QueueFlower: Orchestrating Microservice Workflows via Dynamic Queue Balancing

Hongchen Cao†, Xinrui Liu†, Hengquan Guo, Jingzhu He* and Xin Liu*
School of Information Science and Technology, ShanghaiTech University, Shanghai, China
Email:{caohch2023, liuxr2023, guohq, hejzh1, liuxin7}@shanghaitech.edu.cn

***Abstract*—In microservices, requests' workflows with the complex dependency graphs pose challenges to auto-scaling strategies. This paper presents QueueFlower, an adaptive and dependency-agnostic auto-scaling framework for orchestrating microservice workflows. QueueFlower leverages real-time latency feedback to estimate queue lengths, effectively identifying congested services without offline profiling. Unlike previous methods that build dependency graphs between services, QueueFlower operates on individual services and adjusts resources proportionally based on estimated queues, ensuring resources of services are balanced globally. We have implemented a prototype of QueueFlower and evaluated its performance on a real-world microservice application. The experimental results demonstrate that compared to baseline methods, QueueFlower significantly reduces request latencies and percentages of SLA violations under stationary and non-stationary workloads.**

***Index Terms*—Microservices, Auto-scaling, Service-Level Agreements, Queue Balancing, Availability**

I. Introduction

The microservice architecture decouples applications into smaller, autonomously functioning modules, enhancing manageability for agile development and upgrades [1], [2]. Microservice applications usually serve various types of requests that are represented by unique workflows consisting of multiple services. Since the requests' workflows are becoming lengthy and complex, designing a suitable auto-scaling strategy to meet Service-Level Agreements (SLAs) is becoming increasingly challenging. First, it is challenging to have prior knowledge of services' dependencies of all types of requests served by a microservice application. With the increasing volume of microservice applications, the number of served request types is also increasing. Additionally, the services' dependencies of requests' workflows are becoming more and more complex. Fig. 1 depicts the dependency graph for the reservation request served by a microservice application. The vertices/nodes represent the services, and the red dashed lines represent the request's workflow paths. The request's workflow consists of ten flow paths routed through six services. Existing work leverages such dependency among services of requests to design proper scaling strategies for microservices. For example, Nodens [3] recorded the requests' dependency graphs during offline profiling, and leveraged runtime network bandwidth information to further infer requests' congestion states heuristically. Nodens then scaled out resources for congested services. However, such scaling strategies could suffer from the lack of prior knowledge of all the requests' dependency graphs that are offline profiled.

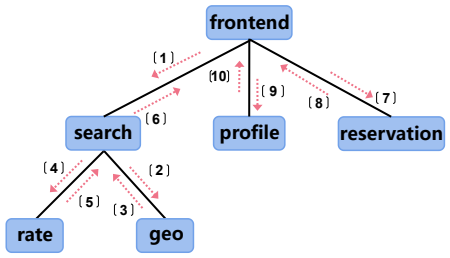


Fig. 1: Dependency graph of the reservation request served by a microservice application.

Second, even though the services' dependencies of a request workflow are known, it is still challenging to scale in and out properly for the services routed through the request, given a fixed number of containers. It is essential to have detailed runtime information (e.g., requests' congestions and services' capacities) to design an "optimal" scaling mechanism. However, such information is difficult to fetch in real-world service systems. Instead, the end-to-end request's latency can be easily obtained, and used as a common criterion by scaling mechanisms. Existing work allocated resources to all the services of a particular type of requests with higher end-to-to latencies that violate SLAs [4], [5].

In this paper, we introduce QueueFlower, an adaptive, dependency-agnostic, and lightweight auto-scaling framework. QueueFlower does not require offline profiling and leverages runtime latency feedback to estimate the queue length (the volume of congested requests). Unlike previous methods that profile requests' dependency graphs, QueueFlower is agnostic to the intricate service dependency and proportionally adjusts resources based on the estimated queue vectors. Our contributions can be summarized as follows:

- **An online adaptive framework.** QueueFlower utilizes real-time latency/SLA information to estimate the real queues of services which track the per-service backlogs and bottlenecks in a microservice application. QueueFlower adjusts resources proportionally to the estimated queues so that all queues/bottlenecks are balanced/ad-

† These authors contributed equally to this work.
* Corresponding authors.

dressed globally.

- **Dependency–agnostic scaling.** QueueFlower operates on individual services and is agnostic to the complex dependency between services. Therefore, it offers a versatile framework that can be easily integrated into any microservice application.
- **Prototype implementation.** We have implemented a prototype of QueueFlower framework, and evaluated its performance on a real-world microservice application. The experimental results show that QueueFlower's scaling strategy minimizes requests' latencies and percentages of SLA violations under both stationary and non-stationary workloads.

II. Related Work

Dependency-aware methods. Deepscaler [6] employed a graph neural network to learn the dependency graph adaptively and used it to configure the resources of the interacting services. Madu [7] also first performed the workload learning by considering dynamic call graphs into the loss function and incorporated OS-level metrics for allocation. The work [8] formulated the microservice placement problem into a fractional polynomial problem based on identified complicated dependencies in order to optimize the average response time. HAB [4] utilized Jackson queueing network [9] to model the intricate and stochastic dependency among the services node.

Machine learning-based methods. Sinan [10] employed a convolution neural network for short-term latency prediction and a boosted tree model for the probability estimation of long-term SLA violation, informing a rule-based scheduler that upholds SLA standards. [11] proposed an integer nonlinear model to minimize cost during the resource allocation process. [12] utilized a dynamic programming-based offline algorithm to reduce latencies. Reinforcement Learning-based approaches [13]–[16] are also used for resource configuration.

However, these approaches must collect substantial historical data for dependency analysis or offline training, which takes time for instantaneous configuration and may fail to be adaptive to unpredictable shifts in workload and changes in the call graph in the practical microservice system.

III. System Model

We consider a microservice application represented by a graph $\mathcal{G} = (\mathcal{N}, \mathcal{L})$, where $\mathcal{N}$ is the set of nodes and $\mathcal{L}$ is the set of links. A node represents a service component, and a link represents the computation or communication dependency between services. The users' requests continuously arrive at the microservice system, with each type of request is abstracted as a flow f, belonging to the flow set $\mathcal{F}$, goes from its source service node $s(f)$ to its destination service node $d(f)$. We denote $\mathcal{R}^f$ as the route of a flow f, including its sequence of service nodes and $\mathcal{F}_n$ as the set of flows going through node n. We further let λ^f be the expected arrival rate of flow f.

Pods are the smallest units in microservices, each hosting one or more containers that share storage and networking. For any time slot t, the system manager needs to orchestrate a resource allocation $x(t) = (x_n(t))_{n\in\mathcal{N}}$ on the service nodes such that all resources are fully utilized, i.e., $\sum_{n\in\mathcal{N}} x_n(t) = X$, where X is the total number of pods allocated to the system. Our goal is to find an optimal resource allocation such that the requests' latency and SLAs can be minimized and satisfied.

We introduce $q_n^f(t)$ as the number of unserved requests waiting for the service n at the period t. According to the well-known Little's law [17], the average latency is proportional to the queue length along its route $\mathbb{E}[\sum_{n\in\mathcal{R}^f} q_n^f(t)]$. Intuitively, to minimize the latency, it is required to minimize/balance all queues along its route, which could be challenging because all queues are coupled, as evident from queue dynamics. Let $A_n^f(t)$ represents the number of flow requests f that arrive in the system and $D_n^f(t)$ denotes the number of requests completed by the end of period t. The queue update at service n is

$$q_n^f(t+1) = (q_n^f(t) + A_n^f(t) - D_n^f(t)), \tag{1}$$

when the service n is the starting service, i.e., $n = s(f)$, and

$$q_n^f(t+1) = (q_n^f(t) + D_m^f(t) - D_n^f(t)), \tag{2}$$

when the precedence of the service n is m for flow f.

Therefore, we must consider its successor service when allocating pod $x_n^f(t)$ to minimize the total queue length. A potential solution is the "backpressure" type allocation [18], which utilizes the queue difference between two adjacent services, i.e., $q_m^f(t) - q_n^f(t)$, as the signal. However, several challenges exist in the practical microservice systems: 1) the knowledge of flow-level queue length and its backpressure is not available; 2) the relationship between the allocation $\{x_n^f(t)\}$ and the actual departure $\{D_n^f(t)\}$ is unknown; 3) executing the per-flow allocation $\{x_n^f(t)\}$ incurs huge additional costs. Fortunately, from the whole system perspective, our target is the system-level performance, that is, to minimize the queue length for all flows as follows

$$\mathbb{E}[\sum_{f\in\mathcal{F}} \sum_{n\in\mathcal{R}^f} q_n^f(t)], \tag{3}$$

We then exchange the order of the summation in Eq. (3) such that

$$\mathbb{E}[\sum_{f\in\mathcal{F}} \sum_{n\in\mathcal{R}^f} q_n^f(t)] = \mathbb{E}[\sum_{n\in\mathcal{N}} \sum_{f\in\mathcal{F}_n} q_n^f(t)].$$

Let $q_n(t) = \sum_{f\in\mathcal{F}_n} q_n^f(t)$, we have decomposed the flow-level queues into the service-level queues, which is the key motivation and theoretical foundation behind QueueFlower, such that we can utilize the real-time service-based information, agnostic to the complex dependence among flows, to minimize the system-level latency.

IV. QueueFlower

This section introduces QueueFlower, a queue-balancing framework designed for efficiently orchestrating the resources in microservice application systems. As shown in Fig. 2, the performance monitor keeps capturing runtime trace data of

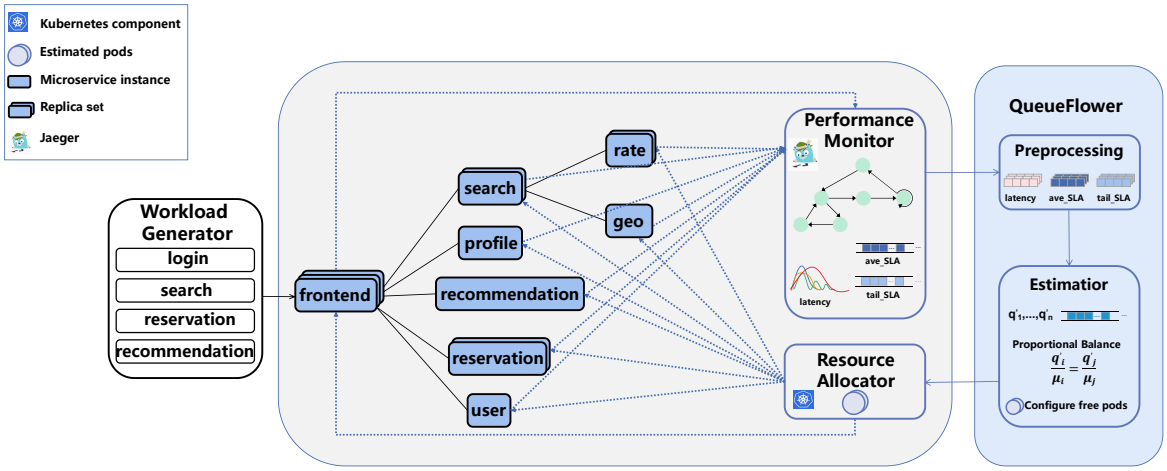


Fig. 2: System overview of QueueFlower.

requests from the microservice system. QueueFlower then *preprocesses* the data, *estimates* the per-service queues, and optimizes the resource allocation via a queue balancing algorithm. The resource allocator deploys the pod allocation into the microservice application.

Preprocessing. The *Preprocessing* module retrieves raw tracing data in JSON format from *Performance Monitor*, which records information about each request by deploying a tracing module within each container. Each trace comprises multiple spans and each span represents a service of the request. By traversing the tree path, we can collect the services routed through a type of request. The duration of a particular span consists of the request's waiting time, service processing time, the latency caused by all the immediate child spans, and the communication time with child spans. We extract the per-service latency as the difference between the duration of the span and its immediate child spans. We observe that when the request arrival rate is large (i.e., request congestion exists), the service processing time and communication time are too small compared with the waiting time of requests. Such waiting time can be incurred by context switches, request buffering, request serialization/serialization and etc. In this case, the per-service latency reflects the request's waiting time, which is useful to estimate the per-service queue described in Sec. III.

Estimator. Motivated by the per-service queueing structure in Sec. III, the key component in QueueFlower is to estimate service queues $\{q_n(t)\}$. To further guarantee SLAs, we consider SLA violations of average and tail latencies as well as average latency at a service. It is worth emphasizing that estimating a flow's latency on individual services along its route is generally impossible. However, service-level latencies and flow-level SLAs are relatively easy to acquire. QueueFlower estimates these three key metrics, carefully chooses the appropriate weights (w_1, w_2, w_3), and calculates the virtual service queue

$$\hat{q}_n(t) = w_1 l_n(t) + w_2 \sum_{f \in \mathcal{F}_n} c_f(t) + w_3 \sum_{f \in \mathcal{F}_n} g_f(t).$$

Algorithm 1 Queue Balancing in QueueFlower

1: **Input:** service nodes set $\mathcal{N}$, flow-node set $\mathcal{F}_n$, total number of pods X, and queue weights (w_1, w_2, w_3).
2: **Initialization:** $\hat{q}_n(1) = 1, \ \forall n \in \mathcal{N}$.
3: **for** $t = 1, ..., T$ **do**
4: **Queue Estimation:**

$$\hat{q}_n(t) = w_1 l_n(t) + w_2 \sum_{f \in \mathcal{F}_n} c_f(t) + w_3 \sum_{f \in \mathcal{F}_n} g_f(t).$$

5: **Queue Balancing Allocation:** construct the pods configuration $x_n(t)$ so that

$$x_n(t) = \left\lfloor \frac{\hat{q}_n(t)}{\sum_{n \in \mathcal{N}} \hat{q}_n(t)} \cdot X \right\rfloor.$$

Allocate the remaining pods $r(t) := X - \sum_{n \in \mathcal{N}} x_n(t)$ (if exist, i.e., $r(t) > 0$) according to the probability distribution$\{\hat{q}_n(t) / \sum_{n \in \mathcal{N}} \hat{q}_n(t)\}_n$.
6: **Deploy Pods Allocation and Extract Key Metrics:**

$$\{l_n(t)\}_{n \in \mathcal{N}}, \{c_f(t), g_f(t)\}_{f \in \mathcal{F}},$$

where $l_n(t)$ represents the average latency (node-level); $c_f(t)$ and $g_f(t)$ represent the averaged latency SLA violation and tail latency SLA violation (flow-level).
7: **end for**

The virtual service queues $\{\hat{q}_n(t)\}$ indicate the level of congestion in real-time in the microservice graph. Intuitively, a service with a large queue demands more resources. However, as services are coupled through flows in a complex manner, it is very likely to cause queue imbalance by using a greedy or localized allocation. Motivated by the fairness scheduling in stochastic data networks [19], [20], QueueFlower allocates pod resources on the services globally and dynamically by

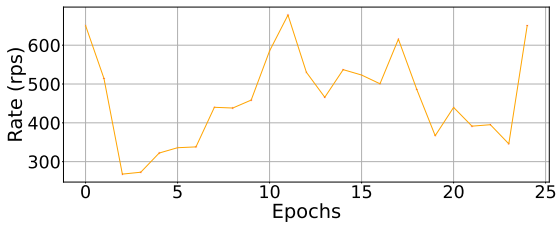


Fig. 3: The workload pattern of a non-stationary traffic.

solving the following proportionally fair optimization

$$\max_{\{x_n\}} \sum_{n=1}^{N} \hat{q}_n(t) \log x_n \quad \text{s.t.} \sum_{n=1}^{N} x_n = X,\ x_n \geq 0, \forall n.$$

Fortunately, the optimal solution has the closed form that is proportional to their queue lengths as follows:

$$\frac{\hat{q}_n(t)}{x_n(t)} = \frac{\hat{q}_m(t)}{x_m(t)}, \quad \forall n, m \in \mathcal{N},$$

where $\sum_{n\in\mathcal{N}} x_n(t) = X$. This proportional allocation balances all service queues simultaneously. We summarize the queue balancing in QueueFlower in Algorithm 1, and we indeed observe that QueueFlower adjusts the resource dynamically and globally so that the average latency is minimized. We can always find a set of queue weights (w_1, w_2, w_3) such that $\hat{q}_n(t) = q_n(t)$ and the total queue length is bounded under QueueFlower. In other words, QueueFlower can support the maximum flow arrival rate such that every flow has a finite latency, i.e., QueueFlower achieves throughput optimality. We partially reference [21] for our proof.

V. Experimental Evaluation

In this section, we present the experimental evaluation. We have implemented a prototype of QueueFlower and conducted experiments on a container cluster on a host equipped with a 2.5GHz Intel i7-11700 CPU, 32GB memory, and running a 64-bit Ubuntu 18.04 operating system. QueueFlower's source code is available at https://github.com/caohch-1/QueueFlower.

A. Implementation

Containerization and orchestration. We utilize Docker Desktop [22] and Kubernetes [23] as our containerization engine and container orchestration platform. Kubernetes manages several deployments and each deployment provisions a type of service. By simply changing the number of pods within each deployment, QueueFlower can rapidly scale in or out particular services along requests' routes. Note that the total number of pods allocated for any application has a maximum value, meaning that the system cannot allocate too many resources for an application, further causing resource shortages for other co-located applications. In our experiment, we set the maximum pod number of an application to 24.

Benchmarking microservice and workload generator. To evaluate QueueFlower's performance, we deploy the Hotel Reservation application of the open-source microservice benchmark Deathstar [24]. We employ wrk2 [25], a widely-used HTTP benchmarking tool, as our workload generator, to send four types of requests, including login, search, reservation, and recommendation. Login and recommendation requests are processed by short workflows. Search and reservation requests are processed by lengthy workflows. We can easily change the request rate by tuning the configurable parameters of the workload generator.

Performance monitor. We employ Jaeger [26] for implementing end-to-end distributed tracing in our system to collect per-service latencies under various requests.

Resource allocator. We implement a pod number controller using the Kubernetes Python client [27]. By dynamically adjusting the number of pods within each deployment, we can effectively manage resource allocation based on QueueFlower's estimation results.

B. Evaluation Methodology

1) Evaluation Metrics: Our evaluation focuses on three primary metrics: average latency, P90 tail latency, and Service-Level Agreement violations (SLAVs).

Average latency. The metric represents the average response time of each type of request.

P90 tail latency. The metric represents the latency value below which 90% of requests' response time falls. The metric is useful for latency-sensitive tasks.

Percentages of Service-Level Agreement violations (SLAVs). SLAVs measure the percentage of requests that fail to meet user-defined SLAs, typically in terms of response time thresholds. A lower SLAV percentage indicates better adherence to performance guarantees and better service availability.

2) Baseline Methods: We compare QueueFlower with the average allocation strategy (AVG), Kubernetes Auto-scaler (HPA) [28], and the state-of-the-art algorithm Holistic Auto-scaling (HAB) [4].

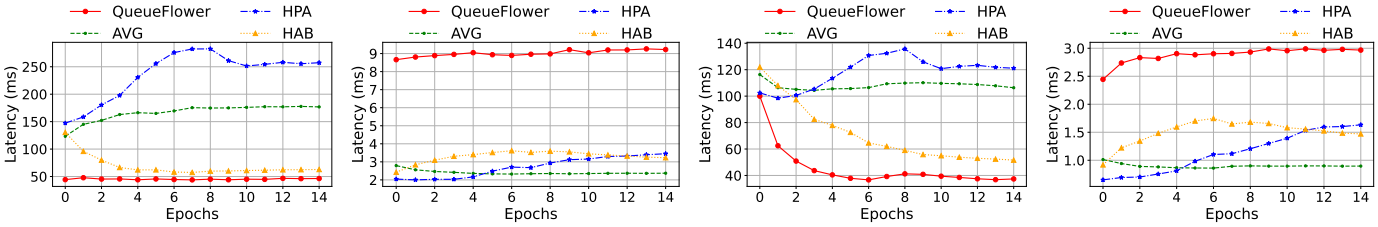
Average allocation strategy (AVG). This strategy distributes the available resources equally to each service in the microservice system.

Horizontal Pod Autoscaler (HPA). HPA continuously monitors CPU or memory usage and adjusts the pod count accordingly, ensuring that the current metric value stays below the target resource utilization. In our experiment, we set the target CPU utilization to 60% by default.

Holistic auto-scaling (HAB): HAB [4] is based on Jackson queuing network framework. HAB establishes the close form of system latency under strong assumptions on the arrival and service processes, which is used to search for the best resource allocation for services.

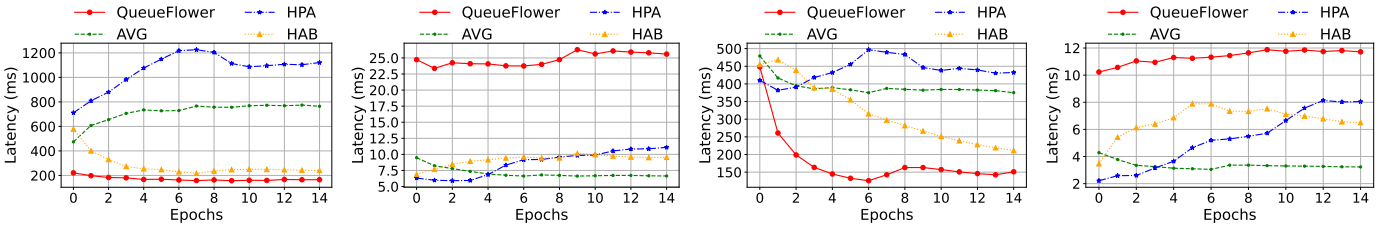
3) Experiment Design: We design two sets of experiments to validate the effectiveness of QueueFlower and compare it with baseline methods. We repeat either set of experiments for 10 times and get the averaged values of evaluation metrics to reduce randomness.

In the first set of experiments, four types of requests arrive at a stationary rate with the distribution [35%, 30%, 20%, 15%], and they are sent at the rate of 800 requests per second.



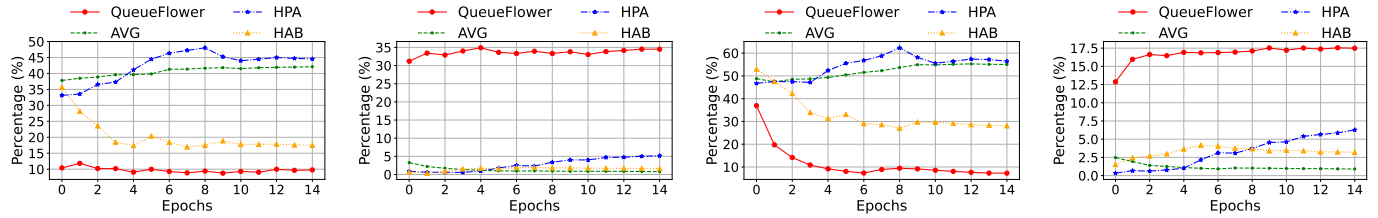
(a) Search. (b) Recommendation. (c) Reservation. (d) Login.

Fig. 4: The average latency for the four types of requests under a stationary traffic pattern.



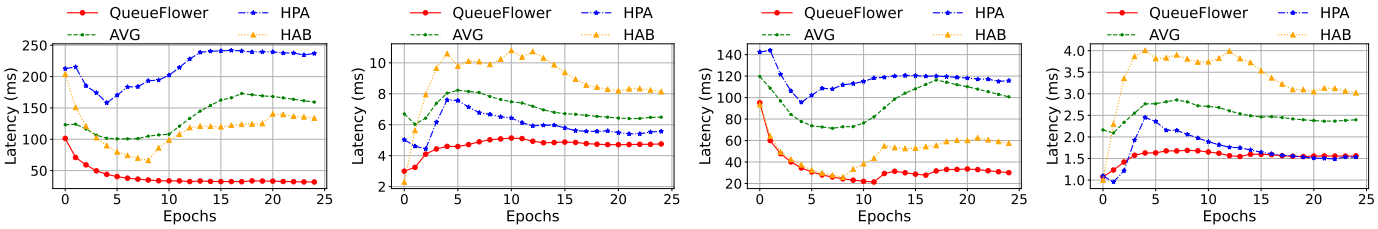
(a) Search. (b) Recommendation. (c) Reservation. (d) Login.

Fig. 5: The P90 tail latency for the four types of requests under a stationary traffic pattern.



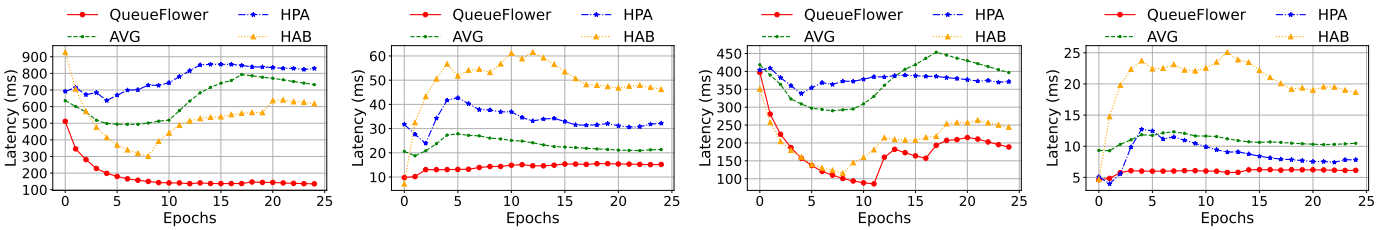
(a) Search. (b) Recommendation. (c) Reservation. (d) Login.

Fig. 6: The SLA violations for the four types of requests under a stationary traffic pattern with strict SLA [50ms, 50ms, 3ms, 1ms] for search, reservation, login, and recommendation.



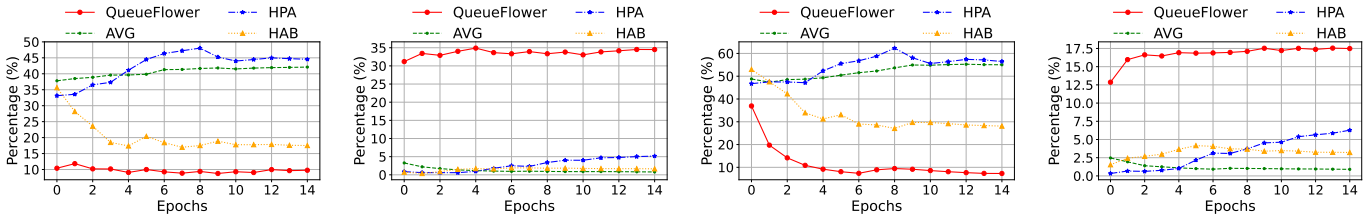
(a) Search. (b) Recommendation. (c) Reservation. (d) Login.

Fig. 7: The average latency for the four types of requests under a non-stationary traffic pattern.



(a) Search. (b) Recommendation. (c) Reservation. (d) Login.

Fig. 8: The P90 tail latency for the four types of requests under a non-stationary traffic pattern.



(a) Search. (b) Recommendation. (c) Reservation. (d) Login.

Fig. 9: The SLA violations for the four types of requests under a non-stationary traffic pattern with strict SLA [50ms, 50ms, 3ms, 1ms] for search, reservation, login, and recommendation.

We set strict SLAs, i.e., user-defined response time thresholds, for four types of requests to 50, 50, 3, and 1 milliseconds, respectively. We set strict SLAs to evaluate QueueFlower's reaction to various requests' SLAs.

For the second set of experiments, we assess QueueFlower and other baselines under a non-stationary workload. The distribution of four types of requests remains the same. However, the request sending rate is dynamically changing. As shown in Fig. 3, we change the request rate by applying a daily workload pattern extracted from real-world data collected in a production MLaaS cluster in Alibaba [29].

C. Result Analysis

1) Scaling under Stationary Workload: Fig. 4a- 4d, 5a- 5d, and 6a- 6d plot the average latencies, P90 tail latencies, and percentage of SLA violations under QueueFlower, AVG, HPA and HAB for the search, recommendation, reservation, and login requests, respectively. The results show that since HPA does not take the request congestion state into account, not all the available pods are allocated. HPA determines the resource allocation only based on CPU or memory consumption, resulting in high latency and a percentage of SLA violations. AVG allocates all available resources evenly, this purely static policy cannot dynamically allocate the right amount of resources to particular services with high latencies, and thus is not as effective as QueueFlower and HAB.

We observe that QueueFlower achieves lower latencies than the baseline methods on search and reservation requests, while it achieves higher latencies than the baseline methods on login and recommendation requests. The result is expected, since login and recommendation requests' routes are short, and search and reservation requests' routes are lengthy. To reduce the overall latency of all requests, QueueFlower tends to allocate more resources to services along the lengthy requests' routes. To summarize, compared with the baseline methods, QueueFlower achieves significant performance gains in all three metrics. QueueFlower has a lower average latency with gains of 64.93, 97.85, and 8.84 milliseconds; a lower P90 tail latency with gains of 272.35, 415.93, and 40.92 milliseconds; and a lower SLA violation with gains of 17.26%, 20.28%, and 1.17%. These results show that QueueFlower can automatically allocate different amounts of resources based on the latency of different services to reduce the overall latency and SLA violations.

2) Scaling under Non-stationary Workload: Compared to stationary workload, non-stationary workload requires scaling mechanisms to quickly reallocate resources to blocked services in case of an unknown change in request arrival rate.

Fig. 7a- 7d, 8a- 8d, and 9a- 9d plot the average latencies, P90 tail latencies, and percentages of SLA violations under QueueFlower, AVG, HPA and HAB for the search, recommendation, reservation, and login requests, respectively. Benefiting from the design of proportional adjustment based on the updates to the virtual queues, QueueFlower can quickly reallocate available resources after the services' latencies have changed. More resources are allocated to highly loaded services to reduce the overall latency. Considering all four requests together, QueueFlower has a lower average latency with gains of 66.25, 97.59, and 44.7 milliseconds; a lower P90 tail latency with gains of 273,02, 300.58, and 192.86 milliseconds; and a lower SLA violation with gains of 23.16%, 35.08%, and 16.16%. These results show that QueueFlower can efficiently allocate available resources to guarantee low latencies and a percentage of SLA violations under the highly dynamic non-stationary workload.

VI. Conclusion

In this paper, we propose QueueFlower, a novel online autoscaling framework that leverages the real-time system information, identifies congested services, and adjusts the number of resources allocated for service nodes via queue balancing algorithm. QueueFlower can support the maximal flow arrival rate (throughput optimal), thus providing guarantees for the user-defined Service-Level Agreements. We have implemented a prototype of QueueFlower, and conducted experiments on a real-world microservice application. The results show that QueueFlower outperforms the existing mechanisms in reducing requests' latencies and minimizing SLA violations under both stationary and non-stationary workloads.

Acknowledgement

The work was partially supported by the Shanghai Sailing Program 22YF1428600 and 22YF1428500, the National Nature Science Foundation of China under grant 62302305.

REFERENCES

[1] L. Bănică, C. Ștefan, and A. Hagiu, "Leveraging the microservice architecture for next-generation iot applications," *Scientific Bulletin – Economic Sciences*, 2017.

[2] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019.

[3] J. Shi, H. Zhang, Z. Tong, Q. Chen, K. Fu, and M. Guo, "Nodens: Enabling resource efficient and fast qos recovery of dynamic microservice applications in datacenters," in *USENIX Annual Technical Conference*, 2023.

[4] J. Tong, M. Wei, M. Pan, and Y. Yu, "A holistic auto-scaling algorithm for multi-service applications based on balanced queuing network," in *IEEE International Conference on Web Services (ICWS)*, 2021.

[5] H. Guo, H. Cao, J. He, X. Liu, and Y. Shi, "Pobo: Safe and optimal resource management for cloud microservices," *Performance Evaluation*, 2023.

[6] C. Meng, S. Song, H. Tong, M. Pan, and Y. Yu, "Deepscaler: Holistic autoscaling for microservices based on spatiotemporal gnn with adaptive graph learning," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[7] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, G. Yang, and C. Xu, "The power of prediction: microservice auto scaling via workload learning," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022.

[8] X. He, Z. Tu, M. Wagner, X. Xu, and Z. Wang, "Online deployment algorithms for microservice systems with complex dependencies," *IEEE Transactions on Cloud Computing*, 2023.

[9] J. R. Jackson, "Jobshop-like queueing systems," *Manag. Sci.*, 2004.

[10] Y. Zhang, W. Hua, Z. Zhou, 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*, 2021.

[11] Z. Ding, S. Wang, and C. Jiang, "Kubernetes-oriented microservice placement with dynamic resource allocation," *IEEE Transactions on Cloud Computing*, 2023.

[12] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, 2021.

[13] C. Meng, J. Tong, M. Pan, and Y. Yu, "Hra: An intelligent holistic resource autoscaling framework for multi-service applications," in *2022 IEEE International Conference on Web Services (ICWS)*, 2022.

[14] N. Liu, Z. Li, Z. Xu, J. Xu, S. Lin, Q. Qiu, J. Tang, and Y. Wang, "A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[15] Z. Yang, P. Nguyen, H. Jin, and K. Nahrstedt, "Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[16] G. Tesauro, N. Jong, R. Das, and M. Bennani, "A hybrid reinforcement learning approach to autonomic resource allocation," in *IEEE International Conference on Autonomic Computing*, 2006.

[17] J. D. C. Little, "A proof for the queuing formula: $l = \lambda w$," *Operations Research*, 1961.

[18] L. Tassiulas and A. Ephremides, "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks," *ieeetac*, 1992.

[19] F. Kelly, "Charging and rate control for elastic traffic," *European Transactions on Telecommunications*, 1997.

[20] F. P. Kelly and R. J. Williams, "Fluid model for a network operating under a fair bandwidth-sharing policy," *The Annals of Applied Probability*, 2004.

[21] B. Li and R. Srikant, "Queue-proportional rate allocation with per-link information in multihop wireless networks," *Queueing Systems*, 2016.

[22] "Docker desktop," https://www.docker.com/products/docker-desktop/.

[23] "Kubernetes," https://kubernetes.io/.

[24] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "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*, 2019.

[25] "Wrk2," https://github.com/giltene/wrk2.

[26] "jaeger," https://www.jaegertracing.io/.

[27] "Kubernetes python client," https://github.com/kubernetes-client/python.

[28] "Kubernetes horizontal pod autoscaling," https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[29] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "Mlaas in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation,*, 2022.