AlphaR: Learning-Powered Resource Management for Irregular, Dynamic Microservice Graph

Xiaofeng Hou[1], Chao Li[1], Jiacheng Liu[1], Lu Zhang[1], Shaolei Ren[2], Jingwen Leng[1], Quan Chen[1], and Minyi Guo[1]

[1]Department of Computer Science and Engineering, Shanghai Jiao Tong University
[2]Department of Electrical and Computer Engineering, University of California, Riverside
Emails: {xfhelen, chaol, liujiacheng, luzhang, leng-jw, chen-quan, myguo}@sjtu.edu.cn, shaolei@ucr.edu

***Abstract*—The microservice architecture is a hot trend which proposes to transform the traditional monolith application into massive dynamic and irregular small services. To boost the overall throughput and ensure the guaranteed latency, it is desirable to process massive service requests in parallel with efficient resource sharing in data centers. However, the disaggregation nature of microservice unavoidably upscales the design space of resource management and increases its complexity.**

In this paper, we propose AlphaR, a learning-powered resource management system tailored to the microservice environment. The basic idea of AlphaR is to generate microservice-specific resource management policies for improving efficiency. Specifically, we take the first step to use bipartite graph as a convenient abstraction for application built with microservices. Based on this, we devise a bipartite feature inference approach named Bi-GNN to extract the temporal characteristics of microservices. Furthermore, we implement a policy network to select appropriate resource allocation choices for maximizing the performance in resource-constrained data centers. AlphaR can improve the mean and p95 response time by up to 80% and 77.5% respectively compared with conventional schemes.

I. Introduction

In recent cloud system designs [43], there is a trend to factorize a monolithic application into massive microservices [16], [24], [41], [45]. The new architecture allows data centers to balance user load distribution [6] and improve the overall system throughput [29]. Many cloud computing giants such as Microsoft [5], Google [45] and Amazon [42] are actively accelerating their adaption of microservices.

However, it is a daunting task to devise an efficient resource manager for the new applications due to the irregular and dynamic characteristics of their small services. As shown in Figure 1, a cloud application is disaggregated into multiple microservices organized as different graphs. In Figure 1(b), we use larger circles to represent microservices of more importance and thicker arrows to indicate higher user load. We observe that the characteristics of the microservices dynamically vary with various user load [30]. Particularly, the most critical microservice which can be the dominant performance factor of the whole application (as circles marked in red) often changes. To efficiently share the computing resources in data centers, it is quite crucial to expose the irregularity and dynamicity of microservices to the resource management.

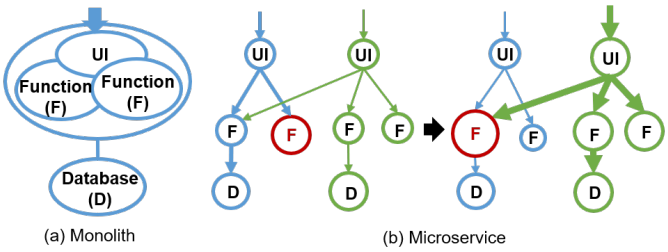


Figure 1: Microservice discomposes traditional monoliths into massive, irregular and dynamic service graphs.

Moreover, managing resources for microservices would be more complex in production data centers. Specifically, there are hundreds and thousands of microservices at the cluster level, which leads to a larger problem size both in the amount of information and the number of choices for the resource manager. Meanwhile, the manager needs to be aware of the irregular characteristics of numerous microservices. This includes obtaining the multi-dimensional resource requirements of different microservices and identifying their importance to the performance of the entire application. It also needs to be self-optimizing in response to the unexpected changes of the environment. This requires that the system can automatically make decisions without any human intervention whenever the execution states such as user load change.

Current approaches are insufficient to handle the complexity of resource management in microservice era. Most of the data centers use straightforward schemes [7], [25], [36], [37] or generalized heuristics [11], [14], [28] to make resource allocation decisions. They oftentines overlook the relationship between the allocated resource of each individual microservice and the overall performance. As a result, it could waste precious resource on some less critical microservices. Some other existing resource management approaches are static or offline-tuned based on pre-defined profiles of different applications [15], [29]. They tend to require active actions, which often incur unnecessary overheads like long response latency. Compared to the existing methods, graph neural network (GNN) techniques are promising in enabling efficient resource management approach since they excel at handling the complex issues in an automatic way.

In this paper, we propose AlphaR, a learning-powered

resource management system that leverages GNN to generate microservice-specific resource management policies. To handle the irregularity and dynamicity of microservices, we take bipartite graph (BG) as a new abstraction for applications built with microservices. A well-defined BG can incorporate the multi-dimensional resource requirement of different microservices as well as the relationship between the allocated resource and the application performance. Based on this abstraction, we devise a bipartite graph neural network (Bi-GNN). The core part of Bi-GNN is an adaptive bipartite feature inference strategy that can extract the time-varying characteristics of microservices. Meanwhile, we also implement a policy network to select the allocation choices that could maximize the gross normalized product (GNP) with limited resources. Our experimental results show that AlphaR can improve the mean response time by up to 80% compared with the existing schemes. To sum up, we make the following contributions:

1) We examine the characteristics of microservices through extensive analysis and experiments. For synthesizing their heterogeneous characteristics and interconnections, we take bipartite graph (BG) as the new presentation of an application implemented by microservices.
2) We construct AlphaR, a learning-powered resource management system of microservices, which translates the problem of resource management into a learning problem. It takes BGs as inputs and automatically outputs the optimal resource allocation policy.
3) We simulate AlphaR with realistic trace as input. We define gross normalized product (GNP), a new metric for evaluating overall performance with consideration of request latency and throughput. We show that AlphaR greatly outperforms the state-of-the-art schemes.

The remainder of the paper is organized as follows. Section 2 introduces the background and motivation. Section 3 analyzes the characteristics of microservices. Section 4 describes the design and implementation details of the AlphaR. Section 5 presents experimental methodologies. Section 6 demonstrates the important experimental results. Section 7 discusses related work. Finally, Section 8 concludes this paper.

II. Background and Motivation

A. Challenges of Microservices

In recent cloud-native design, microservice is the main way to build large, complex applications with scenario-focused mini services. As shown in Figure 1, unlike a monolithic application that consists of several huge functions, microservices are structured with massive bipartite service graphs [9], [30], [41], [44]. In the service graphs, an API layer consists of multiple API microservices which mainly act as the access portal. A Service layer contains massive loosely-coupled function services and their separate database microservices. These function microservices are mainly responsible for completing the requested service such as data analysis, etc. With the unique two-tier topology, user queries always access many mini services through the given API layer.

Service graphs of microservices often have irregular and dynamic characteristics, which make the microservice resource management a more complex issue. Specifically, the smaller size of microservice significally scales up the service number as well as exposes irregular characteristics in data centers. Meanwhile, each microservice's characteristics dynamically vary with unpredictable changes of user behaviors. In this case, even if each service graph might be small, the irregular, dynamic characteristics of numerous microservices requires the resource manager to be able to automatically make decisions whenever the user load changes. Particularly at the cluster level, there are hundreds and thousands of microservices, which leads to a larger problem size both in the amount of information and the number of choices for the resource manager.

B. Limitations of State of the Art

The existing approaches allocate resources by simply discerning resource or performance requirements of different applications [7], [25], [36], [37]. They overlook the dependency among these small services and their importance to the performance of the entire application. Thus, they can hardly make informed decisions when facing applications composed of multiple microservices. Traditional heuristics-driven policies provide application-specific solutions [11], [14], [28]. These methods initially profile resource-performance relationship of various applications. After that, they continuously tune parameters and validate until a satisfactory model is obtained. For microservices, the procedure of tuning might be endless before convergence since the features of each mini service change with the unpredictable variety of the environments.

Resource management in the microservice era must be able to identify different microservices' characteristics and their effects on the whole performance in a highly dynamic environment. Compared to the simple strategies or ad-hoc heuristics, GNN approaches excel at computing the characteristics of irregular and dynamic graphs. Therefore, they are promising in benefiting the cloud-native system with intelligent resource allocation, choosing the best suitable strategy based on execution states [27]. At present, ML is pervasively used to discover relevant features in many other fields. Similarly, the features relative to scheduling could be very powerful and critical in targeting the optimal scheduling. Generally, wherever we use heuristics, we can leverage ML to improve the accuracy and effectiveness of decision making in resource management.

III. Analysis and Characterization

In this section, we explore the heterogeneity and dynamicity of microservices by analyzing the trace data of a production data center (global view) and characterizing a running application composed of over 40 microservices (local view).

A. Analysis with Realistic Cluster Data

We analyze microservice behavior with the cluster trace released by Alibaba group in 2018 [1]. Alibaba implements most of their online services based on microservice architecture [4]. The trace is collected from a production data center with around

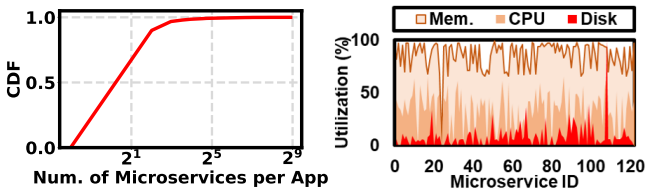
Trace characteristics	Value
Time span of trace	8 days
Number of machines	4000
Amount of applications	600
Usage records	28 GB
Container meta data	2.4 MB

Table I: Raw data from the realistic data center [13].

4000 machines in a period of 8 days. The trace contains rich information/metadata and Table I summarizes the key statistical properties of the trace. We mainly use two data files, namely *container_meta.csv* and *container_usage.csv*. These two files respectively describe the meta information and resource usage of all the containers hosting online applications of Alibaba. The real names of the applications have been obfuscated for confidentiality. Each application involves many containers and each container almost executes a unique kind of microservice.

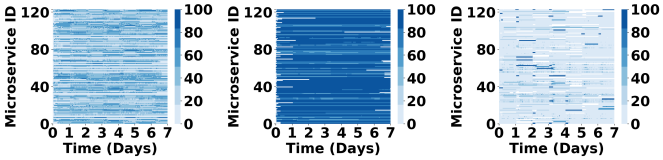
We observe that the number of microservices changes a lot (**dynamicity**) in the production data center. We first count the quantities of microservices constituting each individual online application in the Alibaba data center in one day. The statistical result shows that these applications contain several to tens of hundreds microservices. As shown in Figure 2(a), over 90% online applications contain about 25 microservices and about 5% applications even include over 100 microservices. We then randomly choose an online service numbered as *app_66* and observe its component microservices during 24 hours. Our result also demonstrates that the number of microservices in *app_66* varies a lot in the whole day.

We show that microservices belonging to the same service graphs use distinct resources (**irregularity**) during execution. We plot the resource utilization of each microservice in *app_66* during a week as shown in Figure 3(a)-(c). The result shows that each microservice's utilization changes sharply. Many microservices consume very little CPU (as shown in Figure 3(a)) for most of the times. This is because each microservice is dedicated to a simple task without large computation. In



(a) Upsizing Number (b) Granular Demand

Figure 2: Microservice upscales service density and exposes granular service demands in production data centers [13].



(a) CPU Utilization (b) Mem Utilization (c) Disk Bandwidth

Figure 3: The resource utilization of the microservices changes a lot in the realistic cluster [13].

Node name	Role	Description
Server A	Zipkin/UI	Observation interface.
Server B	User program	Sending user requests.
Server C1,3	Host server	Running microservice.

Cluster's server configurations	
Cluster	4 host servers + 1 interface server
Server	Dell Edge Power R730 6-core, 2.4 GHz
Host OS	Ubuntu trusty kernel running docker

Table II: Testbed configuration in our experiment.

Figure 3(b), almost all the microservices utilize much memory during the 7 days because of its data-driven design. Figure 3(c) demonstrates that only a few microservices take up a lot of disk bandwidth since most infrastructure microservices are stateless without database [44].

B. Testing Configurations and Tools

We characterize the irregularity and dynamicity of microservices through experimental analysis. We build up a cluster environment which contains 2 server nodes with docker swarm. As shown in Table II, the interface node provides web interfaces for gathering timing data of each microservice related to user's requests. We deploy all the microservices in the cluster for debugging specified microservice properties and ensuring functionality integrity of the application. The interface node is a six-core system with a peak frequency of 2.4GHz. Docker swarm leverages a fair docker scheduling algorithm (round-robin) to deploy all the related microservice dockers among the two server nodes. Besides, we run a user program on the client node to continuously access the running service.

We experiment with TrainTicket [31], a railway ticketing application implemented by microservices. The number of microservices in TrainTicket is more than any other existing benchmarks [31]. In the following experiment, we mainly consider two services each serving a kind of request in TrainTicket, i.e., *Advanced Ticketing* service and *Basic Search* service. We deploy TrainTicket in the cluster. We bound the user interface microservice with the 80 port on the interface node. The system fairly schedules the remaining microservices on the two servers. We deploy these microservices with containers and each container only executes a single microservice. By collecting the request-tracing data with Zipkin [3], we can obtain the request response time and the microservice execution time. We repeat our experiment for 1000 times and report the average results. We study the CPU, memory and disk resources in total. Table IV describes the tools for monitoring and controlling these resources. We write Python programs to adjust the ratio of requests accessing the services.

C. Characterizing Active Microservices

To exploit the irregularity and dynamicity of service graph in the real world, we monitor the resource consumption of each microservie. We analyze the relationship between microservice performance (response time) and its allocated resources.

We first evaluate how microservice workload varies with the incoming request types and quantities. We consider three different load scenarios, i.e., the ratio of requests accessing *Advanced*

A:B	\	config	basic	order	route	price	station	travel
0 : 100	CPU (%)	25.97	**183.13**	10.11	61.47	129.74	131.86	163.23
	Mem. (%)	2.6	5.058	3.59	4.41	5.23	5.44	**5.968**
	Net. (KB)	4.68	**11.14**	0.93	8.47	8.43	9.65	9.87
	Disk (MB)	11.86	10.43	**13.86**	5.19	3.94	10.43	5.45
	Delay (ms)	1.52	8.38	1.498	1.458	29.33	13.4	25.46
50 : 50	CPU (%)	25.07	205.92	4.41	60.18	113.98	131.28	**258.06**
	Mem. (%)	2.602	5.06	3.59	4.414	5.23	5.44	**5.98**
	Net. (KB)	4.69	**11.18**	0.94	8.49	8.45	9.68	9.92
	Disk (MB)	10.75	10.39	15.42	4.37	6.1	**15.56**	5.73
	Delay (ms)	1.46	10.206	0.13	1.39	64.24	12.64	114.81
100 : 0	CPU (%)	19.37	180.3	0.11	47.87	88.37	109.62	**284.9**
	Mem. (%)	2.6	5.06	3.59	4.42	5.23	5.43	**6**
	Net. (KB)	4.7	**11.18**	0.94	8.51	8.46	9.69	9.93
	Disk (MB)	10.67	10.98	**20.33**	6.51	9.73	8.39	7.45
	Delay (ms)	1.38	9.44	0.84	1.41	50.14	13.17	117.61

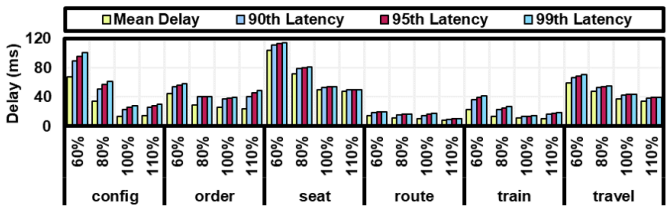
Table III: The effect of users' behaviors on granular demands.

Ticketing and *Basic Search* service is 0:100, 50:50 and 100:0, respectively. Table III illustrates the resource utilization of microservices in these scenarios. We can see that the resource utilization of a microservice changes with the request states. For example, when the ratio of requests accessing *Advanced Ticketing* and *Basic Search* service transfers from 0:100 to 50:50, *travel* becomes the most CPU-intensive microservice. With an increase in the number of requests accessing *Advanced Ticketing* service, *station* replaces *order* as the most write-intensive microservice. In this experiment, the input network of all the microservices is almost unchanged since the total number of requests is the same. The variation of memory usage is also closely related to user input.

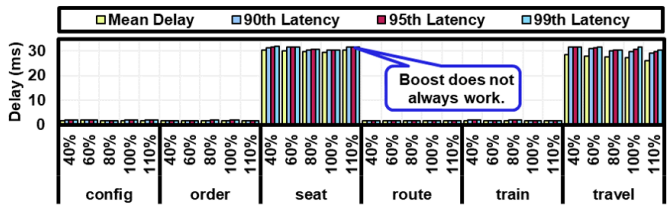
To further explore the relationship between allocated resource and performance, we characterize the variation of microservice's execution time under different resource allocation conditions. Specifically, we respectively limit the CPU, disk and memory usage of each component microservice in the *Advanced Search* service at 60%, 80%, 100% and 110% normalized to its original demand. For measuring the performance effect, we compute the mean response time (namely delay) of 1000 requests in each experimental setting.

Figure 4 shows the results. It is obvious that each microservice in the *Advanced Ticketing* service is differently sensitive to the variation of the allocated resource. In Figure 4(a), *travel* is the slowest component microservice in *Advanced Ticketing* service when the allocated CPU of all the microservices is 100%. However, when the allocated CPU reduces to 60%, *seat* becomes the slowest ones. Namely, the performance of the entire application depends on different microservices once the allocated resource changes. Figure 4(b) and Figure 4(c) indicate that reducing the disk and memory of most microservices (like *config*, *route*) makes no difference on the performance of the entire application. This is because the execution time of these microservices at 60% allocation is still far less than the execution time of some microservices like *seat* and *travel* at 100% allocation. Notably, boosting doesn't always work such as the execution time of *seat* increases when being allocated 10% more disk resource as shown in Figure 4(b). In Figure 4(c), although *travel* is not sensitive to memory resource, it would fail if the allocated memory is less than 60%.

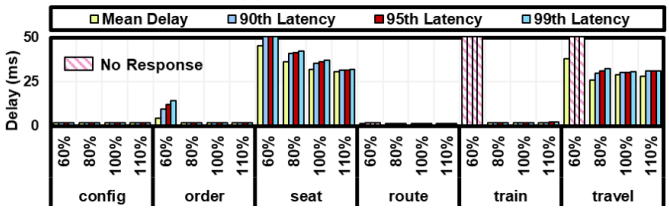
Summary: The microservice architecture leads to increasing



(a) CPU Utilization



(b) Disk Bandwidth



(c) Memory Utilization

Figure 4: The relationships between the allocated resources of different microservices and overall performance.

irregularity and dynamicity in data centers. Resource management of microservices should be able to learn their features to make informed decision. Considering the large problem size and complexity of resource management in the microservice era, it is promising to leverage learning methods to manage resources. Compared to conventional methods, learning-based approaches can make proactive decisions, which is more tailored to the characteristics of microservices.

IV. The Design of AlphaR

A. Overview

We propose AlphaR, a learning-powered resource management scheme. The nature behind AlphaR is to construct an agent with continuous environment states and rewards. The trained agent can automatically achieve microservice-specific resource management. As shown in Figure 5, it mainly contains three components.

1) *Interface Layer*: The interface layer module is responsible for interacting with the cloud environment as well as expressing the states and the rewards. It monitors the execution states of microservices, collects the resource information and executes the allocation decisions.
2) *Analysis Layer*: The analysis layer leverages a bipartite graph neural network (Bi-GNN) to infer microservice characteristics. It represents each application as a bipartite graph (BG) which expresses the irregularity and dynamicity of microservices with graphs.
3) *Execution Layer*: The execution layer ultimately produces the optimal resource managing strategies based on the inferred features. It implements a policy network to select

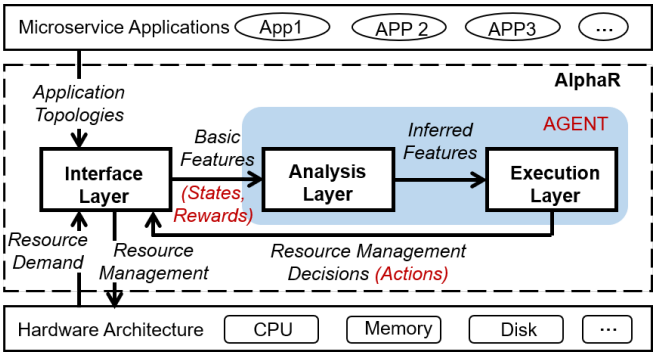


Figure 5: Overview of AlphaR.

allocation choices for maximizing the gross normalized product (GNP) with limited resources.

B. Interface Layer

We can extract performance information from the collected trace. The interface layer uses hierarchical message passing technique to interact with the agent. It contains a local data-collecting toolkit on each container running microservices and a global massage-passing network distributed across applications. The local data-collecting toolkit encapsulates several existing interfaces and tools to obtain the execution states of the containers. Table IV lists the related interfaces for monitoring and controlling resources like CPU cores, memory and disk. Our local data-collecting toolkit also relies on the lightweight distributed tracing systems such as *Jaeger* [2] and *Zipkin* [3] to obtain the end-to-end request time of each container. The global message-passing network consists of massive agents. As shown in Figure 6, the information collected with our local data-collecting toolkit is fed into the Bi-GNN to generate inferred features. Each Bi-GNN has an agent to report per-microservice features and per-application features to the upper controllers. Ultimately, these features are passed to the resource management module.

The interface layer is also responsible for executing the allocation decisions. As containers offer virtually isolated environments with namespace, microservices can execute independently wherever there are enough resources without worrying about compatibility [20]. Thus, the problem of executing resource allocation transfers into tuning the available resource for each container on local servers. Our interface layer implements multiple sub-controllers to adjust the resources of each containers through mature tuning knobs and configuration files. For example, it can directly use *RAPL* to limit the frequency of cores running specific microservices. It can restrict per-container usage of CPU, memory, disk by throttling the processes and threads that run the containers. For example, it can limit the container's I/O write bytes by writing the expected value to *blkio.throttle.write_bps_device*.

Resource type	Monitor	Control	Metric
Cores	dockerstats	cgroups cpuset	Utilization
Memory	dockerstats	turbostate	Utilization
Disk	dockerstats	cgroups blkio	Write BPS

Table IV: Resource management tools.

C. Analysis Layer

AlphaR uses a bipartite feature inference approach to learn the characteristics of different microservices As shown in Figure 6. It presents the applications as numerous bipartite graphs. A bipartite graph (BG) can effectively synthesize the basic characteristics such as uneven resource demands and interactions of the component microservices of an application. We compute the inferred features with the graph neural network (GNN) method [22]. Unlike traditional GNN focusing on homogeneous graphs with simple features, AlphaR employs a bipartite GNN (Bi-GNN) with a bidirectional message passing mechanism. Bi-GNN can reduce the huge action space incurred by the heterogeneity and dynamicity of microservices.

Specifically, we denote the BG as $g = (\mathbf{V_A}, \mathbf{V_F}, \mathbf{E})$ where $\mathbf{V_A}$ and $\mathbf{V_F}$ are two disjoint sets of vertices and $\mathbf{E}$ is the set of directed edges from vertex in $\mathbf{V_A}$ to vertex in $\mathbf{V_F}$. Each vertex in $\mathbf{V_A}$ and $\mathbf{V_F}$ respectively represents an API and a function service. The edges in $\mathbf{E}$ actually represents the dependencies of API microservices and function microservices, for example $e^i_j \in \mathbf{E}$ represents that the API microservice $i \in \mathbf{V_A}$ invokes the function microservice $j \in \mathbf{V_F}$. In the graph g, the basic feature of each microservice is their resource requirements. We assume n types of resources in the system. Each microservice requires different resources and we use a n-dimensional vector $\vec{h} \in \mathbf{R^n}$ to represent the resource requirements of a microservice. Thus, $\mathbf{H} = \langle \vec{h_1}, ..., \vec{h_p} \rangle$ represents the set of feature vectors of all the API microservices and function microservices, where p is the total number of API microservices and function microservices. Thus, the ultimate goal is to extract a set of inferred features, presented with $\mathbf{H'} = \langle \vec{h'_1}, ..., \vec{h'_q} \rangle$ where $\vec{h'}$ is a n-dimensional vector similar to $\vec{h}$. Vector $\vec{h'}$ specifie how many resources the microservice should be allocated.

Considering that not all microservices are equally important to the performance of the entire application, we propose a bipartite attention mechanism to iteratively update the inferred features. The bipartite attention reflects the uneven effects on each other between the API and function microservices. Specifically, the bipartite attention mechanism implements a forward attention and a backward attention. The forward attention updates the features of API microservices with the features of their related function microservices. The backward attention calculates the feature of function microservices based on its API microservices. In the bipartite graph g, we consider an API microservice i and denote its related function microservices by $\mathbf{A}_i$. Similarly, we also consider a function microservice j and denote the set of the API microservice invoking it by $\mathbf{F}_j$. We present the forward attention of microservice i as α_{ij}, where $j \in \mathbf{A}_i$. Then we leverage a neural network similar to work [26] with parameter a to calculate α_{ij} as,

$$\alpha_{ij} = \frac{\exp\left(\text{ LeakyReLU } \left(\vec{a}^T \left[M_\alpha h_i \| M_\alpha h_j\right]\right)\right)}{\sum_{k \in \mathbf{A}_i} \exp\left(\text{ LeakyReLU}(\vec{a}^T \left[M_\alpha h_i \| M_\alpha h_k\right])\right)} \quad (1)$$

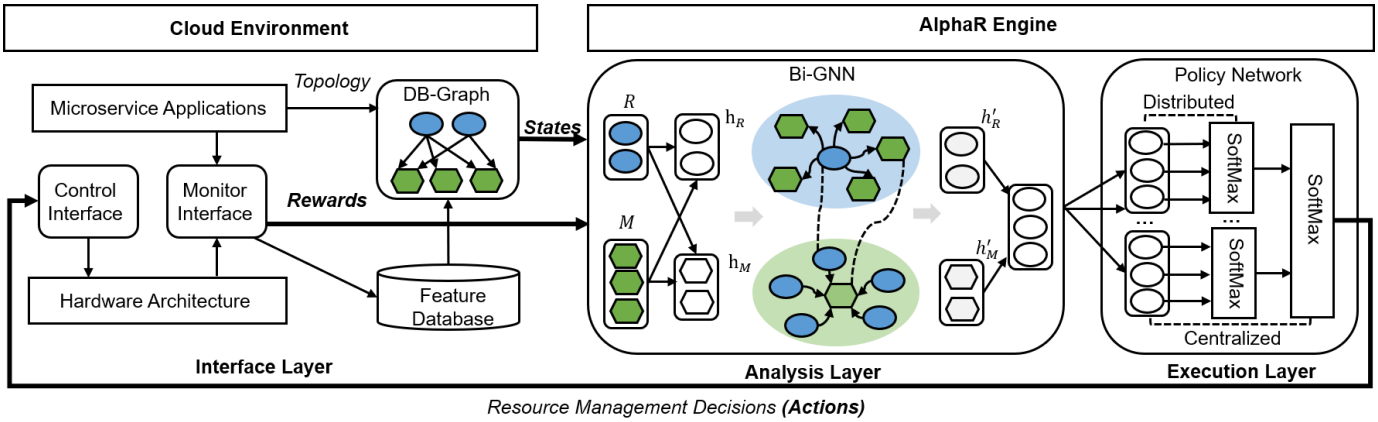


Figure 6: The architecture of the AlphaR. It aims at constructing an agent with continuous environment states and rewards.

where M_α is a weight matrix, $||$ is the concatenation operation and $\cdot^T$ is the transposition operation. The backward attention β_{ji} of function microservice j can be updated in the similar way with its invoked API microservice $i \in \mathbf{F}_j$.

This message passing phase can aggregate the attributes of the vertexes (both API and function microservices) and the edges. To exclude uncertainties in practice, we compute the microservice characteristics related to resource management with K loops in a message passing phase. Thus, the characteristic of the API microservice i is formulated as,

$$\vec{h}'_i = \|_{k=1}^{K} \sigma \left(\sum_{j \in \mathbf{A}_i} \alpha_{ij}^k M_q^k [h_j || e_j^i] \right) \tag{2}$$

where M_w is the weight matrix of the transformation, and σ is the nonlinear transform function. The backward message $\vec{h}'_j$ passed by function microservice is similarly computed.

D. Execution Layer

Even if we can obtain the temporal characteristics in the dynamic environment, it is still challenging to make scheduling decisions. First, assigning m types of resource to different microservices leads to an exponential action space such as at least 2^{mn} for n microservices. When considering the scheduling problem at the entire system level, n can be a quite huge number, which could make the learning process very time-consuming. Second, due to the variation of the quantities of microservices caused by service replacement, the scheduling systems could make fast decision when the input changes. To simplify the learning process, we decompose the decision-making process into two connected learning problems: 1) centralized budgeting on overall applications and 2) distributed allocation on local microservices. The centralized budgeting plans the resource capapcity for different applications according to the graph features. The distributed allocation implements massive local agents, each of which decides the resource among the microservices in an application.

As shown in Figure 6, the execution layer is composed of a two-level policy network. Each policy network contains three main functions: *getFeature()*, *allocate()* and *learn()*. The *getFeature()* function maps the application (node) features or microservice (graph) features to a scalar value for computing the probability of allocating resources to the graphs or microservices. The *allocate()* function takes the scalar values computed by *getFeature()* as inputs. It deploys a basic L layer perceptron (as follow) to make allocation decisions,

$$\mathbf{Y} = \text{Softmax}\left(\mathbf{M}_L \cdots \text{ReLU}\left(\mathbf{M}_1 \vec{\mathbf{H}}_q + \mathbf{b}_1\right) + \mathbf{b}_L\right) \tag{3}$$

After each allocation process, the *learn()* function receives a reward from the environment and optimizes the policy network with the stochastic gradient descent algorithm.

E. Implementation Details

We first denote the reward as r_t in each time slot t for training AlphaR in discrete timesteps. We begin with defining a metric named gross normalized product (GNP), which is deduced from an assessment metric of system quality in prior work [38]. Specifically, GNP is a measure of the degree to which the running performance accomplishes the requirement of service-level objectives (SLO). It equals to the ratio of the average performance AP, namely the mean response time of applications to the required performance RP. Generally, the required performance can be either the maximum response time by running some benchmarks [41] or the standard value given by a reputed organization [39]. Formally, the normalized product NP_i of application i is given by $NP_i = RP_i/AP_i = RP_i/AVG(req\_t_i)$. Thus, GNP is the sum of the normalized products in the system. Based on GNP, The reward should reflect the influence of resource management on the overall system performance. If $NP_i \geq 1$, the agent gets more reward, otherwise it gets less reward. Therefore, the reward r_t is computed with,

$$r_t = a \sum_{0}^{N_c} NP_i + b \sum_{0}^{N_t - N_c} NP_i \tag{4}$$

where a and b respectively reflect the weight of the normalized products of N_c applications whose $NP \geq 1$ and $N_t - N_c$ applications whose $NP<1$ to the overall reward.

The training process is to construct the optimal agent with continuous environment states and rewards. To minimize

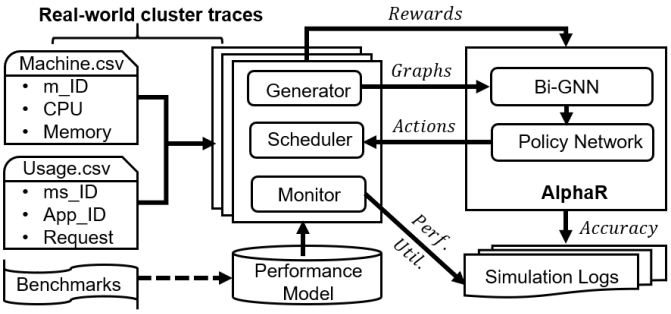


Figure 7: The simulator of AlphaR.

Category	Scheme	Descriptions
Simple scheduler	FairSched	Give each microservice equal resources.
	CurSched	Allocate resources based on current load.
	FullSched	Allocate resources based on accumulated load.
Heuristic profile	PartProfile	Allocate resources based on performance profile.
	FullProfile	Allocate resources based on service-level objectives.
RL-driven scheme	Alpha	RL-based policy with graph neural network (GNN).
	AlphaP	RL-based policy with graph attention network.
	AlphaA	RL-based policy with bipartite GNN.
	AlphaR	Our proposal.

Table V: The evaluated power management schemes.

the online training cost, we train the models offline and conditionally adjust them online according to the feedback. In each training iteration of time slot t, the agent observes the state (i.e. resource demand) s_t to build BGs, and chooses an action (i,e, allocation strategy) a_t, from the action space (all possible allocation strategies) $\mathbf{A}$. Following the action a_t, the environment transits to state s_{t+1} and sends a reward r_t to the agent. Then the agent optimizes its parameters according to the reward r_t. Considering T training slots in total, the goal is to maximize the total future reward R_T. To this end, we define the Q-function $Q(s_t, a_t)$ which represents the quality of a certain action a_t in a given state s_t. Then, for a agent with policy $\pi_\theta(a|s)$, the loss function can be represent as,

$$\mathcal{L} = -\sum_{i=0}^{T} \sum_{a_j \in \mathcal{A}} \pi_\theta(a_j|s_i) Q(s_i, a_j) \tag{5}$$

where s_i is sampled with π_θ. We stop training until the training loss stops falling in given iterations, namely, the value of the loss function stabilizes at a small value as the models converge.

V. Experiment Methodologies

We build an evaluation platform of AlphaR as shown in Figure 7. We simulate a large-scale, cloud-native system with the trace data of a realistic data center [13] as depicted in Table I. Since the original trace data hides some information like the application types, we further include two types of microservice applications, namely an industrial application named *Trainticket* [31] and an academic benchmark named *DeathStarBench* [35]. In other words, we assume that each container in the raw trace data executes a component microservice of these two applications. We run the simulator on servers with the configurations as depicted in Table II. We consider 6 dynamic and emergent user patterns as shown in Figure 8. These are derived from the realistic one. Particularly, we define gross normalized product (GNP), a new metric for evaluating overall performance with consideration of request latency and throughput. As depicted in Section 5.5, GNP is the total normalized product of all the applications.

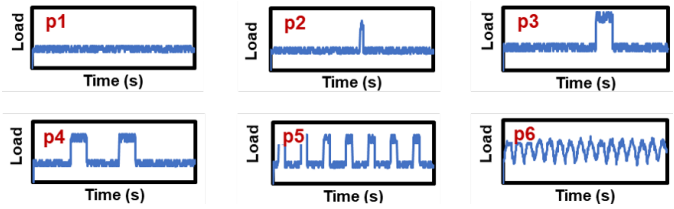


Figure 8: Peak user load patterns in the realistic cluster [13].

We compare AlphaR with several representative resource management schemes as summarized in Table V. Simple scheduler represents a group of methods, which allocate resources to different services to ensure that they are executed as required without any consideration of their performance features. *FairSched* represents the fair scheduling approaches which give each container equal resources [7], [25]. *CurSched* allocates resources to each container in accordance with its current load state, such as its task queue length, and *FullSched* manages per-container resources based on its historical and current load. Heuristic profile refers to a series of techniques that manage resources based on the performance model of individual services [29]. *PartProfile* allocates resources to various microservices simply depending on the relationship between its performance and allocated resources [29]. *FullProfile* represents the state-of-the-art workload-specific strategies that manage resources based on the QoS requirements of the whole application like Paragon [11]. We also establish some resource management schemes based on RL . Among these, AlphaR is our proposal. *Alpha* uses GNN to generate features relative to resource management. *AlphaP* deploys a graph attention network. *AlphaA* only leverages the bipartite feature inference to learn the characteristics of microservices.

VI. Experiment Results

A. Configurations for AlphaR

We first train AlphaR for generating the optimal decision-making model according to the implementation details as depicted in Section 4.5. Figure 9 shows the training process of AlphaR. As shown in Figure 9(a), During the initial training iterations, AlphaR cannot efficiently allocate resources, completing only a small number of requests within the required

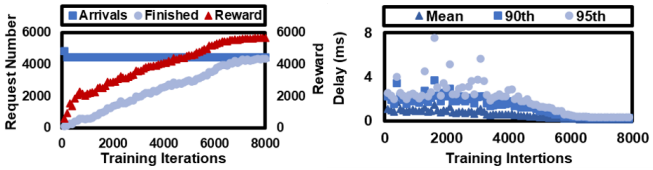
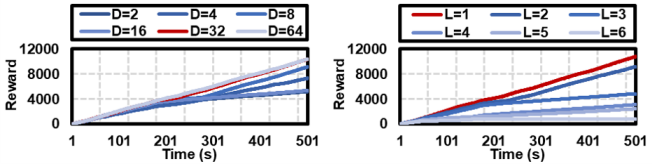
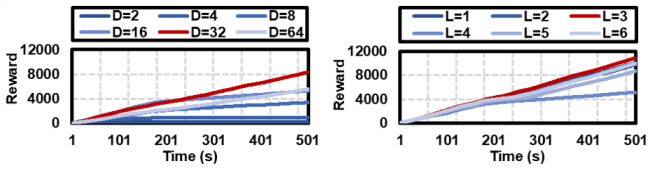


Figure 9: Training iterations of AlphaR.

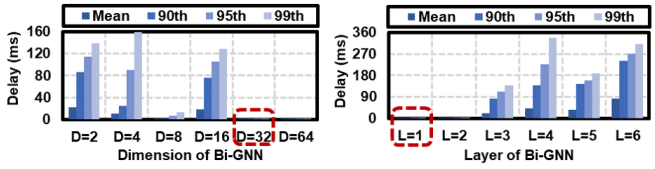


(a) Bi-GNN

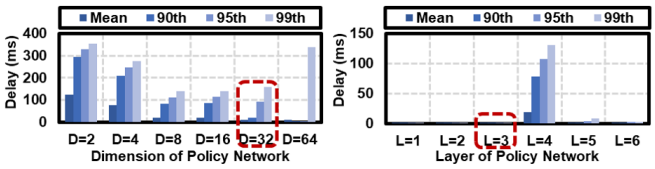


(b) Policy Network

Figure 10: Accumulated reward with different dimension (D) and layer (L) configurations.

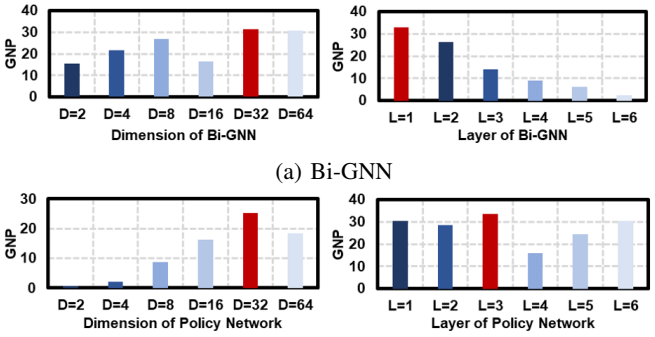


(a) Bi-GNN



(b) Policy Network

Figure 11: Mean response time and tail latency with different dimension (D) and layer (L) configurations.



(a) Bi-GNN

(b) Policy Network

Figure 12: Gross normalized products (GNP) with different dimension (D) and layer (L) configurations.

response time (although there are about 4000 arriving requests). In this scenario, some requests have response time of even over 100 milliseconds as shown in Figure 9(b). However, after multiple training iterations, AlphaR can optimize itself and achieve high reward and low latency. In our simulation, it eventually converges at around the 8000^{th} iteration.

We further examine how the dimension and layer of the proposed Bi-GNN and policy network affect the effectiveness of AlphaR. We train multiple models of AlphaR by building several Bi-GNNs and policy networks with different dimension and layer configurations. The green lines/bars highlight the optimal parameter value. Figure 10 shows the reward under different configurations. In Figure 12 and Figure 11, we also compare the GNP of the system and the response time of requests under different parameter configurations. The results show that the optimal layer of Bi-GNN and policy network is respectively 1 and 3, and their optimal dimension is 32.

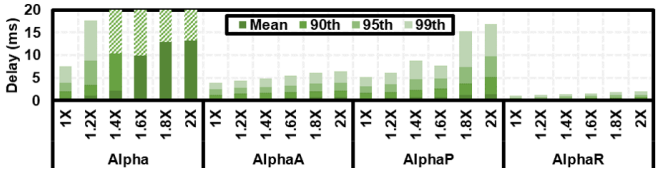
B. Effectiveness of AlphaR

Embracing the scale-out users. It is highly desirable if AlphaR could empower the cloud-native data centers to support more scale-out user demands. In Figure 13(a), we present the mean response time as well as the tail latency of users' requests under different RL-driven schemes. It shows that AlphaR can still respond to the clients' requests within 1 ms even when the user load increases to 2X. Our result shows that AlphaA exhibits 34% less mean delay than *Alpha* implemented with conventional RL algorithms since it can learn more features of microservices by incorporating the bipartite feature inference. Similarly, the mean delay of *AlphaP* is 13% better than *Alpha* since it incorporates attention network. Particularly, AlphaR gives 76% mean delay and 87% 95^{th} percentile latency better than *Alpha*. Figure 13(b) demonstrates that AlphaR can complete more requests within the required time than the other three RL-driven methods. Therefore, AlphaR maximizes the GNP in the cloud-native systems as shown in Figure 13(c).

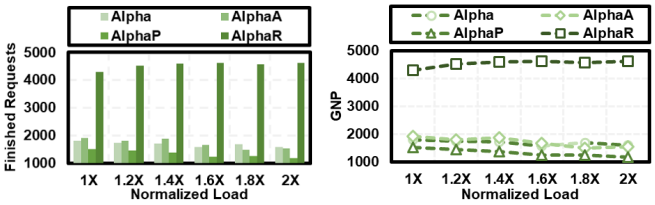
Adapting to the dynamic environment. One important advantage of AlphaR is its ability to handle the dynamicity. To verify this, we show the detailed running GNP from two sides. The first one is to examine how AlphaR reacts to the increasing peak loads. Figure 14(a) shows the GNP of AlphaR at different peak loads arriving at the 200^{th} second. It is evident that AlphaR can adaptively manage resources according to the load variation. We also evaluate the GNP of different RL-driven schemes with the same peak load. In Figure 14(b), the peak load comes at the 100^{th} second and lasts 200 seconds. The figure shows that AlphaR is the only one that can immediately react to load peaks and quickly return to the normal state. As a result, the GNP of AlphaR outperfroms the other three methods since it always makes smart allocation decisions through bipartite feature inference.

C. Comparison with Existing Schemes

Performance and scalablity. We compare AlphaR to existing schemes in terms of the overall performance. As shown in Figure 15, we observe that both mean response time and tail latency for different resource management schemes increase as the user load grows. The response time under AlphaR is much lower than other techniques such as *PartProfile* and *FullProfile*. Although the mean delay of *CurSched* and *FullSched* is also very small at low user load, it grows quickly with the increase of user load. We also compare the GNP of different methods under the six user patterns as depicted in Figure 8. As shown in Figure 16, AlphaR guarantees higher GNP than any others. The reason behind this is that AlphaR always tries to find the optimal allocation for maximizing the efficiency of resources. It well adapts to the unpredictable user behaviors.



(a) Mean Response Time



(b) Fulfilled Request Number (c) Gross Normalized Products

Figure 13: AlphaR empowers cloud-native systems to support more scale-out user demand than existing RLs.

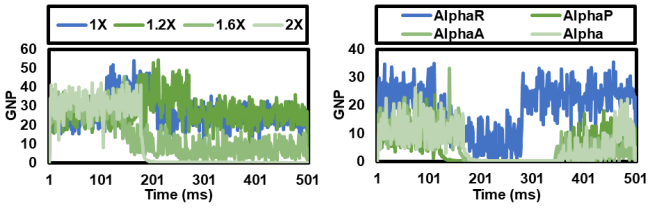


Figure 14: AlphaR can adapt to online load changes: (a) AlphaR's reaction to different peak loads; (b) The reaction of different RL-driven schemes to the same peak load.

Analysis of resource utilization. In this part, we further evaluate the resource utilization of various microservices under different power management schemes. In Figure 17, figures in the leftmost column respectively present the CPU utilization under *FairSched*, the disk bandwidth under *CurSched* and memory usage under *FullSched* from top to bottom. We only present the results for these three results to avoid redundancy since the distribution of resource utilization under these three schemes are the same. This is because they manage resources simply based on the resource demand whilst overlook per-microservice utilization-performance relationship. The last two columns respectively demonstrate the results of *FullProfile* and AlphaR from left to right. *FullProfile* tends to manage different resources in a similar way such as it always allocates CPU, memory and disk to top-tier microservices. Compared to *FullProfile*, AlphaR can always allocate various resources to the most important microservices. Therefore, it can guarantee the best GNP of the systems as shown in Figure 16.

VII. Related Work

Resource management in data centers: Resource management in large-scale systems has been studied for a long time. General-purpose cluster resource management systems [7], [10], [28], [36] assign resources to different incoming jobs to ensure that they are executed as required. They mainly deploy generalized scheduling policy like fair scheduling [7], [25], [36], [37] and packing strategies [14], [28], which would cause low-efficiency hardware resource utilization [13],

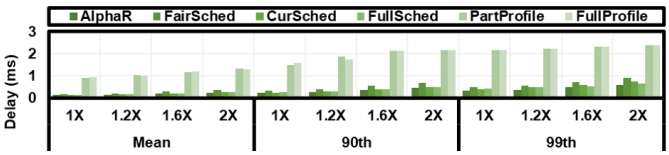


Figure 15: The impact of different resource management schemes on response time with increasing user load.

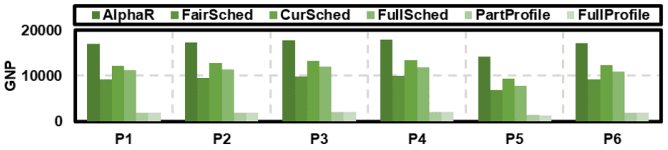


Figure 16: The system's GNP for different resource management schemes under various user patterns.

[32] and poor application performance [21], [23]. Workload-specific schedulers [11], [14], [28] perform better through considering the individual performance characteristic. They assign resources based on pre-defined profiling or plausible heuristics at the initial time, and then tune to reach the expected performance level. These optimizations often deploy some simple, heuristic techniques like feedback control [14], collaborative filtering [11] to collocate several applications.

Cloud-native design and microservice: Many cloud-native design has been proposed [16], [17], [40], [43]. Most prior works emphasize verifying and enhancing the robustness of this software architecture itself [9], [12], [16], [35], [35]. Some proposals focus on optimizing the execution of microservice-based applications. For example, Yu et al. intends to mitigate the performance unpredictability of microservice [33], [34]. Few work consider the resource management for microservices. Chou et al. [14] propose μDPM to manage the power resource of microservices through adapting the energy-saving

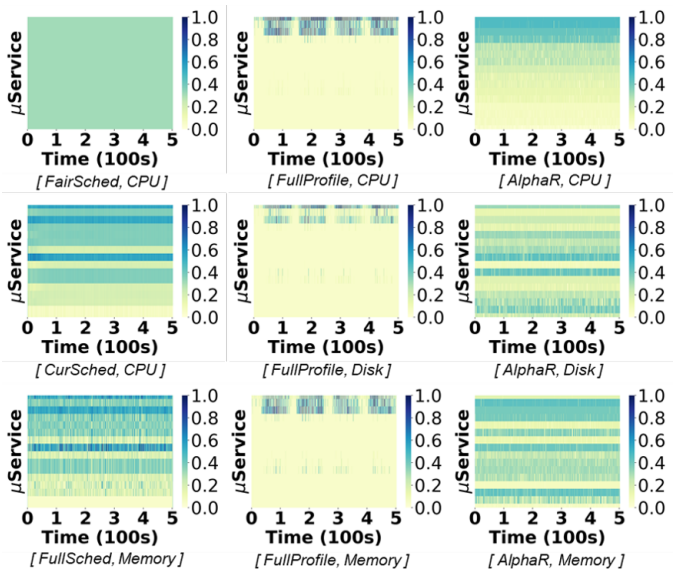


Figure 17: The utilization of CPU, disk and memory by each microservice under different resource management schemes.

mechanisms to prolonging their execution.

Machine learning and resource management: In recent years, ML techniques are widely used to solve complex problems in many fields. Nevertheless, there is little prior work on applying ML techniques to cluster resource management. DeepRM [18] uses RL to train a neural network for multi-dimensional resource packing. However, DeepRM only deals with a basic setting in which each job is a single task. Mirhoseini et al. [8] also uses RL, but relies on recurrent neural networks to scan through all nodes for state embedding, rather than a graph neural network. The objective here is to schedule a single TF job well, and the model cannot generalize to unseen job combinations. Decima proposed by Mao et al. [19] is closely related to our work. It uses RL and GNN to learn scheduling algorithms for short-lived job DAGs. However, Decima, that adopts graph embedding cannot handle the dynamicity of the long-running microservices.

VIII. Conclusion

In this paper, we explore the complexity of resource management brought by the irregular, dynamic and small service in microservice era. To enable efficient resource sharing, we propose AlphaR, a learning-powered resource management policy based on the emerging intelligent techniques. Although the system of AlphaR is not highly interpretable, we have validated that AlphaR can boost the throughput and ensure better latency of the cloud system compared to the conventional resource management approaches. The key novelty of AlphaR such as its graph-based feature inference technique may be applicable to other graph-structured applications. In future work, we will incorporate hardware techniques to improve the stability of our method and further reduce the training cost.

Acknowledgment

We thank all the reviewers and for their valuable comments and suggestions. We also thank our anonymous shepherd for helping improve the paper. This work is supported by the National Key R&D Program of China under Grant 2019YFF0302600. It is also sponsored by the National Natural Science Foundation of China (No. 61972247). Corresponding author is Chao Li from Shanghai Jiao Tong University.

References

[1] "Alibaba cluster data," https://github.com/alibaba/clusterdata, 2019.
[2] "Jaeger," https://www.jaegertracing.io/, 2019.
[3] "Zipkin," https://zipkin.io/, 2019.
[4] Baeldung, "Introduction to dubbo," https://www.baeldung.com/dubbo, 2018.
[5] A. Buck, "Microservices architecture style," https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices, 2019.
[6] A. Clouder, "Capacity planning for alibaba's double 11 shopping festival - alibaba cloud community," https://www.alibabacloud.com/blog/594164.
[7] A. G. et al., "Dominant resource fairness: Fair allocation of multiple resource types," in *NSDI*, 2011.
[8] A. M. et al., "Device placement optimization with reinforcement learning," in *ICML*, 2017.
[9] A. S. et al., "μsuite: A benchmark suite for microservices," in *IISWC*, 2018.
[10] A. V. et al., "Large-scale cluster management at google with borg," in *Eurosys*, 2015.
[11] C. D. et al., "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ASPLOS*, 2013.
[12] C. L. et al., "Power attack defense: Securing battery-backed data centers," in *ISCA*, 2016.
[13] C. L. et al., "Imbalance in the cloud: An analysis on alibaba cluster trace," in *Big Data*, 2017.
[14] C. C. et al., "μdpm: Dynamic power management for the microsecond era," in *HPCA*, 2019.
[15] C. H. et al., "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *HPCA*, 2015.
[16] D. E. et al., "Towards the understanding and evolution of monolithic applications as microservices," in *CLEI*, 2016.
[17] G. K. et al., "Service fabric: a distributed platform for building microservices in the cloud," in *Eurosys*, 2018.
[18] H. M. et al., "Resource management with deep reinforcement learning," in *HotNets*, 2016.
[19] H. M. et al., "Learning scheduling algorithms for data processing clusters," *SIGCOMM*, 2019.
[20] I. A. et al., "Sand: Towards high-performance serverless computing," in *Usenix ATC*, 2018.
[21] J. D. et al., "The tail at scale," in *ACM Communication*, 2013.
[22] J. Z. et al., "Graph neural networks: A review of methods and applications," in *ArXiv*, 2018.
[23] L. B. et al., "The datacenter as a computer: An introduction to the design of warehouse-scale machines, second edition," in *SLCA*, 2013.
[24] L. B. et al., "The datacenter as a computer: Designing warehouse-scale machines, third edition," in *SLCA*, 2018.
[25] M. I. et al., "Quincy: fair scheduling for distributed computing clusters," in *SOSP*, 2009.
[26] P. V. et al., "Graph attention networks," in *ICLR*, 2018.
[27] R. B. et al., "Toward ml-centric cloud platforms," *Communications of the ACM*, 2020.
[28] R. G. et al., "Multi-resource packing for cluster schedulers," in *SIGCOMM*, 2014.
[29] X. H. et al., "Power grab in aggressively provisioned data centers: What is the risk and what can be done about it," in *ICCD*, 2018.
[30] X. H. et al., "Unleashing the scalability potential of power-constrained data center in the microservice era," in *ICPP*, 2019.
[31] X. Z. et al., "Poster: Benchmarking microservice systems for software engineering research," in *ICSE-Companion*, 2018.
[32] Y. C. et al., "Characterizing co-located datacenter workloads: An alibaba case study," in *ArXiv*, 2018.
[33] Y. G. et al., "The architectural implications of cloud microservices," in *CAL*, 2018.
[34] Y. G. et al., "Seer: leveraging big data to navigate the complexity of cloud debugging," in *HotCloud*, 2018.
[35] Y. G. et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ASPLOS*, 2019.
[36] A. Hadoop, "Hadoop fair scheduler," https://hadoop.apache.org/common/docs/stable1/fair_scheduler.html, 2014.
[37] P. Helland, "Cosmos: big data and big challenges," http://research.microsoft.com/en-us/events/fs2011/helland_cosmos_big_data_and_big_challenges.pdf, 2011.
[38] B. Moffat, "Normalized performance ratio - a measure of the degree to which a man-machine interface accomplishes its operational objective," *International Journal of Man-Machine Studies*, 1990.
[39] J. Nielsen, "Usability engineering," 1993.
[40] I. C. R. of Eventuate, "Introduction to microservices," https://www.nginx.com/blog/introduction-to-microservices/, 2015.
[41] M. Richards, "Microservice vs service-oriented architecture," in *O'Reilly Media*, 2015.
[42] A. W. Services, "Microservices on aws," https://docs.aws.amazon.com/aws-technical-content/latest/microservices-on-aws/microservices-on-aws.pdf, 2019.
[43] R. Vettor and S. Smith, "Architecting cloud native .net applications for azure," https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/.
[44] T. Wenisch, "μtune: Auto-tuned threading for oldi microservices," in *OSDI*, 2018.
[45] M. Wu, "Taking the cloud-native approach with microservices," https://cloud.google.com/files/Cloud-native-approach-with-microservices.pdf, 2017.