as short as possible while still allowing sufficient time for container setup. Conversely, the decreasing cooling-down period cannot be too short to ensure stabilization. Therefore, the values are set as $\nu^u = 2$ for the increasing phase and $\nu^d = 20$ (same with HPA) for the decreasing phase. The upper-limit T_p of 95^{th} response time of path p_2=($S_1 \rightarrow S_2$) and p_3=($S_1 \rightarrow S_3 \rightarrow S_4$) are set to be 0.7s and 1.05s according to the computation complexity of the tested micro-services, respectively. The mean response time violation ratio of a path (**PVR**) represents the proportion of the average response time of the path that exceeds the sum of reference times for the services on that path. The 95^{th} response time violation ratio of a path (95^{th}**PVR**) is the fraction of instances where the 95^{th} response time of the path surpasses a predefined threshold T_p. The threshold of PVR and 95^{th}PVR is set to be 5%. The ratio of average response times for S_i that surpass the predefined reference time W_i^r is termed the Service reference time Violation Ratio (**SVR**). This metric represents the ability of auto-scaling algorithms to adhere to the specified service reference times. The control interval is set to 15 seconds and containers are charged by intervals of 15 seconds too.

5.1 Ablation experiments and parameter tuning

To illustrate the performance of CDNumber, CDRatio and IsFluctuating, algorithms with and without these operators are evaluated. QFC-FFF means QFC without any operators, while QFC-TTT means QFC with all operators. In other words, the first "F" or "T" indicates whether the first operator, CDNumber, is turned off or on. Similarly, the second "F" or "T" signifies whether CDRatio is turned off or on, and the third "F" or "T" indicates whether IsFluctuating is turned off or on. Because CDNumber and CDRatio need to work collaboratively, they need to be turned on or off together. In this experiment, reference times are set to be $0.21s$.

In table 3, DNN-FC-TTF's SVRs are lower than those of DNN-FC-FFF, which depicts that the CD operators are helpful to decreasing SVRs. The reason is that CD operators are able to avoid releasing containers too frequently. When the arrival rate only decrease temporarily, the algorithms without CD operators are likely to release containers too hastily. If the arrival rate restore to original high values suddenly, the remaining containers are not able to cope with them leading to high SVRs. Similarly, DNN-FC-FFT is able to decrease the SVR compared with DNN-FC-FFF because IsFluctuating is able to supress the switching between states with excess and insufficient resources repeatedly. Finally, DNN-FC-TTT gets the best performance than all other methods which proves that using CD and FS operators collaboratively is able to improve the performance further. In the following sections, all of our approaches and QFC contain CD and FS operators and "TTT" is omitted.

The Mean Absolute Error (MAE) has been employed to evaluate how well the DNN-based performance model fits the sample data under different network parameters. For example, for hidden-layer sizes of (30, 30), (60, 60), (120, 120), and (240, 240), MAEs are 28.80 ms, 28.26 ms, 27.99 ms, and 29.53 ms, respectively, while the training times are 26.6s, 28.3s, 37.2s, and 46.6s (offline training). Longer training durations typically necessitate larger training datasets to achieve model convergence. Therefore, the (60, 60) hidden-layer size is selected as it optimally balances the trade-off between MAE performance and computational efficiency.

5.2 Performance of HPA with different CPU limitations

In this experiment, the performance of HPA with different CPU-utilization limitations (HPA-CPU-x%, x is the limitation value) is evaluated. Table 4 and 5 show the SVRs of Kubernetes's HPA [28] with different CPU-limitation and $W_i^r = 0.20s, 0.21s, 0.22s$ or $0.23s$. A larger CPU-limitation means allowing more requests processed on each container per second leading to longer average response times. HPA-CPU-40%, HPA-CPU-50% and HPA-CPU-60% obtain gradually higher SVRs. When a lower CPU-limitation is set, it usually needs to allocate more containers to the micro-service incurring a higher resource cost. On Wiki trace, the costs of HPA-CPU-40% are 33041 which is higher than those of HPA-CPU-50% and HPA-CPU-60% greatly. For the same CPU-limitation, the SVR increases as W_i^r decrease from 0.23s to 0.20s. It is much harder to guarantee W_i^r of micro-services with calls to others. For example, SVRs of S_1 and S_3 are much higher than those of S_2 and S_4 under the

TABLE 4: SVRs and costs of Kubernetes's HPA with different limitations of CPU-usage percentage on the Wiki workload trace

Service	HPA-CPU-40%				HPA-CPU-50%				HPA-CPU-60%			
	0.20	**0.21**	**0.22**	**0.23**	**0.20**	**0.21**	**0.22**	**0.23**	**0.20**	**0.21**	**0.22**	**0.23**
S_1	65.25%	14.97%	1.98%	1.55%	95.34 %	74.01%	35.17%	10.73%	98.59%	92.09%	73.73%	47.32%
S_2	0.71%	0.0 %	0.0%	0.0%	0.85%	0.14%	0.0%	0.0%	9.04%	1.27%	0.56%	0.14%
S_3	12.57%	0.99%	0.0%	0.0%	74.15%	32.49%	9.75%	1.41%	79.8%	51.55%	24.86%	9.46%
S_4	1.98%	0.0 %	0.0 %	0.0%	21.89%	1.98%	0.0%	0.0%	26.27%	5.65%	0.85%	0.0%
Total costs	33041				27346				22724			

TABLE 5: SVRs and costs of Kubernetes's HPA with different limitations of CPU-usage percentage on the Nasa workload trace

Service	HPA-CPU-40%				HPA-CPU-50%				HPA-CPU-60%			
	0.20	**0.21**	**0.22**	**0.23**	**0.20**	**0.21**	**0.22**	**0.23**	**0.20**	**0.21**	**0.22**	**0.23**
S_1	82.77%	32.91%	6.36%	1.55%	98.03%	83.5%	48.66%	16.5%	99.15%	92.23%	76.84%	49.29%
S_2	27.93 %	5.22%	1.41%	1.27%	3.67%	0.85%	0.56%	0.56%	31.36%	7.77%	3.81%	1.84%
S_3	24.82%	3.53%	1.41%	1.41%	70.52%	31.31%	9.87%	2.68%	93.36%	67.51%	28.53%	12.57%
S_4	1.55%	0.85%	0.7%	0.42%	4.8%	0.71%	0.28%	0.14%	5.93%	2.4%	1.55%	0.99%
Total costs	43978				35479				29676			

TABLE 6: Response time violation ratio of path-SLAs obtained based on 210 ms reference times

SLA	HPA-CPU-40%		HPA-CPU-50%		HPA-CPU-60%		QFC		DNN-FC	
	Wiki	**NASA**	**Wiki**	**NASA**	**Wiki**	**NASA**	**Wiki**	**NASA**	**Wiki**	**NASA**
PVR for Path 1-2	1.41%	12.29%	10.17%	16.93%	44.92%	58.19%	1.55%	1.13%	2.82%	2.12%
95^{th} PVR for Path 1-2	0.14%	1.84%	0.85%	3.81%	11.44%	17.8%	0.0%	0.0%	0.85%	0.0%
PVR for Path 1-3-4	0.71%	2.4%	20.76%	14.81%	44.77%	38.56%	1.55%	1.55%	1.97%	1.98%
95^{th} PVR for Path 1-3-4	0.14%	0.71%	0.71%	1.97%	3.53%	5.79%	0.71%	0.71%	1.41%	0.14%
Cost	33041	43978	27346	35479	22724	29676	30142	38675	**29453**	**37496**

same CPU-limitation. For HPA-CPU, the allocated number of containers is purely adjusted based on the CPU utilization without considering W_i^r. Because there is no explicit and constant relationship between the response time and the CPU utilization, it is hard to satisfy W_i^r by setting CPU-utilization limitations.

5.3 Results under different reference times

Different from HPA-CPU which only focuses on CPU-utilization, QFC and DNN-FC adjust resources based on allocated W_i^r directly. In other words, QFC and DNN-FC are able to adjust resources to follow the given W_i^r. One of the main objectives of experiments in this section is to compare the ability of satisfying diverse given reference times and minimizing costs.

Table 6 shows PVRs and 95^{th} PVRs for different paths when reference times are set to be $W_1^r = W_2^r = W_3^r = W_4^r = 0.21s$. As a whole, PVRs of QFC and DNN-FC are similar and all below the threshold 5%, and the costs of our approach DNN-FC are 2.29% and 3.05% lower than those of QFC on Wiki and NASA, respectively. HPA algorithms are all not able to fulfill the threshold 5% of PVRs. To find the reason of above results, SVRs of each micro-service are compared separately. Table 7 and 8 show SVRs under different W_i^r from $\{0.20s, 0.21s, 0.22s, 0.23s\}$. DNN-FC is

TABLE 7: SVRs and costs of QFC and DNN-FC on Wiki

Service	QFC				DNN-FC			
	0.20	**0.21**	**0.22**	**0.23**	**0.20**	**0.21**	**0.22**	**0.23**
S_1	-	26.66%	31.59%	11.85%	25.53%	7.9%	4.51%	4.94%
S_2	-	0.0%	1.97%	0.14%	2.4%	2.82%	4.23%	5.5%
S_3	-	0.0%	4.37%	0.28%	14.95%	3.39%	1.27%	1.41%
S_4	-	0.0%	1.41%	0.0%	1.27%	2.12%	4.09%	2.96%
Cost	-	30142	32414	25628	35166	29453	26996	24728

TABLE 8: SVRs and costs of QFC and DNN-FC on Nasa

Service	QFC				DNN-FC			
	0.20	**0.21**	**0.22**	**0.23**	**0.20**	**0.21**	**0.22**	**0.23**
S_1	-	18.36%	14.25%	8.33%	52.89%	9.04%	4.51%	2.82%
S_2	-	0.0%	0.0%	0.14%	0.85%	1.27%	1.69%	3.1%
S_3	-	0.0%	0.85%	0.99%	12.98%	0.85%	1.41%	2.12%
S_4	-	0.14%	0.0%	0.0%	0.28%	2.68%	1.41%	0.71%
Cost	-	38675	34892	32986	42130	37496	35279	31791

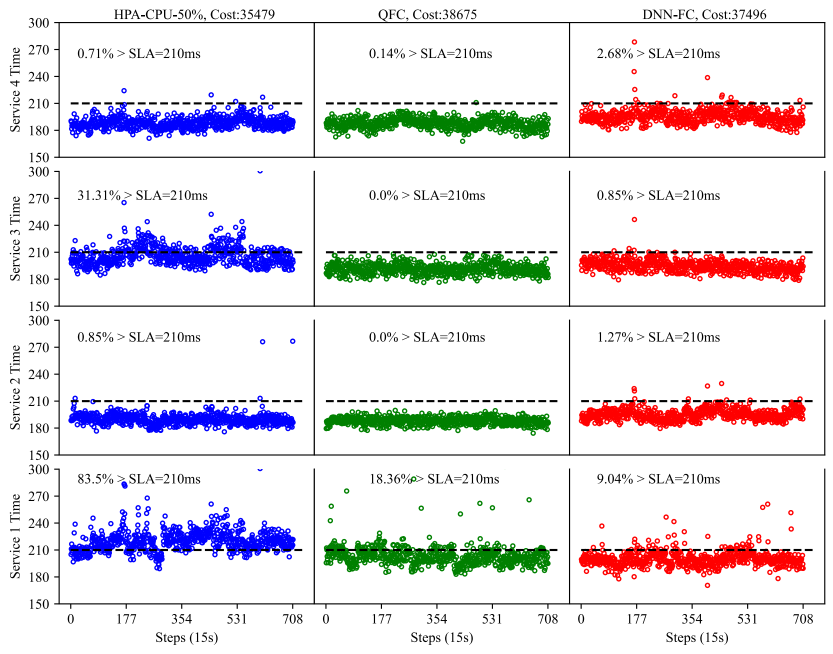

Figure 6: Service times of HPA, QFC and DNN-FC on NASA trace with reference times of 0.21s

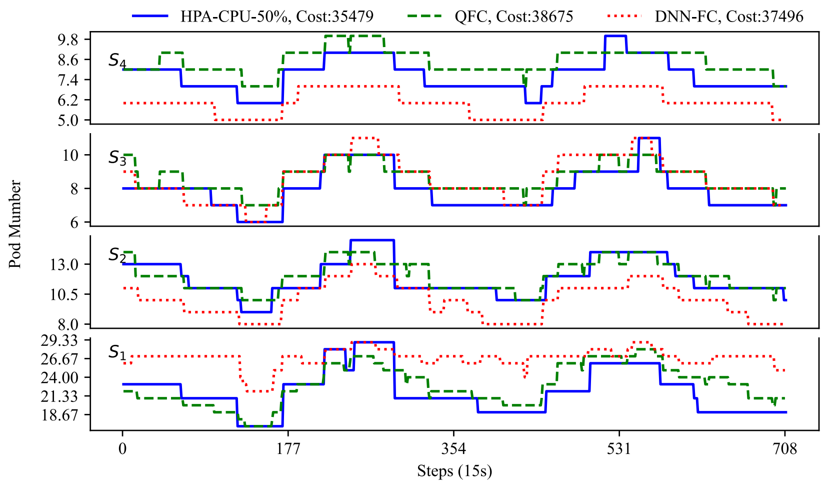

Figure 7: Container numbers of HPA, QFC and DNN-FC on NASA trace with reference times of 0.21s

able to follow W_i^r more closely. Compared with QFC, DNN-FC is able to decrease the SVR of S_1 greatly which is the most hard micro-service to control. Figure 6 denotes that QFC's average response times of S_2 and S_4 are more far from the reference times (dotted black horizontal lines) than the DNN-FC's. At the same time, a large part of QFC's average response times of S_1 are higher than the reference times. As shown in Figure 7, the reason is that excess amount of containers are allocated to S_2 and S_4 while insufficient amount of containers are assigned to S_1 by QFC because of inaccurate queuing models. QFC consumes significantly more resources across the majority of services, resulting in service completion times that are substantially lower than the reference times. Consequently, QFC is more robust to abrupt workload spikes, resulting in lower service violation ratios compared to DNN-FC across most services. However, this enhanced robustness is achieved at the expense of higher cost. Meanwhile, QFC does not work when $W_i^r = 0.20s$ (labeled '-') because the theoretical response time of M/M/N in QFC cannot be below 0.20s. As a whole, DNN-FC is able to follow the given reference times more closely by the aid of the DNN-based performance model.

Different micro-services require diverse amount of control effort. For example, we need allocate more containers to S_3 than to S_4 given the same W_i^r because each request of S_3 need to invoke S_4 and wait for the return consuming more resources. Therefore, it is critical to find an appropriate path-SLA division which minimize the total cost while satisfying the path-SLA constraint. For example, Table 9 denotes PVRs when the reference times are set to be $W_1^r = W_3^r = 0.245s$, $W_2^r = W_4^r = 0.205s$. For our approach DNN-FC, the costs of containers are decreased from 29453 to 25248 and 37496 to 31563 for Wiki and Nasa compared to costs when all reference times are set to be $0.21s$ in Table 6, separately. The main reason is that when larger reference times are allocated to services S_1 and S_3 which are hard to control, fewer number

TABLE 9: Response time violation ratio of path-SLAs obtained based on 245 and 205 ms reference times

SLA	HPA-CPU-40%		HPA-CPU-50%		HPA-CPU-60%		QFC		DNN-FC		C-DQN	
	Wiki	**NASA**	**Wiki**	**NASA**	**Wiki**	**NASA**	**Wiki**	**NASA**	**Wiki**	**NASA**	**Wiki**	**NASA**
PVR for Path 1-2	1.41 %	1.69%	1.13%	2.54%	9.04%	12.57%	1.13%	1.83%	2.54%	2.82%	38.08%	15.54%
95^{th} PVR for Path 1-2	0.14 %	1.84%	0.85%	3.81%	11.44%	17.8%	0.85%	1.83%	1.69%	2.54%	39.35%	15.68%
PVR for Path 1-3-4	0.71%	1.13%	0.85%	1.41%	2.68%	4.8%	1.27%	1.83%	1.98%	2.4%	33.29%	4.94%
95^{th} PVR for Path 1-3-4	0.14%	0.71%	0.71%	1.97%	3.53%	5.79%	0.56%	1.41%	1.55%	1.83%	30.18%	5.08%
Cost	33041	43978	27346	35479	22724	29676	28483	35440	25248	31563	**20387**	**28573**

of containers are required. Meanwhile, under the new reference time division combination, the costs of DNN-FC are 11.36% and 10.94% lower than those of QFC, while PVRs of QFC and DNN-FC are all smaller than the threshold 5%. In other words, with the assistance of its precise response time control capability, DNN-FC is capable of achieving greater cost savings when service reference times are set more appropriately. After undergoing training for over 24 hours, C-DQN still fails to achieve highly satisfactory performance. The primary cause is that reinforcement learning requires a very long sampling time for initialization and can only gradually enhance its performance after an extended period of sampling. Our approach can function as a lightweight initialization protocol to accelerate the convergence of C-DQN during the bootstrapping phase.

When a new micro-service is deployed on a Kubernetes-based cluster, JProfiling usually takes about 21 hours to collect performance data automatically. The data sampling process can be shortened to about 5 hours by decreasing the number of samples. Experimental results show that the performance is not greatly impacted. For example, the 95^{th} PVR for path 1-3-4 is increased from 1.55% to 2.68%, but the cost is decreased from 25248 to 24867. Meanwhile, if JProfiling is allocated a shorter startup time, the DNN model can conduct online training as a compensatory mechanism. After that, the average training time for the DNN model is approximately 48 seconds. Additionally, the execution times for container auto-scaling using the trained model range from 0.2 seconds to 0.45 seconds which is able to fulfill the requirement of quick response for Cloud Native platforms.

6 Conclusions and Future Work

To address the challenges of container elastic provisioning in Kubernetes like systems, a container provisioning algorithm (DNN-FC) that integrates deep learning, Jackson Queuing Networks (JQN), and feedback control is proposed. Experimental results prove that lightweight neural network based performance model is able to accommodate complex nonlinear performance models better than existing queuing models. Furthermore, feedback control based on deep neural networks enhances the real-time adaptability of the JQN-based scheduling framework to fluctuating workloads. Additionally, tailored cooling-down strategies for feedback control can further stabilize performance. Developing faster profiling data collection methods and path-SLA division techniques for Cloud Native applications represents a promising direction for future work.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No.61972202).

References

[1] H. Cheng, Q. Li, B. Liu, S. Liu, and L. Pan, "Dgercl: A dynamic graph embedding approach for root cause localization in microservice systems," *IEEE Transactions on Services Computing*, vol. 17, no. 6, pp. 3417–3428, 2024.

[2] Z. Ding, S. Wang, and C. Jiang, "Kubernetes-oriented microservice placement with dynamic resource allocation," *IEEE Transactions on Cloud Computing*, vol. 11, no. 2, pp. 1777–1793, 2023.

[3] Z. Wen, Q. Chen, Q. Deng, Y. Niu, Z. Song, and F. Liu, "Combofunc: Joint resource combination and container placement for serverless function scaling with heterogeneous container," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 11, pp. 1989–2005, 2024.

[4] P. Raith, G. Rattihalli, A. Dhakal, S. R. Chalamalasetti, D. Milojicic, E. Frachtenberg, S. Nastic, and S. Dustdar, "Opportunistic energy-aware scheduling for container orchestration platforms using graph neural networks," in *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, Philadelphia, PA, USA, 2024, pp. 299–306.

[5] W.-K. Lai, Y.-C. Wang, and S.-C. Wei, "Delay-aware container scheduling in kubernetes," *IEEE Internet of Things Journal*, vol. 10, no. 13, pp. 11 813–11 824, 2023.

[6] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Diktyo: Network-aware scheduling in container-based clouds," *IEEE Transactions on Network and Service Management*, vol. 20, no. 4, pp. 4461–4477, 2023.

[7] S. Shen, Y. Feng, M. Xu, Y. Ren, X. Wang, V. C. Leung, and W. Wang, "Tango: Harmonious optimization for mixed services in kubernetes-based edge clouds," *IEEE Transactions on Services Computing*, vol. 17, no. 6, pp. 4354–4367, 2024.

[8] Chenyang Lu, Ying Lu, T. F. Abdelzaher, J. A. Stankovic, and Sang Hyuk Son, "Feedback control architecture and design methodology for service delay guarantees in web servers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 9, pp. 1014–1027, Sep. 2006.

[9] Y. Hu, G. Dai, A. Gao, and W. Pan, "A self-tuning control for web qos," in *International Conference on Information Engineering and Computer Science*. Wuhan, China: IEEE, Dec 2009, pp. 1–4.

[10] T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A multi-model framework to implement self-managing control systems for qos management," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, pp. 218–227.

[11] L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A discrete-time feedback controller for containerized cloud applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 217–228.

[12] K. Salah, K. Elbadawi, and R. Boutaba, "An analytical model for estimating cloud resources of elastic services," *Journal of Network and Systems Management*, vol. 24, no. 2, pp. 285–308, 2016.

[13] Z. Cai and R. Buyya, "Inverse queuing model based feedback control for elastic container provisioning of web systems in kubernetes," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 337–348, 2021.

[14] Y. Lei, Z. Cai, X. Li, and R. Buyya, "State space model and queuing network based cloud resource provisioning for meshed web systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 3787–3799, 2022.
[15] D. Jiang, G. Pierre, and C. H. Chi, "Autonomous resource provisioning for multi-service web applications," in *19th Proceedings of International Conference on World Wide Web (WWW)*, Raleigh, North Carolina, USA, 26-30 Apr. 2010, pp. 471–480.
[16] J. Rao, X. Bu, C. Z. Xu, and K. Wang, "A distributed self-learning approach for elastic provisioning of virtualized cloud resources," in *19th IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Singapore, 25-27 July 2011, pp. 45–54.
[17] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 3, no. 1, pp. 1–39, 2008.
[18] Y. Lei, Z. Cai, H. Wu, and R. Buyya, "Cloud resource provisioning and bottleneck eliminating for meshed web systems," in *13th IEEE International Conference on Cloud Computing (CLOUD)*, Virtual Conference, 18-24 Oct. 2020, pp. 512–516.
[19] H. Wu, Z. Cai, Y. Lei, J. Xu, and R. Buyya, "Adaptive processing rate based container provisioning for meshed micro-services in kubernetes clouds," *CCF Transactions on High Performance Computing*, vol. 4, p. 165–181, 2022.
[20] Q. Zong, X. Zheng, Y. Wei, and H. Sun, "A deep reinforcement learning based resource autonomic provisioning approach for cloud services," in *Collaborative Computing: Networking, Applications and Worksharing*, H. Gao, X. Wang, M. Iqbal, Y. Yin, J. Yin, and N. Gu, Eds. Cham: Springer International Publishing, 2021, pp. 132–153.
[21] Y. Fang and Z. Cai, "Reinforcement learning based heterogeneous resource provisioning for cloud web applications," in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; (HPCC)*, Yanuca Island, Cuvu, Fiji, 2020, pp. 206–213.
[22] W. Iqbal, M. N. Dailey, and D. Carrera, "Unsupervised learning of dynamic resource provisioning policies for cloud-hosted multitier web applications," *IEEE Systems Journal*, vol. 10, no. 4, pp. 1–12, 2015.
[23] Y. Kang, Z. Zheng, and M. R. Lyu, "A latency-aware co-deployment mechanism for cloud-based services," in *2012 IEEE Fifth International Conference on Cloud Computing*, Honolulu, HI, USA, 2012, pp. 630–637.
[24] D.-Y. Lee, S.-Y. Jeong, K.-C. Ko, J.-H. Yoo, and J. W.-K. Hong, "Deep q-network-based auto scaling for service in a multi-access edge computing environment," *International Journal of Network Management*, vol. 31, no. 6, p. e2176, 2021.
[25] L. Yazdanov and C. Fetzer, "Lightweight automatic resource scaling for multi-tier web applications," in *7th IEEE International Conference on Cloud Computing (CLOUD)*, Anchorage, AK, USA, 27 June-2 July 2014, pp. 466–473.
[26] "Bookinfo application," Istio Web site, Cloud Native Computing Foundation, 2024, https://istio.io/latest/docs/examples/bookinfo/.
[27] Z. Cai, D. Liu, Y. Lu, and R. Buyya, "Unequal-interval based loosely coupled control method for auto-caling heterogeneous cloud resources for web applications," *Concurrency and Computation Practice and Experience*, vol. 32, no. 23, pp. 1–16, 2020.
[28] "Horizontal pod autoscaling of kubernetes," Kubernetes Web site, Cloud Native Computing Foundation, 2024, https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
[29] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, 2009.
[30] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: The search for invariants," in *19th Proceedings of ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '96)*, Philadelphia, Pennsylvania, USA, 23-26 May 1996, pp. 126–137.

Zhicheng Cai (Member, IEEE) received the Ph.D. degree in Applied Computer Science from Southeast University, Nanjing, China, in 2015. He is an associate professor with the Nanjing University of Science and Technology, China. His research interests focus on resource scheduling in Cloud, Fog and Edge computing. He is the author of about 30 publications in journals such as IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Services Computing, IEEE Transactions on Cloud Computing, IEEE Transactions on Automation Science and Engineering and Future Generation Computer Systems and at conferences such as ICSOC, ICPADS, ISPA, ICA3PP, CLOUD, HPCC, SMC, CBD, and CASE.

Hang Wu received the M.Sc. degree in School of Computer Science and Engineering from Nanjing University of Science and Technology, China, in 2022. He is currently working as a Cloud Native Application development engineer. His main research interests focus on resource scheduling of Fog, Edge and Cloud Computing.

Xu Jiang received the MSc degree from the School of Computer Science and Engineering, Nanjing University of Science and Technology, China, in 2024. He is currently working toward the PhD degree with Nanjing University of Science and Technology, Nanjing, China. His main research interests focus on resource scheduling in Cloud and Edge Computing.

Xiaoping Li (Senior Member, IEEE) is a distinguished professor in Guangdong University of Technology. He received the Ph.D. degree in Applied Computer Science from the Harbin Institute of Technology, Harbin, China, in 2002. He is the author or co-author over more than 100 academic papers, some of which have been published in international journals such as IEEE Transactions on Computers, IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Services Computing, etc.

Rajkumar Buyya (Fellow, IEEE) is a Redmond Barry distinguished professor and director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne, Australia. He has authored more than 625 publications and seven text books. He is one of the highly cited authors in computer science and software engineering worldwide (h-index=170, g-index=324, more than 155,200 citations). Microsoft Academic Search Index ranked him as #1 author in the world (2005-2016) for both field rating and citations evaluations in the area of distributed and parallel computing. He is recognized as a "Web of Science Highly Cited researcher" during 2016-2021 by Thomson Reuters.