QoS-awareness of Microservices with Excessive Loads via Inter-Datacenter Scheduling

1st Jiuchen Shi
Shanghai Jiao Tong University
shijiuchen@sjtu.edu.cn

2nd Jiawen Wang
Shanghai Jiao Tong University
wangjiawen0606@sjtu.edu.cn

3rd Kaihua Fu
Shanghai Jiao Tong University
midway@sjtu.edu.cn

4th Quan Chen*
Shanghai Jiao Tong University
chen-quan@cs.sjtu.edu.cn

5th Deze Zeng*
China University of Geosciences
deze@cug.edu.cn

6th Minyi Guo*
Shanghai Jiao Tong University
guo-my@cs.sjtu.edu.cn

2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-8106-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPS53621.2022.00039

***Abstract*—User-facing applications often experience excessive loads and are shifting towards microservice software architecture. While the local datacenter may not have enough resources to host the excessive loads, a reasonable solution is moving some microservices of the applications to remote datacenters. However, it is nontrivial to identify the appropriate migration decision, as the microservices show different characteristics, and the local datacenter also shows different resource contention situations. We therefore propose ELIS, an inter-datacenter scheduling system that ensures the required Quality-of-Service (QoS) of the microservice application with excessive loads, while minimizing the resource usage of the remote datacenter. ELIS comprises a *resource manager* and a *reward-based microservice migrator*. The resource manager finds the near-optimal resource configurations for different microservices to minimize resource usage while ensuring QoS. The microservice migrator migrates some microservices to remote datacenters when local resources cannot afford the excessive loads. Our experimental results show that ELIS ensures the required QoS of user-facing applications at excessive loads. Meanwhile, it reduces overall/remote resource usage by 13.1% and 58.1% on average, respectively.**

I. Introduction

User-facing applications that have stringent Quality-of-Service (QoS) requirements are deployed on the datacenter for high performance and scalability. While popular Internet service providers and Cloud providers often build multiple datacenters in different geographic regions [12], [13], [22], the applications are often deployed on the ones that close to the end-users for short response latency [14], [27], [32].

For user-facing applications, besides the regular diurnal load pattern (the load is low except the peak hours), occasional unpredictable extremely high load of user queries may happen. For instance, excessive queries happen for e-commence service during the online shopping festivals, for social network services when breaking news happens [15], [29]. The computational ability of the datacenter in one region often may not be able to serve the excessive query load, and the user-facing applications often experience severe QoS violations [16], [39].

Adding more servers in the hosting datacenter can resolve the excessive service loads. However, this method significantly

*Quan Chen, Deze Zeng, and Minyi Guo are the corresponding authors.

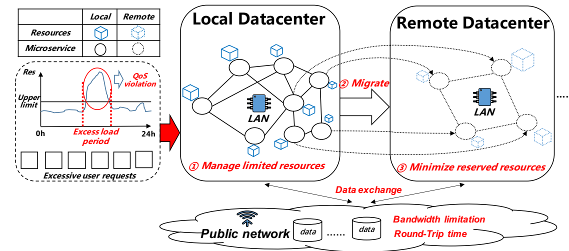


Fig. 1. Deploy microservices across datacenters at excessive loads.

increases the operation cost for the occasional excessive loads. Another solution is leveraging remote datacenters to provide the required computational ability. Especially, many user-facing applications have shifted from monolithic software architecture to microservice architecture [1], [4], [11]. Developed in a microservice architecture, a complex application is implemented by connecting many decoupled micro-services that can be deployed independently and interact with each other through network [19]. It is more resource-efficient to only deploy the necessary microservices on the remote datacenters.

Figure 1 shows an example of using a remote datacenter to support the excessive load of a microservice-based user-facing application. As observed, an efficient inter-datacenter scheduling needs to ① *use the limited resource in the local datacenter the best*, ② *identify the appropriate microservices to be migrated to the remote datacenter*, and ③ *minimize the resource usage in the remote datacenter.* There is some prior work on managing the resource allocation of the microservices for ensuring the QoS of user-facing applications with diurnal load pattern [26], [38], [41]. However, they assume that the local datacenter has sufficient computational power for the user-facing application. In this scenario, the servers are connected with high bandwidth network, and they simply use Kubernetes [7] to place the microservices.

Two new challenges have to be resolved in scheduling microservices across datacenters, compared with the scheduling inside a datacenter. First of all, the bandwidth and latency of the public network between datacenters are much worse than

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00039

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on July 23,2023 at 12:57:11 UTC from IEEE Xplore. Restrictions apply.

those of the network inside a datacenter. The microservice placement matters due to the frequent data communication between microservices. Second, the microservices show different performances (throughput and latency) when they are assigned the same amount of resources (CPU time, memory, etc). It is challenging to determine the microservices that should be migrated to the remote datacenter at excessive local load.

We therefore design **ELIS**, an inter-datacenter scheduling system that ensures the required QoS of user-facing applications while minimizing the resource usage of the remote datacenter at excessive loads. ELIS comprises a *resource manager* and a *reward-based microservice migrator*. For a microservice application to be deployed, the microservices are initially deployed on the local datacenter. When excessive load happens, the resource manager determines the resource adjustment order of the microservices according to their performance status, and uses the Bayesian Optimization (BO) algorithm to minimize resource usage. The resource manager makes the local datacenter support the highest load with the ensured QoS. When the local datacenter is not able to support the load, the microservice migrator migrates some microservices to a remote datacenter. The migrator minimizes the overall resource usage and remote resource reservation on the basis of guaranteeing the QoS and throughput targets. The major contributions of this paper are as follows.

- **Comprehensive analysis of microservice deployment across datacenters under excessive loads.** The analysis motivates us to design an inter-datacenter microservice scheduling system.
- **The design of BO-based resource management policy on the local datacenter.** The policy recycles resources from some microservices and reallocates them to the microservices that tend to result in the QoS violation. It maximizes the supported peak load of the local datacenter.
- **The design of a method to search for the to-be-migrated microservices.** We identify two guidelines with which we can determine the to-be-migrated microservices that reduce the tail latency of user-facing applications the most at excessive load.

We evaluate ELIS on an emulated geo-distributed datacenter environment. The experimental results show that ELIS reduces the overall resource usage by 13.1% and remote resource reservation by 58.1%, while ensuring the required QoS at excessive load, compared with state-of-the-art work.

II. Related Work

Prior related work mainly falls into three categories: task scheduling within datacenters, task scheduling across datacenters, and managing resources for microservice architecture.

A. Task Scheduling within Datacenters

In the aspect of task scheduling inside the datacenter, PARTIES [14], CLITE [32] and Twig [31] co-located latency-critical (LC) services with batch applications and maximized resource utilization while meeting the QoS targets for LC services through resource partitioning. Rhythm [41] quantified the interference tolerance abilities of different stages of an LC service, and deployed ones with higher anti-interference ability along with more batch jobs to improve resource efficiency. These works do not apply to microservice applications whose stages have frequent data interaction. Moreover, they did not consider the public network effect for resource management when the application is deployed geographically distributed.

B. Task Scheduling across Datacenters

In terms of geo-distributed task scheduling, Long et al. [28] formulated the task scheduling problem in a collaborative cloud-edge environment as a non-cooperative game. Some efforts focused on mitigating performance bottlenecks caused by limited public network bandwidth between datacenters. Gaia [23] dynamically eliminated insignificant communication between datacenters while maintaining the correctness of an ML algorithm. Hung et al. [25] reordered the task execution order on the same datacenter to minimize the global job completion time across datacenters. Yugong [24] saved the precious wide area network bandwidth through project placement, table replication and job outsourcing. However, none of these works took into account the excessive load scenario with limited resources of the local datacenter, which is desired to minimize the resource usage and remote resource reservation.

C. Managing Resources for Microservices

Wisp [37] and DAGOR [42] designed load adjustment mechanisms by setting different priorities to balance microservices. FIRM [33] identified the critical microservices that cause SLO violations and dynamically re-provisioned resources to avoid SLO violations. Many other works contributed to the resource management of microservices using machine learning or heuristic methods [18], [20], [21], [38], [40]. These researches did not consider deploying microservices across datacenters. Nautilus [17] focused on the microservice deployment in the public network. However, it did not consider how to maximize the utilization of the local datacenter's resources while minimizing remote resource reservation. When adapting to our scenario of excessive loads, it has low resource efficiency.

III. Motivation

In this section, we investigate the problem of QoS violation due to the poor resource management in a datacenter, and show the factors that impact the response latency when a microservice application is geographically distributed.

A. Investigation Setup

We use three identical servers to emulate two datacenters. One server emulates the local datacenter, and the other two servers are used to be the remote datacenter. To emulate the public network between datacenters, we use the network tool *tc* to limit the bandwidth of 200mbit/s and the Round-Trip Time (RTT) of 10ms between the datacenters. The public network bandwidth and the RTT are the results we profile of two virtual machines on two regions of a popular public Cloud.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on July 23,2023 at 12:57:11 UTC from IEEE Xplore. Restrictions apply.

TABLE I
EXPERIMENT SPECIFICATIONS

	Specifications
Hardware	Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz 20 physical cores, 256GB Memory Capacity 50 MiB L3 Cache Size (20-way set associative) 200mbit/s public network bandwidth, 10ms RTT
Software	Ubuntu 20.04.2 LTS with kernel 5.11.0-34-generic Docker version 20.10.8, Kubernetes version v1.20.4
Benchmarks	DeathStarBench: SocialNetwork (SN), MediaService (MS), HotelReservation (HR)

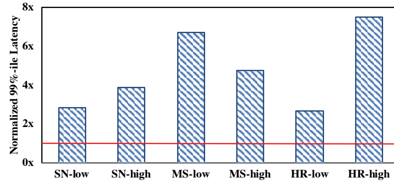


Fig. 2. The 99%-ile latencies of the benchmarks with Kubernetes. The red line is the normalized 99%-ile QoS target of the benchmarks, whose Y value is “1x”.

We use three open-sourced microservice benchmarks SocialNetwork (SN), MediaService (MS), and HotelReservation (HR) in the DeathStarBench [19] to perform our experiments. Each benchmark in DeathStarBench has multiple microservices of different characteristics [19]. When the load of a service increases, its compute-intensive microservices (e.g., compose-review) often require more CPU time and LLC space, while backend database services (e.g., memcached) require more memory and IO bandwidth.

The detailed hardware, software configurations and benchmarks are shown in Table I.

B. Resource Inefficiency in Local Datacenter

This experiment investigates whether the current microservice management system can effectively utilize the resources in the local datacenter. Kubernetes, one of the most popular scheduling frameworks [17], [33], [41], is used to manage the microservices. We measure the tail latencies of the benchmarks at low load and high load. We use 70% and 90% of the peak supported load of the local server if the resource allocation is optimized to be the low load and the high load respectively.

Fig. 2 shows the normalized 99%-ile latency of the benchmarks with Kubernetes. As shown, the benchmarks experience QoS violations at both the low load and the high load. The QoS violation originates from the poor resource allocation. Take the benchmark *HR* at high load as an example. Fig. 3 shows the CPU time usage of the microservices with Kubernetes normalized to the case with the hand-tuned CPU time allocation that supports the highest load. The hand-tuned CPU allocation ensures the required QoS target of the benchmark. As observed, some of the microservices use smaller CPU time while other microservices use larger CPU time.

The resource contention with Kubernetes does not use the computational resource efficiently, and results in the QoS violation even at a low load.

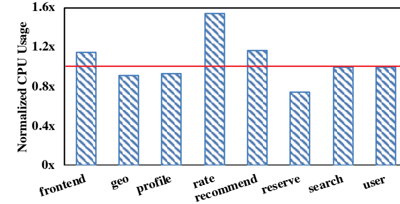


Fig. 3. The normalized CPU allocation relative to the hand-tuned CPU allocation of *HR* at high load with Kubernetes. The red line is the normalized hand-tuned CPU allocation for each microservice which can ensure the QoS, whose Y value is “1x”.

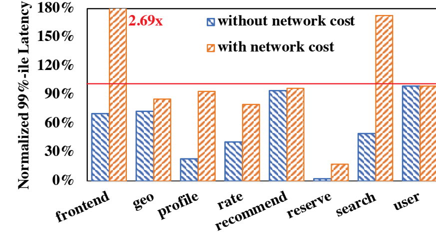


Fig. 4. The normalized 99%-ile latencies of *HR* at excessive load if a microservice is migrated to the remote datacenter. The latency is normalized to the case when all the microservices are in the local datacenter.

C. Problems of Migrating Microservices

When the local datacenter cannot afford an excessive load, some microservices should be migrated to remote datacenters for ensuring the QoS. In this subsection, we investigate the factors that determine the end-to-end latency of a user-facing application when some of its microservices are deployed in the remote datacenter. In this experiment, we use *HR* as the example to show the factors that impact the response latency of a user-facing application when some of its microservices are migrated to the remote datacenter. The load of *HR* in this experiment is $1.5X$ of the peak supported load of the local datacenter, if the resources are carefully allocated. We report the 99%-ile latency of the benchmark in this experiment.

Figure 4 shows the normalized 99%-ile latencies of the benchmark if a microservice is migrated to the remote datacenter. In the figure, the x-axis shows the migrated microservice. In the experiment, the bar “with network cost” shows the case that the bandwidth between the datacenters is limited to be 200Mbps and the RTT is 10ms, and the bar “without network cost” shows the case that the public network bandwidth is the same as the local network bandwidth and has no RTT.

As observed from Figure 4, *HR* achieves shorter 99%-ile latency when a microservice is migrated to a remote datacenter. The microservice is able to obtain more computational resources from the remote datacenter. However, when the microservice *frontend* or *search* is migrated to the remote datacenter, the 99%-ile latency of *HR* actually increases significantly. If *frontend* is migrated, the normalized 99%-ile latency of *HR* is 2.69, although more computational resources is obtained. After looking into the communication topology of the microservices, we find that *frontend* and *search* needs to

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on July 23,2023 at 12:57:11 UTC from IEEE Xplore. Restrictions apply.

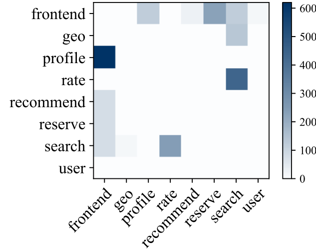


Fig. 5. The bytes of data transferred between microservices for a single query in *HR*. The x-axis is the source, and the y-axis is the destination.

transfer a large volume of data to or from other microservices. In this case, the slow public network results in the large data communication overhead, and the overhead results in the longer response latency.

In more detail, Fig. 5 shows the size of data transferred between microservices in *HR*. As shown in this figure, *frontend* and *search* communicate with the most microservices (*frontend* communicates with four microservices and *search* communicates with three microservices). In this case, if *frontend* is migrated, the communication overhead is large.

The case "without network cost" in Figure 4 reveals the normalized 99%-ile latency of *HR* if the communication overhead is eliminated. Observed from this result, the serious latency increase due to the migration of *frontend* or *search* is eliminated. However, migrating different microservices show different impacts on reducing the response latency. For instance, migrating *reverse* is able to reduce the response latency by 97.1%, while migrating *user* has a negligible effect on reducing the response latency. This is because the microservices have different sensitivities to the insufficient resource at excessive load. Looking into the resource usage of the experiment, the microservices mainly contend for the CPU resource, the *reserve* microservice is CPU-intensive, while *user* is not. It is more profitable to migrate microservices that are sensitive to the current stressed resources.

Therefore, when a microservice-based user-facing application suffers from an excessive load, it is necessary to carefully determine the microservices that should be migrated to the remote datacenter. The decision should be made based on the characteristics of the microservices, the contention situation on the local datacenter, and the public network situation.

IV. ELIS Methodology

Based on the above observations, we design and implement **ELIS**, an inter-datacenter scheduling system that ensures the required QoS of user-facing applications, while minimizing the resource usage of remote datacenter at excessive loads.

Fig. 6 shows the overview of ELIS, which comprises a *resource manager* for resource adjustment and a *reward-based microservice migrator* to migrate microservices between local and remote datacenters. The resource manager allocates each microservice "just-enough" resources (e.g., core time, shared cache ways, memory space, and network bandwidth), so that the local capacitated datacenter is able to maximize

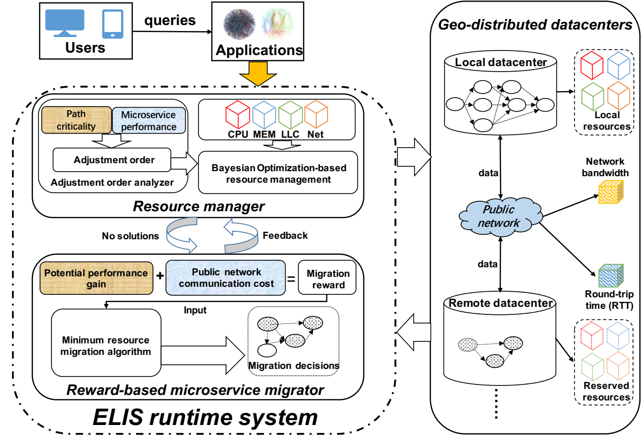


Fig. 6. Design overview of ELIS.

the peak achievable throughput of the local datacenter without QoS violations. When a user-facing application suffers from an excessive load (i.e., the load that higher than the "peak achievable throughput"), the microservice migrator selects and migrates some microservices to the remote datacenter for ensuring the required QoS. Specifically, ELIS manages the deployment of a microservice-based user-facing application s in three steps.

(1) ELIS monitors the load of the service s periodically. Based on the loads in the last interval, the resource manager adjusts the amount of resources allocated to each microservice. While the microservices on the same datacenter tend to contend for the resources if the load is relatively high, the challenging parts here are identifying the critical path in the microservice graph, and reallocating some resources to the microservices on the critical path from other microservices, without making them become a new performance bottleneck. A Bayesian Optimization based method is leveraged to handle the above complex contention situation (Section V).

(2) If the resource manager finds that the local datacenter is not able to host the excessive load, ELIS migrates some microservices to the remote datacenter. As the effect of migrating a microservice is impacted by many factors (discussed in Section III-C), the challenging part here is to determine the appropriate microservices to the remote datacenter. Therefore, we define a migration reward to quantify the benefit of migrating a microservice. The reward of migrating a microservice is determined by the potential performance gain and the pubic network communication cost (Section VI).

(3) If the load of s drops, ELIS migrates some microservices back to local, in order to eliminate the long communication overhead and reduce the remote resource usage.

It is worth noting that ELIS does not need to modify the source code of microservice applications, and can be implemented as a plug-in based on Kubernetes.

V. Resource Manager

In the section, we present a resource manager to maximize the supported throughput of the local datacenter, where

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on July 23,2023 at 12:57:11 UTC from IEEE Xplore. Restrictions apply.

Bayesian optimization is leveraged to adjust the resource allocation for each microservice. Given the limited resource capacity in the local datacenter, it is essential for the resource manager to 1) minimize the resource usage while ensuring the QoS target in *normal* loads, and 2) efficiently utilize resources to achieve as higher peak throughput as possible in *excessive* loads.

A. Microserivce Dependency Graph Abstraction

The request flow passing through the microservices can be abstracted into a directed acyclic graph (DAG) $G = (V, E)$ as shown in top left part of Fig. 7. The vertices V represent the microservices, the edges E represent the communications between microservices (i.e., dependency). Both the vertices and the edges are weighted, referring to the serving time of the corresponding microservice and the data transmission time between corresponding microservices, respectively. Note that there may be multiple paths from the source vertex to the destination vertex. Different paths may have different vertices, representing different microservices. The path with the highest weight sum (i.e., the maximum sum of serving time and data transmission time) is defined as the *critical path* of the corresponding application [33]. The latency sum of the *critical path* is regarded as the overall latency of the application. The data transmission time between dependant microservices in different datacenters imposes a high influence on the application's performance and hence the microservice deployment and resource allocation decisions should be carefully made, especially in the case with an excessive load exceeding the capacity of the local datacenter.

B. Resource Adjustment Order

Considering the capacity limitations of the local datacenter, it is highly desirable to give each microservice "just-enough" resources, so as to achieve as high QoS as possible. To allocate adequate resources to an application, previous works advocated searching all resource dimensions of all microservices together [17]. However, this incurs a huge search space and it is non-trivial to achieve an acceptable solution. Instead of treating an application as a whole, we take a fine-grain approach and allocate resources for each constituent microservice individually first and then adjust the resource allocation to achieve the QoS target by analyzing each microservice's execution characteristics, i.e., the 99%-ile latency. During such process, obviously the resource adjustment order of the microservices is essential to the eventual overall performance as the later adjusted microservice can only use the residual resources. As illustrated in Fig. 7(a), we propose a two-step microservice ordering strategy for resource adjustment.

The first step is to analyze the criticality of each path of a microservice application. Prior works have shown that the microservices on the critical path (CP) are relatively important [33], [41], and intuitively higher priority should be given to them to mitigate possible QoS violations. However, note that the core of resource adjustment is to compensate the microservices with not-enough resources by releasing some

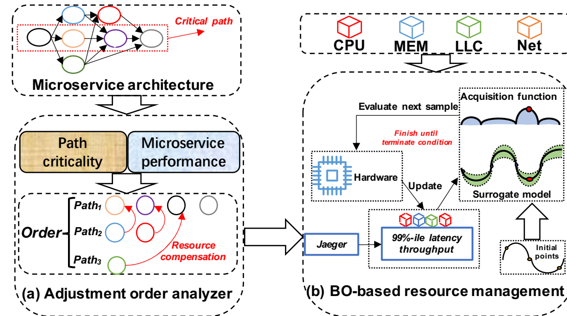


Fig. 7. The resource manager that manages local datacenter.

resources from the over-provisioned ones. The microservices on the critical path are less like to be over-provisioned and therefore we give higher priority to the microservices that are not on the critical path. Consequently, we sort the paths in ascending order by the execution time profiled in the last time interval, and first adjust the resources of the microservices on the shortest path.

The second step is to analyze the criticality of different microservices on the same path. For the microservices on the same path, we similarly should give higher resource adjustment priority to the microservices with better performance, in the expectation that more resources could be released and compensated to the microservices with poor performance. Specially, we use the ratio of 99%-ile latency and 50%-ile lantecy L_{99}/L_{50} [33] to assess the microservice performance, and also sort them in an ascending order.

After the above two steps, we finally get a resource adjustment order of all the constituent microservices of an application. Thereafter, the next critical thing is how to adjust the resource allocation to achieve our goal of minimizing the resource usage while ensuring the QoS.

C. Bayesian Optimization based Resource Adjustment

The core of resource adjustment is to make each microservice allocated with "just-enough" resources to achieve its QoS target. Accordingly, the resource manager will adjust the resources following the two principles: (1) For a microservice that meets the QoS target, the resource manager will try to lower the resource allocation quota without lowering the QoS, in order to release some resources for compensation. (2) For a microservice with QoS violation, the resource manager will try to compensate it by allocating more resources within the available capacity of the local datacenter. Essentially, the resource adjustment of each microservice can be described as an optimization problem that *minimizing the resource usage on the premise of QoS guaranteeing*.

Unfortunately, it is non-trivial to solve such an optimization problem at runtime. Firstly, we do not, and usually it is not practical to, have any prior knowledge on the achievable performance of a microservice for a given resource allocation during runtime. Secondly, a resource allocation actually refers to a multi-dimensional resource configuration, which together constitutes a large multi-dimensional search space. We notice

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on July 23,2023 at 12:57:11 UTC from IEEE Xplore. Restrictions apply.

that Bayesian Optimization (BO) is a promising solution to address the above two challenges. Firstly, BO does not require any prior knowledge about the objective function. As we have known, the relation between a resource configuration and the achievable performance is a black-box function. Secondly, BO is able to find a near-optimal solution with a limited number of resource configuration samples [32], while other black-box optimization methods, such as deep neural networks and reinforcement learning, require lots of training data and are not suitable to apply at runtime. Thus, we explore the BO algorithm to design our resource adjustment strategy. Moreover, to address the curse-of-dimension in microservice applications, ELIS trains an individual BO model for each microservice to find its near-optimal resource allocations. This is different from traditional works [32], [34] using a uniform BO model for all the co-located workloads and is reasonable for long-running services [33].

The BO algorithm first fits a surrogate model using an initial sample set, where the surrogate model models the resource adjustment optimization problem. Then it repeatedly selects the next point to sample and updates the model according to the newly sampled data. Therefore, we should first choose a surrogate model to model the resource adjustment optimization problem as the objective function. In ELIS, we adopt Gaussian Process (GP) as the surrogate model, which is widely used and robust to noise and fluctuations of sampling. In order to evaluate the objective function, we design a normalized *score function* that assigns scores to the objection function evaluation. This function scores the performance (99%-ile latency and throughput) of the microservice under the corresponding resource configurations (CPU core time, memory space, shared cache ways, and network bandwidth) at the end of each time interval, and is defined as follows,

$$Score = \begin{cases} \frac{1}{2} \times \sqrt{QoS_{rate} \times Thr_{rate}}, \text{if QoS/tpt violates,} \\ \\ \frac{1}{2} + \frac{1}{2} \times \sqrt[4]{\prod_{i=1}^{4} (1 - RU_i)}, \text{otherwise.} \end{cases} \quad (1)$$

QoS_{rate} measures the degree that a microservice meets its QoS target with $QoS_{rate} = \max(QoS_{target}/QoS_{eval}, 1.0)$, and is set to 1 if the QoS target is exactly met. Thr_{rate} measures the degree that the microservice meets with its throughput target with $Thr_{rate} = max(Thr_{eval}/Thr_{target}, 1.0)$, and is set to 1 if the throughput target is met. RU_i represents the utilization of resource i relative to its total capacity in the local datacenter with $RU_i = Res_i^{used}/Res_i^{tot}$. This score function ranges from 0 to 1, which guides us to search in the right direction in the searching space. If either the QoS or throughput targets is not satisfied, the function value is between 0-0.5. A higher score implies a higher degree of satisfaction. Otherwise, the function value is between 0.5-1. In this case, we should intend to minimize resource usage after ensuring QoS and throughput targets.

In addition, an acquisition function is needed to select the next point to sample. We adopt Expected Improvement (EI) function, which proves to converge fast and is effective for complex jobs [36]. As shown in Fig. 7 (b), for each time interval, the acquisition function determines which configuration point is the next to be sampled. We sample the corresponding performance metrics (the 99%-ile latency and throughput from Jaeger [5]) under the selected configuration point. We then use Eq. (1) to evaluate this configuration point, and use this new sample to update the surrogate model. This process iteratively proceeds until the EI is converged, e.g., smaller than 10^{-5} in our implementation.

VI. REWARD-BASED MICROSERVICE MIGRATION

Occasionally, the local datacenter may fail to provision enough resources to achieve the desired performance when there are excessive loads. In this section, we introduce a reward-based microservice migrator to deal with the excessive user requests of the local datacenter. The migrator can migrate some microservices instances to the remote datacenter to ease the burden on local datacenter. We first take microservice potential performance gain and public network communication cost into account and define the $reward$ of microservice migration. Then, we propose a minimal resource migration strategy to migrate the microservices to pursue the goal that the resource usage of both the application and the remote resource reservation is minimized.

A. Definition of the Migration Reward

Through the preliminary evaluation, we derive the migration reward following two guidelines:

(1) Migrating the microservice with higher potential performance gain is potential to improve the overall performance. When the excessive workload comes, the local datacenter will be short of different resources, e.g., CPU or LLC. Different microservices have different sensitivities to the resource shortage because of diverse resource demand characteristics, resulting in different performance gains after migrating to the remote datacenters with sufficient resources. Prior works have shown that the microservices with higher latency and latency variability exhibit higher potential to improve the overall performance [33], [41]. Therefore, for each microservice to be migrated, we use the product of staged latency percentage (relative to the total latency) and Coefficient of Variance (CV) of the latency to derive the potential performance gain of each microservice as follows,

$$Perf_k = Pct_k \times CV_k = \frac{\overline{L_k}}{\overline{L_{tot}}} \times \frac{1}{\overline{L_k}} \sqrt{\frac{1}{n} \times \sum_{i=1}^{n} (L_k^i - \overline{L_k})^2}, \quad (2)$$

where $\overline{L_k}$ and $\overline{L_{tot}}$ represent the average latency of microservice k and overall latency of the application during the last time interval, respectively, n is the number of queries for microservice k in the last time interval, L_k^i represents each query's latency, and $\overline{L_k} = \frac{1}{n} \sum_{i=1}^{n} L_k^i$.

(2) Migrating the microservice that results in less public network communication cost is beneficial to the overall performance. The network cost from the public network can harm the QoS, and it is mainly from two aspects: the bandwidth limitation and RTT.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on July 23,2023 at 12:57:11 UTC from IEEE Xplore. Restrictions apply.

Nowadays cloud giants provide the public network bandwidth up to 200mbit/s [3], which is not sufficient if there is too much traffic in the public network, leading to long transmission time between dependant microservices. Therefore, we derive the network cost caused by network bandwidth limitation as the data transmission time increment for the queries in each second after migrating microservice k as follows,

$$cost_band_k = \sum_{(i,j)\in pub} \left(\frac{dv_{i,j}}{bw_pub_{i,j}} - \frac{dv_{i,j}}{bw_ori_{i,j}}\right) \times qps \tag{3}$$

where qps denotes the queries per second received, $dv_{i,j}$ is the data transmission volume from microservice i to microservice j, and $bw_ori_{i,j}$ and $bw_pub_{i,j}$ represent the network bandwidth usage from microservices i to j when they communicate with each other not through and through public network, respectively. Constraint $(i, j) \in pub$ means the data communication between microservices i and j is through the public network after migrating the corresponding microservice. The network bandwidth usage is determined by the resource manager in Sections V. If multiple microservice pairs communicate through public network and their required bandwidth sum is higher than the highest available public network bandwidth, they will share the total public network bandwidth proportionally. For instance, if i transfers $dv_{i,j}$ to j and x transfers $dv_{x,y}$ to y through public network with the total bandwidth b, the bandwidth between i and j is calculated by $\frac{dv_{i,j}}{dv_{i,j}+dv_{x,y}} \times b$, and the remaining part is allocated between x and y.

On the other hand, the RTT between the datacenters in different regions can be ranged from several milliseconds to hundreds of milliseconds [30]. The RTT between different geographical locations can also affect the application performance. Thus, we also derive the network cost caused by RTT as the latency increment for the query after migrating microservice k as follows,

$$cost_rtt_k = e_k \times rtt \tag{4}$$

where rtt represents the RTT between the local and the remote datacenters, and e_k represents the number of edges (defined in Section V-A) of the microservice application DAG that are in the public network after migrating microservice k.

Thus, we can define the reward of migrating microservice k as

$$reward_k = \frac{Perf_k}{cost_net_k}, \tag{5}$$

where

$$cost_net_k = cost_band_k + cost_rtt_k \tag{6}$$

is the total network communication cost.

To measure the network communication cost, we profiled the data transmission volume between microservices offline (reasonable for long-running services). Moreover, the real-time public network bandwidth and the RTT are periodically measured via monitoring tools, e.g., kubenurse [6] and Prometheus [8].

From Eq. (5), we can see that the reward could be interpreted as the *the potential performance gain per unit of network cost*. Since the public network overhead needs to be compensated with more computing resources, i.e., reducing the serving time to compensate the data transmission time for achieving the same QoS target, we prioritize migrating the microservice with a higher $reward$. According to the reward definition, higher $reward$ implies that the microservice has greater potential performance gain with the same amount of remote resources, and smaller data communication overhead requiring computation-based compensation.

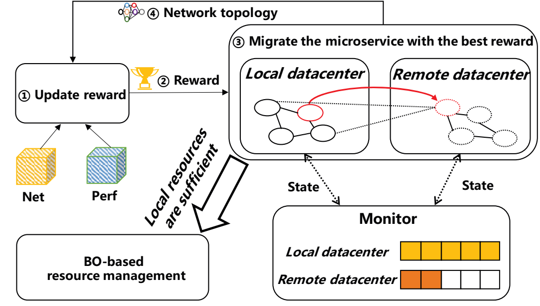


Fig. 8. Minimum resource migration strategy.

B. Minimum Resource Migration Strategy

As it is desirable to reserve remote computing resources as little as possible, we design a minimum resource migration strategy, which can migrate some microservices to the remote datacenter for alleviating the excessive loads, while minimizing the resource usage of the remote datacenter.

As shown in Fig. 8, the monitor collects the resource usage state of local and remote datacenters periodically. When the monitor identifies that the local datacenter faces the excessive load, we first calculate the $rewards$ of migrating each microservice based on Eq. (5), as shown in the steps of ①-②. Each time, we migrate the microservice with the highest $reward$, and then update other microservices' $rewards$ again because of the change of microservice placement in both the local and remote datacenters, as shown in the steps of ③-④. The migration decisions are iteratively made until the local resources are sufficient to afford the remaining microservices. Each microservice's resource usage is determined by the resource manager in Section V. After the migration process, we use the BO-based resource manager in Section V to adjust the resource quotas of each microservice again, to minimize the resource usage while guaranteeing the QoS.

When the degree of excessive load reduces, the monitor can be aware that the local datacenter may have surplus resources to accommodate more microservices locally. To minimize the remote resource reservation, some microservices could be migrated back to the local datacenter. ELIS uses a lazy migration back strategy to eliminate Ping-Pong microservice migrations. Only when the load of the entire service decreases to a certain extent (e.g., 90% of the maximum local achievable throughput) and stabilizes for a period (e.g., 10 minutes), we migrate back some microservices. The migration order is the reverse order of the one from the local datacenter to the

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on July 23,2023 at 12:57:11 UTC from IEEE Xplore. Restrictions apply.

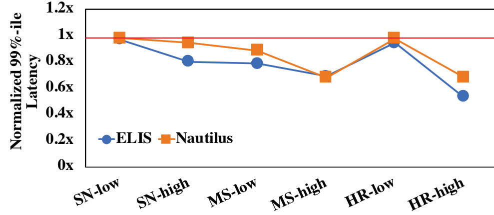


Fig. 9. The 99%-ile latency of benchmarks with ELIS and Nautilus.

remote datacenter. Take the microservices of HR benchmark as an example, if the migration order from local to remote is "reserve", "profile", and "frontend", the migration back order will be "frontend", "profile", and "reserve". The process of migrating back is terminated until the local datacenter cannot afford any other microservices in the remote datacenter or all the microservices have been migrated back.

VII. Evaluation

In this section, we report our performance evaluation on the effectiveness of efficiency of ELIS for minimizing the total and remote reserved resource usage of microservice applications, as well as the achieved QoS and throughput.

A. Evaluation Setup

If not specifically pointed out, we use the experimental configurations and testing benchmarks in Table I for our experiments. ELIS does not depend on any specific software and hardware, and can be easily set up on the local and remote datacenters. In our experiments, we use load generator wrk2 [9] to generate different loads (queries per second) to the user-facing application. The performance statistics are collected from Jaeger [5] during each time interval, and ELIS uses the statistics to make the next scheduling decision.

We compare ELIS with a state-of-the-art microservice deployment framework Nautilus [17]. We adapt Nautilus to the local and remote datacenter environment with excessive loads. Specifically, we implement the microservice migrator of Nautilus that migrates microservices based on the load of the nodes. Since Nautilus's resource manager takes all microservices' configurations as input for its reinforcement learning agent, it has the scalability issue and large training overhead in our local and remote environment with excessive loads. Therefore, for fair comparisons, we use our BO-based resource manager for Nautilus.

Moreover, to show the effectiveness of our resource manager for a local datacenter with limited local resources, we compare ELIS with FIRM [33] that only tunes the resource allocation of the microservices. As FIRM assumes unlimited resources, we adapt FIRM by: (1) finding the critical path, (2) determining the critical microservices, and (3) increasing their resources to find the optimal resource allocations. The optimal allocation is determined in a brute force way.

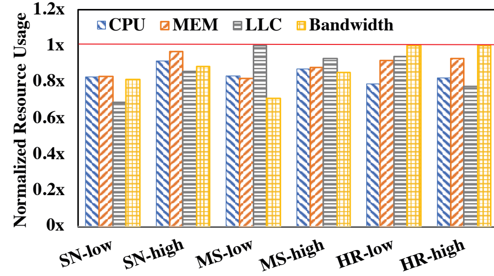


Fig. 10. The normalized total resource usage of ELIS to Nautilus.

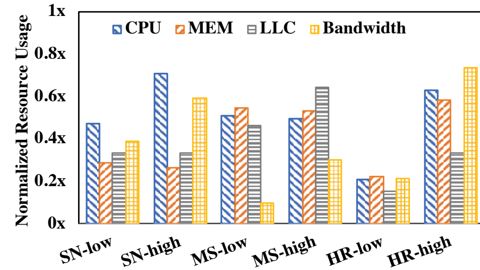


Fig. 11. The normalized remote resource reservation of ELIS to Nautilus.

B. Minimizing the Resource Usage while Guaranteeing QoS

In this subsection, we compare ELIS with Nautilus in minimizing the resource usage while guaranteeing the QoS, when the local resources cannot afford the excessive loads. The resource usage includes the overall resource usage of the application and the resource usage in the remote datacenter.

Fig. 9 shows the 99%-ile latency of the benchmarks in 6 scenarios with excessive loads (3 benchmarks under low excessive load and high excessive load). The low/high excessive load is defined to be $1.2X$ and $2X$ of the peak supported load of the local datacenter with ELIS. As ELIS and Nautilus both migrate some microservices to the remote datacenter to alleviate the resource insufficiency of local datacenter with excessive loads, both of them are able to guarantee the QoS.

Fig. 10 shows the total resource usage of the benchmarks in different load levels. In this figure, we show the four kinds of resource usage of ELIS, i.e., CPU, MEM, LLC, and network bandwidth, and each kind of resource is normalized to the usage amount of Nautilus. We can observe that the resource usage with ELIS is smaller than the resource usage with Nautilus for all the three benchmarks. Compared with Nautilus, ELIS reduces the CPU usage, MEM usage, LLC usage, and the network bandwidth usage by 15.7%, 11.0%, 13.4%, and 12.4%, respectively.

In addition, Fig. 11 shows the resource usage in the remote datacenter with ELIS normalized to resource usage in the remote datacenter with Nautilus. Observed from this figure, the CPU, memory, LLC, and network bandwidth usage in the remote datacenter is reduced by 49.8%, 59.4%,62.4%, and 60.8%, respectively. Since the resources in the local datacenter are packet periodic, while the resources in the remote datacenter are used on demand when local resources

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on July 23,2023 at 12:57:11 UTC from IEEE Xplore. Restrictions apply.

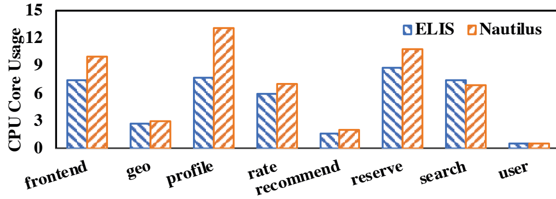


Fig. 12. CPU usage of all microservices with Nautilus and ELIS of HR benchmark in high excessive load.

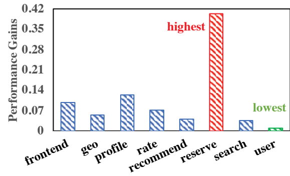


Fig. 13. Potential performance gains of microserives at high excessive load of HR benchmark.

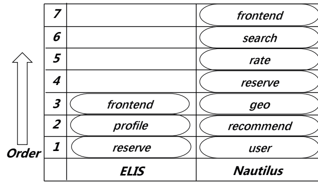


Fig. 14. Microservice migration order of ELIS and Nautilus for HR benchmark at high excessive load.

are insufficient, minimizing the resource usage in the remote datacenter can reduce the cost of cloud service providers during the excessive load time.

We also build ELIS with two Cloud Elastic Servers [2] in two different regions of Alibaba Cloud. The two cloud servers represent the local and remote datecenters, respectively. The local/remote datacenters have 24/48GB memory capacity, and Intel(R) Xeon(R) Platinum 8369B CPU @ 2.9GHz with 12/24 logical cores, respectively. The public network peak bandwidth of the two servers are 200mbit/s, and the RTT between them is about 10ms. Since LLC cannot be adjusted in the cloud virtual machines, we only adjust CPU, memory, and network bandwidth in this experiment. We test the HR benchmark with the same two qps of Fig. 9, and the results show that: Compared with Nautilus, ELIS reduces the CPU usage, MEM usage, and the network bandwidth usage by 19.6%, 7.2%, and 15.7%, respectively. Moreover, the CPU, memory, and network bandwidth usage in the remote datacenter is reduced by 34.4%, 42.2%, and 54.9%, respectively.

ELIS reduces the overall resource usage and resource usage of the remote datacenter compared with Nautilus.

C. Diving into the Microserivice Migrator

In this subsection, we use an experiment with the benchmark *HR* at a high excessive load to better understand the reasons that ELIS reduces the resource usage compared with Nautilus. Fig. 12 shows the CPU cores allocated to each microservice of *HR* at the high excessive load.

As observed, ELIS reduces the resource usage of almost all the microservices. Two main reasons result in the resource reduction. Firstly, ELIS comprehensively considers the data transmission volume, network bandwidth limitation, and the Round-Trip Time (RTT) when deploying the microservices, while Nautilus only considers the data transmission volume. This leads to better optimization of the network cost. As the network cost can cause longer data transmission time, less computing time is acquired to achieve the QoS, so that Nautilus requires more CPU resources than ELIS. Secondly, ELIS prioritizes migrating the microservices with greater potential performance gain. Under the same quota of remote resources, ELIS's migrated microservices can reduce more latency than Nautilus. While Nautilus does not take potential performance gain into account, it requires more resources to achieve the same QoS target with ELIS.

Corresponding to Fig. 12, Fig. 13 shows the potential performance gain (calculated by Eq. (2)) of all the microservices.The results show that *reserve* microservice has the largest value and *profile* is the second largest, while *user* and *recommend* have the smallest values. Fig. 14 shows the migration order of ELIS and Nautilus during excessive load time, respectively. We can observe that ELIS prioritizes migrating the microservices with greater potential performance gains, allowing the application to better reduce overall latency, even though these microservices take up some remote resources. Later, for the joint consideration of network cost and potential performance gain, ELIS migrates a few other microservices and then restores to the QoS target.

On the contrary, Nautilus prefers to migrate the microservices with the least amount of data transmission (shown in Fig. 5), i.e., *user*, *recommendation*, and *geo*. However, these microservices provide little performance gains. They use heavy remote resources but only reduce little overall latency. Nautilus then has to migrate other microservices until the microservices with higher potential performance gains (i.e., most of the microservices ELIS migrates at the beginning), and finally restores the QoS target.

ELIS avoids migrating the redundant microservices with little potential performance gains to the remote datacenters, so that can reduce the remote resource reservation.

D. Effectiveness of the Resource Manager

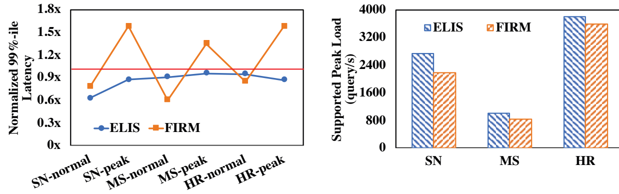
The resource manager monitors load and latency changes, and adjusts the resource allocations for the microservices. In this subsection, we compare ELIS's resource manager with FIRM, to verify the effectiveness of ELIS's resource manager on local resource utilization efficiency.

In this experiment, we deploy all the microservices in the local datacenter, then assign the same initial resource configurations to all the microservices of ELIS and FIRM, and compare them at normal and peak loads, respectively. Normal load refers to the qps that local resources can easily support, and peak load refers to the maximum qps that ELIS supports with the local datacenter.

Fig. 15 shows the 99%-ile latencies of ELIS and FIRM for 3 benchmarks in 6 scenarios (each benchmark has the normal and peak load scenarios). As observed, ELIS ensures the QoS targets, while FIRM results in QoS violations in all the peak load scenarios. Fig. 16 shows the supported peak load with ELIS and FIRM for the 3 benchmarks. ELIS improves the supported peak load by 17.1% on average.

Fig. 17 shows the resource usage of the benchmarks with ELIS normalized to the resource usage with FIRM in all

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on July 23,2023 at 12:57:11 UTC from IEEE Xplore. Restrictions apply.



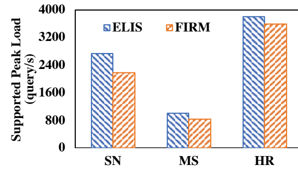


Fig. 15. The 99%-ile latency of benchmarks with ELIS and FIRM.

Fig. 16. The supported peak load of benchmarks with ELIS and FIRM.

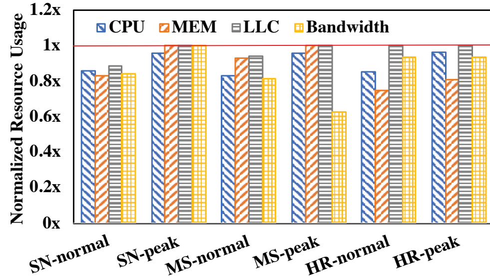


Fig. 17. The normalized total resource usage of ELIS to FIRM.

the scenarios. Compared with FIRM, ELIS uses much fewer resources in the normal load and slightly fewer resources in the peak load. The statistics results show that ELIS can reduce CPU resource usage by 15.3% and 3.8% relative to FIRM in the normal load and peak load, respectively, while guaranteeing the QoS targets for all the scenarios.

The benchmarks use fewer resources with ELIS because it allocates the microservices just-enough resources for achieving the QoS. At the normal load, ELIS not only adjusts the resources of the microservices with QoS violations (critical microservices), but also recycles the redundant resources of the other microservices. On the contrary, FIRM only identifies the critical microservices and adjusts the resources for them. At the peak load, ELIS recycles the resources of the non-critical microservices and compensates them to the critical microservices, so as to make the critical microservices have the opportunities to acquire larger resource allocation upper bound. FIRM only identifies and adjusts the critical microservices, but the resources for critical microservices cannot be raised up to the optimal value (satisfying QoS) as the remaining resources in the local datacenter are insufficient.

Fig. 18 shows the resource allocation of an example of *HR* at peak local load. As observed from this figure, ELIS first recycles the resources of *geo*, *profile*, *rate* and *search*, and then allocates them to the critical microservices for raising the resource quotas, so as to meet the QoS and throughput targets. While, FIRM only identifies the critical microservices *frontend* and *reserve*, but does not recycle the resources of other microservices. Then, FIRM expects to successively raise up the resources of the two critical microservices, but the resources of local datacenter are full when it raises the resources of *frontend* from 2.6 to 3 CPU cores. However, the resources are not adjusted to the optimal values and the QoS target is still not be achieved.

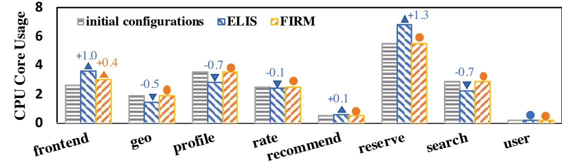


Fig. 18. The CPU allocation change with ELIS and FIRM for the HR benchmark at peak load. The upper triangle, the lower triangle and the circle represent the increase, decrease and unchange of CPU allocation, respectively.

ELIS ensures the QoS target of a user-facing application by allocating each microservice "just-enough" resources. On the contrary, some microservices may be allocated too many resources with FIRM.

E. Overhead of ELIS

Resource manager overhead. In Section V, ELIS uses the Bayesian Optimization algorithm to find the optimal resource allocations for the microservices, to make each of the microservice acquire the "just-enough" resources. Our results show that the algorithm searches for the optimal configurations for all the microservices after about 92 performance samplings. In detail, ELIS takes about 1-2 minutes to pre-train a BO resource allocation model for each microservice. After deploying the BO-based resource manager online, each prediction time is about 30ms. Moreover, the resource adjustment order decision is made in 10 milliseconds.

Microservice migrator overhead. In Section VI, when ELIS is aware of the excessive load of local datacenter, we migrate some microservices from the local datacenter to the remote datacenter. The migration is based on the functionality of Kubernetes. The amount of data transferred between all microservice pairs is profiled offline in 5 seconds. Each migration time is not more than 550ms, which includes the migration decision overhead (about 200ms) and the cold start overhead of the migrating microservice (about 350ms). The migration overhead can be further reduced by integrating with works on reducing the cost of spinning up resources [10], [35].

VIII. Conclusion

In this paper, we propose ELIS to ensure the QoS of the microservice application with excessive loads through inter-datacenter scheduling. ELIS' resource manager searches for the near-optimal resource allocations for the microservices based on the Bayesian Optimization to minimize the resource usage while ensuring the QoS. It can allocate the just-enough resources for microservices to maximize the achievable throughput of the local datacenter. We also propose a microservice migrator to migrate some microservices to the remote datacenters when local resources cannot afford the excessive loads. Considering the network cost and potential performance gain, ELIS minimizes the overall resource usage and the remote resource usage while ensuring the QoS. We have practically implemented ELIS and the experiment results show that, compared with the state-of-the-art microservice

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on July 23,2023 at 12:57:11 UTC from IEEE Xplore. Restrictions apply.

orchestration technology towards the public network, ELIS can effectively reduce the overall and the remote resource usage by 13.1% and 58.1% on average, respectively.

IX. Acknowledgment

This work is partially sponsored by the National Natural Science Foundation of China (62022057, 61832006, 61872240), and Shanghai international science and technology collaboration project (21510713600).

References

[1] Adopting microservices at netflix: Lessons for architectural design. www.nginx.com/blog/microservices-at-netflix-architectural-best-practices.

[2] Aliyun. https://www.aliyun.com/.

[3] AWS. https://aws.amazon.com/cn/.

[4] Decomposing twitter: Adventures in serviceoriented architecture. www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture.

[5] Jaeger. docs.instana.io.

[6] kubenurse. https://github.com/postfinance/kubenurse.

[7] Production-grade container orchestration. kubernetes.io.

[8] Prometheus. https://prometheus.io/.

[9] wrk2. http://github.com/giltene/wrk2.

[10] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[11] AWS. Implementing microservices on aws. docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices-on-aws.html.

[12] AWS. Regions and availability zones. aws.amazon.com/about-aws/global-infrastructure/regions_az/?nc1=h_ls.

[13] Baxtel. Facebook data center locations. baxtel.com/data-centers/facebook.

[14] S. Chen, C. Delimitrou, and J. F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[15] A. Cloud. Case studies of sina weibo. www.alibabacloud.com/help/doc-detail/193187.htm.

[16] EmptyQ. Sina weibo's architecture evolution to respond to elastic expansion. blog.emptyq.net/a?ID=00004-e7e98710-4c75-40a1-808a-28994bed273c.

[17] K. Fu, W. Zhang, Q. Chen, D. Zeng, X. Peng, W. Zheng, and M. Guo. Qos-aware and resource efficient microservice deployment in cloud-edge continuum. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 932–941, 2021.

[18] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou. Sage: practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 135–151, 2021.

[19] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[20] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, page 19–33, 2019.

[21] A. U. Gias, G. Casale, and M. Woodside. Atom: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1994–2004, 2019.

[22] GoogleCloud. Global locations - regions zones. cloud.google.com/about/locations.

[23] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu. Gaia: Geo-distributed machine learning approaching LAN speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 629–647, 2017.

[24] Y. Huang, Y. Shi, Z. Zhong, Y. Feng, J. Cheng, J. Li, H. Fan, C. Li, T. Guan, and J. Zhou. Yugong: Geo-distributed data and job placement at scale. *Proceedings of the VLDB Endowment*, 12(12):2155–2169, 2019.

[25] C.-C. Hung, L. Golubchik, and M. Yu. Scheduling jobs across geo-distributed datacenters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, page 111–124, 2015.

[26] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth European Conference on Computer Systems*, pages 1–16, 2019.

[27] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[28] S. Long, W. Long, Z. Li, K. Li, Y. Xia, and Z. Tang. A game-based approach for cost-aware task assignment with qos constraint in collaborative edge and cloud environments. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1629–1640, 2021.

[29] K. Matthews. Can your data center handle black friday and cyber monday? www.vxchnge.com/blog/black-friday-and-cyber-monday.

[30] Microsoft. Azure network round-trip latency. docs.microsoft.com/en-us/azure/networking/azure-network-latency.

[31] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander. Twig: Multi-agent task management for colocated latency-critical cloud services. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 167–179, 2020.

[32] T. Patel and D. Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 193–206. IEEE, 2020.

[33] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825, 2020.

[34] R. B. Roy, T. Patel, and D. Tiwari. Satori: efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 292–305, 2021.

[35] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[36] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[37] L. Suresh, P. Bodik, I. Menache, M. Canini, and F. Ciucu. Distributed resource management across process boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing*, page 611–623, 2017.

[38] H. Yang, Q. Chen, M. Riaz, Z. Luan, L. Tang, and J. Mars. Powerchief: Intelligent power allocation for multi-stage applications to improve responsiveness on power constrained cmp. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 133–146, 2017.

[39] D. Yu. How does alibaba cloud power the biggest online shopping festival? www.alibabacloud.com/blog/how-does-alibaba-cloud-power-the-biggest-online-shopping-festival_231673.

[40] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 167–181, 2021.

[41] L. Zhao, Y. Yang, K. Zhang, X. Zhou, T. Qiu, K. Li, and Y. Bao. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[42] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, page 149–161, 2018.

Authorized licensed use limited to: Shanghai Jiaotong University. Downloaded on July 23,2023 at 12:57:11 UTC from IEEE Xplore. Restrictions apply.