GRAF: A Graph Neural Network based Proactive Resource Allocation Framework for SLO-Oriented Microservices

Jinwoo Park*
KAIST
South Korea
jinwoo520528@kaist.ac.kr

Byungkwon Choi*
KAIST
South Korea
nfsp3k@gmail.com

Chunghan Lee
Toyota Motor Corporation
Japan
lch@toyota-tokyo.tech

Dongsu Han
KAIST
South Korea
dhan.ee@kaist.ac.kr

ABSTRACT

Microservice is an architectural style that has been widely adopted in various latency-sensitive applications. Similar to the monolith, autoscaling has attracted the attention of operators for managing resource utilization of microservices. However, it is still challenging to optimize resources in terms of latency service-level-objective (SLO) without human intervention. In this paper, we present GRAF, a graph neural network-based proactive resource allocation framework for minimizing total CPU resources while satisfying latency SLO. GRAF leverages front-end workload, distributed tracing data, and machine learning approaches to (a) observe/estimate impact of traffic change (b) find optimal resource combinations (c) make proactive resource allocation. Experiments using various open-source benchmarks demonstrate that GRAF successfully targets latency SLO while saving up to 19% of total CPU resources compared to the fine-tuned autoscaler. Moreover, GRAF handles traffic surge with 36% fewer resources while achieving up to 2.6x faster tail latency convergence compared to the Kubernetes autoscaler.

CCS CONCEPTS

• **Software and its engineering** → **Cloud computing**; • **Networks** → **Network resources allocation**.

KEYWORDS

microservices, resources optimization, graph neural networks, applied machine learning, cloud computing, autoscaler

ACM Reference Format:
Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. GRAF: A Graph Neural Network based Proactive Resource Allocation Framework for SLO-Oriented Microservices. In *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21), December 7–10, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3485983.3494866

1 INTRODUCTION

Microservice is an architectural style that structures an application into loosely coupled services, which are also called microservices. Microservice is gaining popularity, the survey report from O'Reilly in 2020 [10, 11] says 1502 respondents who took a technical role in the company have applied microservices into their business. This is because microservice's modular architecture enhances the reliability, maintenance, and scalability of overall service [58]. Also, its ease at making small deployments enables applications to be continuously developed and updated at a small cost.

Microservices are often executed on cloud servers to carry out user-facing latency-sensitive applications including Netflix [24], Amazon [2], and Airbnb [3]. Considering applications' purpose and target, indicators that represent user experiences such as availability and tail latency are often set as service level objectives (SLOs) [35]. Latency SLO in these microservices applications is especially important. Therefore, many operators choose to overprovision resources on clouds [7, 9, 60]. Unfortunately, overprovisioning comes with the cost, about $6.6 billion are wasted in the cloud because of overprovisioning [8, 14, 54]. Even slight improvements in the cloud resources allocations would result in saving millions of dollars at scale. An efficient resource allocation framework would tightly minimize associated resources in terms of tail latency to satisfy SLO.

In addition, an efficient resource allocation framework should allocate resources to every microservices on the application proactively, according to the change in front-end workload. Such measure is necessary to avoid cascading effect (discussed in § 2.1), which severely degrade microservices performance when traffic surges. The root causes of cascading effect are; inevitable deployment overhead (i.e., 15s startup latency [39]), and the nature of microservice architecture that the requests are processed along with series of microservices. When the traffic surges at the front, microservices in depth are not directly affected until prior microservices process increased workloads downwards after deploying additional resources. Proactive resource allocation to the microservice chain is the key to avoiding cascading effect, otherwise, the delay is accumulated while congestion mitigation from the first to the last microservice in the chain takes place one at a time.

Therefore, the resource allocation framework for microservices should aim for two major goals: optimizing CPU resources in terms of latency SLO (i.e., end-to-end tail latency), and proactively deploying CPU resources for every microservices according to the expected impact of the front-end workload.

Until now, autoscaler [21, 41, 46, 52, 53, 55, 62, 63] has been widely developed for resource allocation in microservices but no existing approaches consecutively address the two major goals introduced above. First, they do not target latency SLO except for FIRM [53], yet FIRM also fall in sub-optimal configuration (discussed in § 5.3). Existing autoscalers target objectives such as CPU

*The first two authors contributed equally to the paper.

CoNEXT '21, December 7–10, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-9098-9/21/12…$15.00
https://doi.org/10.1145/3485983.3494866

utilization or performance (e.g., throughput, profit/cost). For example, Kubernetes (K8s) autoscaler observes resource utilization of each microservice and make scaling decision to keep utilization under certain threshold. K8s autoscaler's such behavior can achieve optimal at server-side resources utilization, but this does not lead to achieving optimal in latency SLO's perspective. For the above reasons, K8s autoscaler does not fix the utilization threshold near 100% which would minimize total CPU resources but provides a function that can change the threshold of resource utilization on-demand. However, this only leaves the burdens of balancing between operating costs and quality-of-service to human operators. The operators are left with no choice but to set a threshold with a large extra margin to satisfy latency SLO, which results in overprovisioning. Second, many existing autoscalers [21, 53, 55, 62, 63] control resources of each microservice individually, which can not avoid cascading effects and suffer from severe performance degradation when traffic surges. Those autoscalers are blindsight until the deepest microservice in the chain is affected by the workload change, unable to fully react to traffic surge. Combined with the delay caused by instance creation time in each microservice, such immature behavior of autoscaler is not negligible in terms of tail latency.

Devising a resource allocation system that achieves two major goals rises several non-trivial requirements:

- First, predicting end-to-end percentile latency of microservices under multi-APIs workload, assuming planned CPU resources are deployed. Deploying CPU resources and observing the end-to-end percentile latency is not affordable while servicing a microservices application. Any kind of trial and error method will immeasurably harm user experience, degrading the performance of microservices.
- Second, finding optimal CPU resource configuration for every microservice component at once, within the decision time window. Unfortunately, it's almost infeasible to find optimal resource configuration from x to the nth possible combinations within few seconds with global optimization algorithms.
- Third, proactive resource allocation should be made to avoid cascading effect when traffic increases. This requires the resource allocation for every microservices to be made according to the expected impact of the front-end workload change.

To accomplish the above requirements, we present GRAF: a graph neural network-based proactive resource allocation framework for SLO-oriented microservices. First, GRAF analyzes the frontend's workloads of various APIs into workloads distribution upon microservices, which are paired with microservice's CPU quota to represent the node state of microservices' graphical system. Then GRAF leverages the graph neural network (GNN) to predict the end-to-end tail latency of microservices with the microservices' node states. Second, GRAF utilizes gradient descent algorithm to find out minimal resource configuration which satisfies tail latency SLO. In the loss function, the fully trained end-to-end tail latency prediction model is used as a detector for possible latency SLO violations. Gradient descent algorithm is lightweight and finds adequate resource combination fast enough. Third, GRAF only accesses frontend workload and microservice's trace data history, which allows GRAF to make proactive resource allocation decisions immediately when front-end workload differentiates. Thus, GRAF saves microservice applications from suffering extensive latency elongation when traffic surges.

To the best of our knowledge, GRAF is the first work to optimize microservice resources in terms of end-to-end tail latency. GRAF outperforms state-of-the-art resource allocation systems such as K8s autoscaler. Since Kubernetes autoscaler is not designed to target latency SLO, we have fine-tuned the threshold value of K8s autoscaler to meet latency SLO for comparison. GRAF saves 14-19% total CPU resources compared to fine-tuned threshold-based K8s autoscaler while satisfying latency SLO. GRAF utilizes 36% fewer CPU resources and achieves up to 2.6x faster tail-latency convergence compared to K8s autoscaler when traffic surges.

In summary, we make the following key contributions:

- **Tail latency prediction with GNN:** A graph neural network design that can process graph-structured systems like microservices to model complex end-to-end variables such as tail latency.
- **Proactive optimal resource allocation:** GRAF optimizes resources for every microservices in the application according to the expected impact of front-end workload change, while directly targeting end-to-end tail latency.
- **End-to-end resource allocation framework:** An end-to-end implementation and evaluation of GRAF with various applications in a real Kubernetes cluster and comparison with the state-of-the-art autoscaler.

2 OBSERVATIONS AND OPPORTUNITIES

Microservices communicate with one another and form a chain of microservices. Most existing autoscalers do not consider the microservice chain and suffer from the limitations [34]. Specifically, we observed that they experience a phenomenon called cascading effect and severely degrade the performance when traffic changes abruptly (§ 2.1) and also inefficiently optimize resources for microservices (§ 2.2). In this section, we describe our observations in detail and investigate the opportunity behind them to effectively optimize resources for microservices.

2.1 Cascading Effect

A cascading effect is a phenomenon that subsequent microservices in a chain slowly perceive changes in the workload because of the instance creation delay of previous microservices in the chain. The cascading effect severely degrades the performance of microservices when traffic surges. As described in Section 7, most existing autoscalers do not consider the microservice chain and face the cascading effect. We describe the cascading effect based on K8s autoscaler [19] that is widely used [5, 9, 18, 27]. Then, we show how to avoid it.

K8s autoscaler operates with a pre-determined resource utilization threshold. When a microservice's resource utilization reaches the threshold, the autoscaler creates more instances of the microservice to keep the utilization below a certain level [1]. Service operators can control the threshold to adjust how quickly the autoscaler reacts

[1]One can vertically scale a microservice instance up by allocating more low-level resources such as CPU or memory. However, it is insufficient because the amount of resources allocated to an instance cannot get larger than the total amount of resources in the machine it is running on [55].

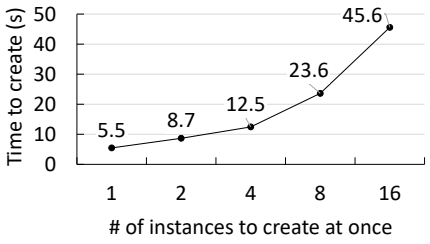


Figure 1: Time to create microservice instances.

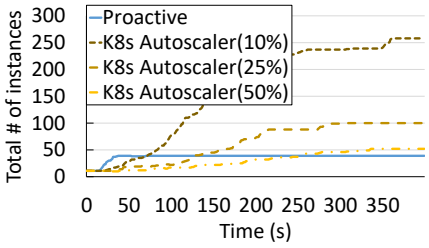


Figure 2: Total Number of microservice instances when traffic surges.

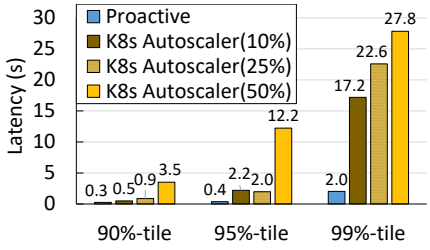


Figure 3: End-to-end latency when traffic surges.

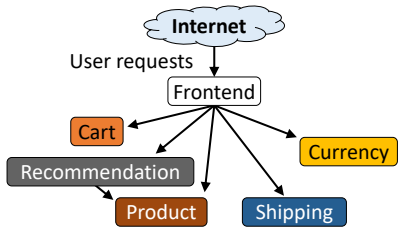


Figure 4: A microservice chain of an open source benchmark called Online Boutique [25]

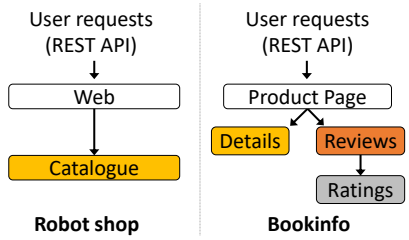


Figure 5: Microservice chain of Robot Shop [6] and Bookinfo [16]

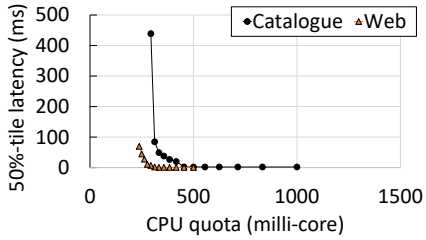


Figure 6: Latency curves of each microservice in Online Boutique application

to the changes in resource utilization. The autoscaler monitors the usage of the resources such as CPU and memory to independently creates instances for each microservice.

Cascading effect. Instance creation time is accumulated and propagated down through the microservice chains and we call this phenomenon the cascading effect. Figure 1 shows the time it takes to create instances [2]. Across microservices, it takes 5.5 seconds on average to create a single instance. Under circumstances where multiple instances are created at once, the creation time increases even more. Furthermore, in production settings, operators set an interval (e.g., 15 seconds) of how often the autoscaler makes scaling decisions to prevent the number of instances from being fluctuated. This instance creation time and control interval makes the autoscaler slow at perceiving changes in the microservices of an application.

Due to the cascading effect, the further back a microservice is located within a chain, the longer it takes for the microservice to experience the changes in workload and resource usage. For example, Figure 4 shows one of the microservice chains in an open-source benchmark called Online Boutique [25]. This microservice chain is to get a cart page of the service. When 'Frontend' receives a request from an end-user, it first sends a request to the following microservice called 'Currency'. 'Frontend' then sequentially sends a request to the successive microservice 'Cart' and so forth.

Assume that 'Currency' receives an excessive amount of requests and the autoscaler creates more instances for 'Currency'. During that time, 'Cart' is not aware of the change in the workload. After the additional instances of 'Currency' are created, 'Cart' perceives the increase of the workload. This happens sequentially to the following microservices.

Experimental result. We can observe this phenomenon in experiments. The upper graph in Figure 7 shows the workload that each microservice perceives when using K8s autoscaler. We transmit queries for the cart page at a rate of 300qps by using Vegeta [13]. While 'Frontend' perceives its peak traffic at 31s, 'Cart' starts handling its peak workload at 118s. It is because until enough number of instances for 'Frontend' is created the workload for 'Cart' does not reach the peak. The subsequent microservices see the peak even further later at 155s. A naïve approach to reducing the delay caused by the cascading effect is to lower the utilization threshold, but it comes with a lot of costs. We run the same experiment while varying the CPU utilization threshold from 10 to 50%. Figure 2 and 3 show the total number of microservice instances and the end-to-end latency, respectively. When we adjust the threshold from 50 to 10%, the 99%-tile latency decreases from 27.8 to 17.2 seconds but the total number of instances dramatically increases from 51 to 258.

Opportunity. When we create the instances for all microservices in a chain at once, we could avoid the cascading effect. We first transmit the cart queries and then manually create the heuristically determined number of instances for each microservice. As shown in Figure 2 and 3, this approach 'Proactive' reduces the 99%-tile latency by 8.6 times compared to the 10% threshold setting of K8s autoscaler while creating 6.6 times less amount of the total instances. The time to reach the peak workload for all microservices is also similar to each other at 58s, as shown in the lower graph of Figure 7. If we can automatically determine the appropriate number of instances,

[2]We create instances of microservices in [25] on a single worker node and ignore the network delay to download the container images.

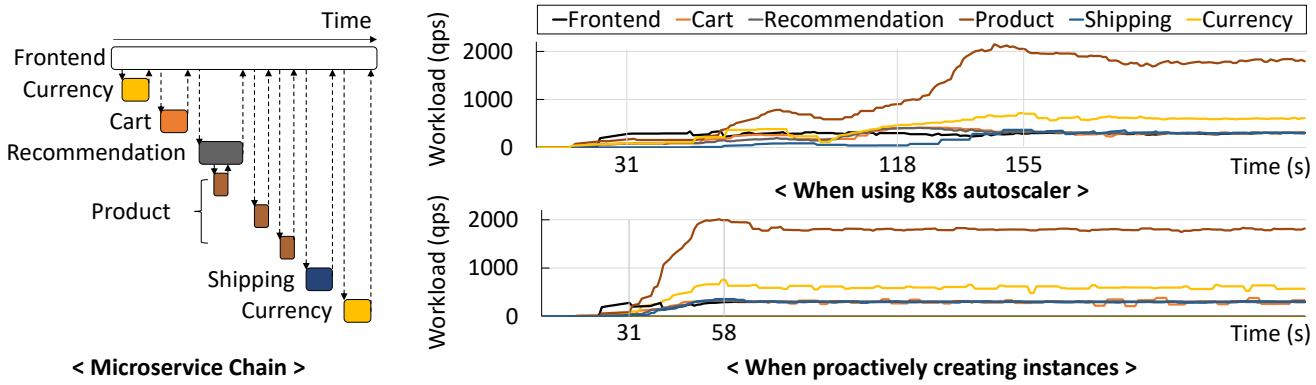


Figure 7: Microservice chain and workload that each microservice perceives when traffic surges

we can develop a resource allocation framework that avoids the cascading effect.

2.2 Microservice Latency Characteristics

Microservices share the limited resources. Depending on how to distribute resources across microservices, an application could deliver better performance with less amount of resources. For example, Figure 5(left) and 6 shows a microservice chain of a demo application called Robot Shop [6] and the relationship between latency and resource for each microservice, respectively. Because the 'Catalogue' microservice shows a more sharp curve in latency than 'Web' microservice as shown in Figure 6, one could deliver the same performance with less amount of resources by allocating more resources to 'Catalogue'. Figure 5 (right) shows a microservice chain of another demo application called Bookinfo [16]. 'Product Page' transfers requests to 'Details' and 'Reviews' in parallel, so that the end-to-end latency is determined by the longest one between them, which is 'Reviews' and 'Ratings' in this case. Even if we reduce resource allocated to 'Details', Bookinfo would show the same end-to-end latency unless the time to handle a request by 'Details' go beyond the time handled in 'Reviews' and 'Ratings'.

Opportunity. As described above, one could optimize resources for microservices in a better way when using the holistic view of the microservice chain. However, it is known that resolving high-dimensional optimization problems is non-trivial. In our case, it is feasible because the target function has convexity as latency of every microservice monotonically decreases.

3 DESIGN

Approach. Our goal is to identify the minimal CPU quota resources configuration that satisfies latency SLO. This is expressed in the formula as:

$$\min_{\vec{r}} \sum_{r \in \vec{r}} r \tag{1}$$

$$s.t.\ L(\vec{r}, \vec{w}) \leq \text{Latency SLO} \tag{2}$$

where $\vec{r}$ is the CPU quota for each microservice, $\vec{w}$ is the workload for each API (e.g., queries per sec), and $L(\vec{r}, \vec{w})$ is the end-to-end tail latency of microservices. For resource allocation to take place in real-time, the configuration must be solved simultaneously to the change of input workloads. However, difficulties arise since latency can only be measured after deploying a resource configuration to the real cluster. It is infeasible to try possible combinations in real-time because changing resources would affect the performance of the microservices. Moreover, the search space of possible combinations is very large as there are tens [6, 25, 40] to hundreds [4, 12] of microservices in an application.

To overcome this challenge, we train graph neural network (GNN) to model end-to-end tail latency instead of measuring in real-time after making resource deployments. Estimating end-to-end tail latency of the microservice application is non-trivial because every microservices have different latency curves and complex edge connections between microservices. Not only a request's end-to-end latency is a combination of multiple addition and max operations between each microservice's latency, but also some microservices' latency is affected by the performance of neighboring microservices. Accommodating the graphical structure of microservices, we leverage GNN which is known to be scalable when modeling graph-structured workloads [36, 47, 48].

GNN is trained as supervised learning to predict end-to-end tail latency with workloads, resources, and latency paired samples collected from real clusters. Specifically, GNN is structured as message passing neural network (MPNN) [42] with edge information constructed from microservices tracing data. For proactive resources allocation, we precisely restricted state features of our GNN model to use only available information at the front end. With trained GNN, frontend workload, and target latency SLO, we apply gradient descent optimization with microservices' resources as variables. Our loss function is designed to minimize total CPU resources in microservices while avoiding violation of latency SLO. Thus, GRAF finds resource combination that minimizes total CPU resources for microservices while satisfying latency SLO. Meanwhile, GRAF proactively allocates resources for every microservices according to frontend workload changes, avoiding performance degradation at traffic surge.

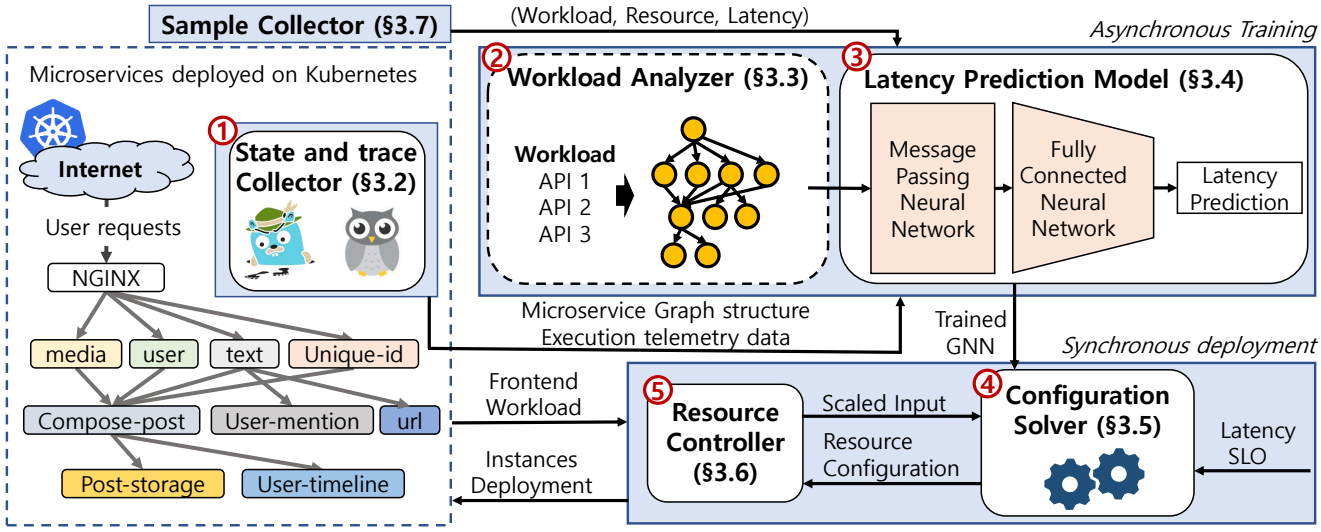


Figure 8: GRAF: Design Overview

3.1 GRAF Overview

Figure 8 illustrates an overview of the GRAF Design. GRAF operates as an end-to-end resource allocator with six components geared up together within a microservice application deployed onto real Kubernetes cluster.

1. State and trace collector (§ 3.2), collects information of microservices deployed on Kubernetes cluster. The collected data is formatted and reshaped then passed to the other components.

2. Workload analyzer (§ 3.3), analyzes front-end workload into distributed workloads to each microservices in the system. Such distributed workloads capture the state of the graph, representing a microservices dependency. The reshaped workload is fed to the latency prediction model as an input feature.

3. Latency prediction model (§ 3.4) outputs end-to-end tail-latency of microservice application with inputs of each microservice's state in pair of workload and CPU resources.

4. Configuration solver (§ 3.5) calculates resource configuration which corresponds to Eq (1), and Eq (2). The optimal configuration is found by iterating through possible resource combinations while identifying a latency SLO violation by estimated end-to-end tail latency from the above latency prediction model. Once the configuration is found, the resource controller (§ 3.6) calculates corresponding instances for each microservice and make scaling decision to the cluster.

5. For the training process, the sample collector (§ 3.7) gathers sample and build training set, validation set, and test set. Microservice state-aware sampling allows the collector to sample efficiently from real Kubernetes cluster environment where search space is very large.

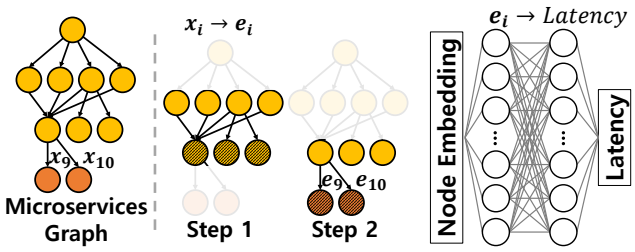
Note that *Latency Prediction Model* asynchronously trains the end-to-end tail latency prediction model with collected samples utilizing a GNN. The samples are collected by state-aware sample collector in pair of frontend workload, resources configuration, and end-to-end tail latency (e.g.,99%-tile) from a microservice application deployed on Kubernetes cluster. Then, *Configuration Solver* calculates resource configuration synchronously to frontend workload and latency SLO using gradient descent optimization and trained *Latency Prediction Model*. Then, the resource control module in *Resource Controller* makes scaling decisions on microservices according to calculated resource configuration. Each component will be further explained in detail throughout the following sections.

3.2 State and Trace Collector

The state and trace collector monitors the current states of microservices and collects the history of trace data. cAdvisor [17] is used to monitor each microservice's current CPU usage and CPU utilization. Jaeger [20] is used to collect trace data of every request executed through the microservices. These trace data include the frontend workloads, the execution path and the amount requests to microservices for each API, individual microservice's latency, and the end-to-end microservice's latency. End-to-end tail-latency is measured by picking percentile rank in the collected latency samples. The frontend workloads and the execution history of requests are delivered to the workload analyzer for further process.

3.3 Workload Analyzer

Capturing the microservice's graphical features. The state collector observes the frontend workload for each API, which we have annotated as $\vec{w}$. Front-end workloads information does not represent graph characteristics of the microservices. The workload analyzer gets front-end workloads as input and outputs distribution of workloads for each microservice in the application. With a distributed tracing called Jaeger [20], the workload analyzer collects the request history to every microservices in the chain for executing each API. Depending on the conditions same API's request history could vary, from the history 90%-ile samples are chosen to represent the behavior of the API. According to determined



(a) Node Embedding (b) Readout

Figure 9: Graph neural network composed of node embedding and readout phase to predict latency

trace data for $\vec{w}$, the amount of workload that each microservice would experience can be calculated. The frontend input workload, $\vec{w}$, is converted to each microservice's workload l_i for the microservice node i. Microservice node feature processing is completed by pairing workload l_i, and allocated CPU resource r_i for every microservices. Then, the following node features are fed into the latency prediction model as an input state for predicting end-to-end tail latency.

3.4 Latency Prediction Model

To successfully train the end-to-end tail latency prediction model, we carefully design input state features, neural network structure, and loss function. In addition, the state-aware sample collector (§ 3.7) is made for an efficient sample collection from real clusters.

Graph neural network. Our GNN is composed of MPNN [42] for graph node embedding and fully connected (FC) layers for readout as notated in Fig 9. MPNN is structured with edge connection details derived from trace data of the microservice application's every APIs.

Given the vectors of $\mathbf{x_i}$ as input features corresponding to the nodes in microservice graph representation, GRAF elicits node embedding ($\mathbf{x_i} \rightarrow \mathbf{e_i}$). The node embedding $\mathbf{e_i}$ is a vector that implies information from all neighboring nodes. To compute node embedding vector, GRAF propagates messages from parent to child nodes in a sequence of message passing steps, starting from the front-end node of the microservice. In every message-passing steps, a node i's embedding is updated with the sum of every message collected from its parent nodes and its own embedding vector as:

$$e_i = \gamma^{(k)}(x_i, \sum_{j \in N(i)} \phi^{(k)}(e_j)) \tag{3}$$

where $\gamma(\cdot)$ and $\phi(\cdot)$ are neural network implemented as multi-layer perceptrons, (k) meaning neural network of k^{th} message passing step. Also, $N(i)$ represents set of parent nodes of node i.

Specifically, our MPNN is designed to carry out two message-passing steps. In the first step, each node embedding is calculated with aggregated messages (i.e., node features), which are from one-hop anterior microservices, shown in Fig 9(a) Step 1. At the next consecutive MPNN layer, new node embedding is updated from messages (i.e., node embeddings), which are again aggregated from one-hop anterior microservices, shown in Fig 9(a) Step 2. This neural network structure allows implicit comprehension of the influences from anterior microservices in the microservice graph.

The final node embedding $\mathbf{e_i}$ from MPNN is passed to FC layers as the readout phase. At readout, two consecutive FC layers carry out end-to-end tail latency regression with input node embedding as shown in Fig 9(b). The dropout is introduced to FC layers and the validation set is used to prevent overfitting and save the best performance GNN.

Handling tail-latency loss. *Latency Prediction Model* targets end-to-end tail latency, specifically 99%-tile in our experiments. 99%-tile latency of a microservice application shows large ranges depending on allocated CPU resources. Also, it shows high variance at latency peaks, and samples of extreme irregular latency values are collected from time to time, even if sampling is done under a similar resource configuration setting.

To accommodate 99%-tile latency's characteristics, we introduce three tricks to our loss function. First, we apply percentage error so our model achieves better predictions in small latency regions (e.g., 0-200ms). Otherwise, the trained model shows the behavior of predicting accurately in big latency regions (e.g., 200-3000ms) compared to small latency regions. Second, we choose the hüber loss function instead of the mean-square loss function to increase stability during training. The Hüber loss function is designed to give out mean-square-error towards small error within bounds and linear error when a large error that's out of bounds occurs. Thus, irregular samples that show extreme values in some of the collected samples have less effect during the training process. Third, we introduce asymmetry in the loss function, it gives more penalty if the latency prediction of the model is lower than the actual value and gives less penalty, otherwise. Considering our original purpose to calculate resources configuration that satisfies latency SLO, it's more critical when the model underestimates end-to-end latency. Therefore, we avoid underestimation of our latency prediction model by penalizing more when it guess latency to be shorter than actual.

$$Loss(x) = \begin{cases} -\theta_L(2x + \theta_L) & \text{for } x < -\theta_L \\ x^2 & \text{for } -\theta_L \le x < \theta_R \\ \theta_R(2x + \theta_R) & \text{for } \theta_R \le x \end{cases} \tag{4}$$

Our loss is calculated following the above equation (4), where x represents the percentage error between model output and true label. θ_L and θ_R are constants that modulate the shape of the latency prediction model's asymmetric hüber loss function. θ_L is chosen as a larger value than θ_R to achieve desired behavior of penalizing underestimation more. As the result, trained *Latency Prediction Model* shows a slight overestimation of end-to-end tail latency.

3.5 Configuration Solver

Synchronous resource configuration. Unlike the latency prediction model which is asynchronously trained, the configuration solver must synchronously find optimal resource configuration according to front-end workload change and latency SLO demand. For synchronous operation, the solver must be light and fast. To meet the demands, GRAF practices gradient descent optimization along with variables $\vec{r}$ for the loss function Eq (5) to find out minimal

resource configuration which satisfies latency SLO.

$$Loss(\vec{r}, SLO) = \sum_{r \in \vec{r}} r + \rho * \phi(L(\vec{w}, \vec{r}), SLO) \quad (5)$$

$$\phi(x, y) = max(0, x - y) \quad (6)$$

Loss is determined with respect to change of resources $\vec{r}$, and latency SLO. The first term in the equation 5, works as the objective term for minimizing total CPU resources, while the second term works as a penalty term for violating latency SLO due to insufficient CPU resources. ρ is a penalty coefficient that amplifies the result of the penalty function $\phi(\cdot)$. Pre-trained latency prediction model $L(\vec{w}, \vec{r})$ estimates tail-latency and is used as a detector for latency SLO violations. Max function is used as a penalty function $\phi(\cdot)$ as stated in equation 6. Gradient descent algorithm can be applied because the equation 5 is end-to-end differential, including neural network model for latency estimate. Gradient descent algorithm reaches optimal value by minimizing the resources without triggering penalty term. The configuration solver iterates until the tolerance, corresponding to $Loss_t - Loss_{t-1}$ at the current t step, is less than the predetermined threshold. Although gradient descent optimization cannot guarantee finding a global minimum in non-convex functions, the monotonic relationship between each microservice's latency and CPU resource enables it. Empirical results are further provided in evaluation (§ 5.2). We use ADAM [45] for optimization of our loss function.

3.6 Resource Controller

Scaling workload and instances. The resource control module scale observed workload moderately to fit into the latency prediction model. Scaled workloads are fed into the configuration solver, which gives out optimal resource configuration as the output. The resource controller brings back the previous scaling process by multiplying the resource configuration from the configuration solver according to the magnitude of scaling done to the workload. With the processed resource configuration, the resource controller calculates the number of instances to scale in/out for every microservices.

$$Instances = Ceil(CPUquota/CPUunit) \quad (7)$$

The resource configuration from the configuration solver comes out as a combination of a real number (i.e. CPU quota). The number of instances is rounded up by an instance's CPU quota unit as represented in the equation 7. Finally, the scaling decisions are handed over to the Kubernetes cluster and each microservice's CPU resources are adjusted.

3.7 Sample Collector

Building a big enough training dataset is necessary to train GNN at predicting end-to-end percentile latency with high accuracy. Without simulators or pre-collected trace data, it is necessary to collect samples by interacting with real clusters which are time-consuming. GRAF makes use of the state-aware sample collector, which enables efficient sample collection for the training dataset. Unlike our resource allocation framework, during sample collection, we utilize other state information besides front-end information. Each microservice's current CPU usage, CPU utilization, tail-latency from 50%-tile to 99%-tile, workloads are observed by the state-aware sample collector. The heuristic algorithm caps sample collector from exploring unnecessary resource regions where it's too high that latency would no longer decrease or it's too low that the latency of a single microservice would violate latency SLO.

Algorithm 1 Reducing Search Space

```
Require: Individual microservice's CPU usage, tail-latency
procedure ReducedSearchSpace
    Initialize Microservices with sufficient CPU
    M = Set of every Microservices
    TL ← M.getTailLatency()
    for i ∈ M do
        Reset Microservices with sufficient CPU
        while True do
            Reduce CPU_i
            if M_i.getTailLatency() > TL_i then
                H_i = CPU_i                          ▷ Higher Bound
                Break
        while True do
            Reduce CPU_i
            if M_i.getTailLatency() > LatencySLO then
                L_i = CPU_i                          ▷ Lower Bound
                Break
    Return L, H
```

The sample collector collects samples from reduced search space identified by using Algorithm 1. An upper bound for each microservice is found by collocating large CPU resources for every microservices then reducing the target microservice's CPU resource step by step. The latency for each microservice has a lower bound due to the required minimal CPU cycles to handle a request. The upper bound is set when a decrease in CPU resource for the target microservice results in longer latency. On the other hand, a lower bound for each microservice is found by further reducing the target microservice's CPU resource step by step. If the latency of a single microservice exceeds the latency SLO which targets end-to-end latency of the microservices application, the corresponding CPU resource is set as the lower bound.

3.8 Proactive Resource Allocation

There are two major characteristics in GRAF, which enable proactive resource allocation according to front-end workload change. First, *Latency Prediction Model* only utilizes front-end workloads data, prior information (e.g., tracing data of each API workload), and CPU quota configuration. Therefore, before every microservices are already affected by traffic change at the front-end, *Latency Prediction Model* predicts expected end-to-end tail latency for the front-end workloads and possible CPU resource allocations. Second, the resource controller synchronously copes with the change of front-end workload and change of latency SLO demand. It's possible because our configuration solver is lightweight, using gradient descent optimization. Configuration solver finds out adequate resource configuration, without complex procedures such as training neural network every time or finding optimal configuration with heavy global optimization algorithms. In our experiments, it takes

Parameter	Value
Number of epoch	7×10^4
Batch size	256
Learning Rate	2×10^{-4}
Dropout Probability	0.25
Asymmetric hüber loss	θ_L (0.1), θ_R (0.3)

Table 1: Latency Prediction Model (§ 3.4) training parameters.

3.4-6.8s to calculate resource configuration when front-end workload and latency SLO are given. The proactive resource allocation makes GRAF tolerant to traffic surges, which will be demonstrated in the Evaluation chapter.

4 IMPLEMENTATION

To collect CPU usage and workload data for each microservice, we use Prometheus [26], Linkerd [22], and cAdvisor [17]. For tracing information about the microservice chain, we leverage Jaeger [20]. We implemented our machine learning model using Pytorch [50] and torch geometric [38] modules. Total 92K samples are collected from the microservice applications for training. The training and gradient descent optimization modules are implemented in 0.5K lines of Python code. The data collection and resource control module is implemented in 3.2K lines of Python code.

Training parameters. We implemented the message passing neural network and fully connected readout neural network using PyTorch [50] and torch geometric [38] modules. The MPNN has the input size of the node's features and contains two hidden layers with 20 hidden units, all using ReLU activation function. The output of MPNN for each node is flattened and fed toward a fully connected neural network for the readout phase. The fully connected (FC) neural network also contains two hidden layers with 120 hidden units, having ReLU activation function and one output dimension for the last layer. Dropout layers are applied to every layer except for the last layer for the generalization of the model during training. The input dimension of MPNN equals the number of node features and the input dimension of the FC neural network is linear to the number of nodes. Hyperparameters of the latency prediction model training are listed in Table 1.

5 EVALUATION

We evaluate GRAF to answer the following questions:

- How effectively does each design component in GRAF function?
- How much total CPU resources are saved by GRAF compared to Kubernetes autoscaler?
- How much better does GRAF handle traffic surge compared to existing autoscalers?

Experimental Setup. We evaluate GRAF using two open-source microservice applications: Online Boutique [25] and Social Network [40]. Graph representation of the controlled microservices for Online Boutique and Social Network is depicted in Figure 4 and 10, respectively. Kubernetes [21] is used for underlying container orchestration. We deploy Kubernetes clusters on 7 machines equipped

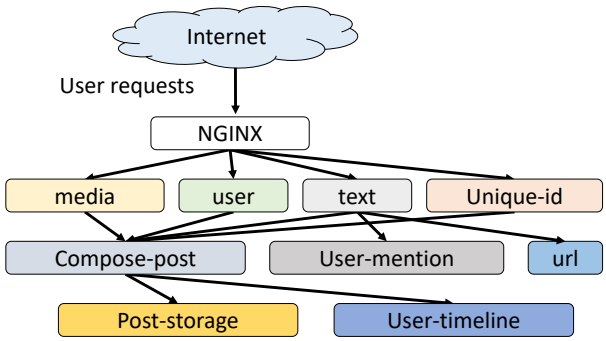


Figure 10: A microservice chain of a post-compose request in Social Network [40].

Latency Region	0-50ms	50-100ms	0-200ms	0-800ms	over-estimate
Percentage Error	21.3%	27.1%	27.1%	31.9%	5.2%

Table 2: Average absolute percentage error of model prediction according to sampled 99%tile-latency region.

with 2 Intel E5-2650 CPUs and 128GB of memory. We use a machine for the Kubernetes master node and the rest for the worker nodes. We use GeForce GTX1080 for training. We use Locust [23] and Vegeta [13] for load generation. The load generators generate workloads for sample collection and the creation of traffic surges. For the Social Network application, Vegeta generates post-compose requests. For the Online Boutique application, Locust generates workloads composed of three multi APIs. Load generation takes place in a separate machine from Kubernetes clusters.

Sample Collection and Training. A sample is collected in the procedure of applying resource configuration, generating load, collecting latency, and initialization. Random resource configuration in a range of Algorithm 1 is chosen and applied to the microservices of an application. After a load is generated, the latencies of requests are collected within 10 seconds time window which measures the percentile latency of a sample pair. Initialization is a 5 seconds process that flushes out possible existing requests queue for the next consecutive sample collections.

During the experiments, a latency prediction model is trained for each application. As the graph structure of microservices for each application is unchanged throughout the experiments, the model is trained once. The trained model is then used to reproduce every result in the evaluation without retraining. For training, 42k and 50k samples are collected for Social Network and Online Boutique respectively.

5.1 End-to-end Tail Latency Prediction

Latency prediction accuracy. Table 2 shows the results of *Latency Prediction Model*. Our collected samples are separated into the training, validation, and test sets. Accuracy of the latency prediction model is measured by having prediction upon test set which is unseen during training. As GRAF used percentage error in the loss function, the latency prediction model tends to have higher accuracy in the lower latency region. This behavior is intended because our focus is to satisfy latency SLO, which is usually set upon

low latency region. Also, Table 2 shows that the latency prediction model tends to overestimate about 5.2% upon overall data points. Over-estimate of latency allows GRAF to avoid resource configuration that might violate latency SLO. Although percentage error shows more than 20% error, this is inevitable due to the natural variance of 99%-tile latency.

Efficacy of GNN. We claim the effectiveness of GNN by comparing the learning curves of our GRAF neural network and GRAF neural network without MPNN. The trained model from GRAF showed better performance than the model from GRAF without MPNN as shown in Figure 11. On the other hand, GRAF without MPNN's training loss converged faster than GRAF sometimes achieving lower value during the training phase. To avoid overfitting of GRAF without MPNN, we tried different network sizes and introduced dropout layers but couldn't achieve a model that performs better than GRAF. While MPNN increased the ability to generalize over unseen data by successfully embedding neighbor's information, GRAF without MPNN tends to suffer from overfitting upon noisy samples.

Efficient Sample Collection. Naïve sample collector for microservices would explore every possible CPU resource combinations of microservices. State-aware sample collector utilizes the current state of microservices and latency characteristics to make efficient exploration. Each microservice's CPU utilization, latency, and input workloads are observed by the sample collector. With additional information, our sample collector does not explore the region where the CPU resource of a microservice is too low that leads to high tail latency, or region where CPU resource is too much that additional resource cannot reduce tail latency anymore. As the result, in the Online Boutique application, our sample collector exploration is done in 0.00027x reduced search space compared to the original search space.

5.2 Resources Optimization Analysis

Configuration solver. We show empirical results that support gradient descent optimization find optimal combinations in our experiments within a reasonable time. The gradient descent algorithm is a local optimization algorithm that converges fast but cannot guarantee to find the global optimum. However, gradient descent can find global optimal in microservices followed by their resource latency relationship. Note that in Figure 6, latency in each microservice is a monotonic decreasing function toward variable of CPU resources. The designed loss function is convex, only affected by the trade-off relationship among resources and latency. For the demonstration of empirical results, heatmap of loss values restricted to two resource variables is illustrated in Figure 12. Every microservices shows a similar latency characteristic which leads the gradient descent optimization to find global optimal resource combinations. CPU resource combinations found by GRAF actually utilize less total CPU resources compared to K8s autoscaler while meeting target latency SLO.

To verify resource combinations from GRAF are tightly minimized in terms of target tail latency, we measured 99%-tile latency after making allocation decisions according to the combinations. Figure 17 plots measured 99%-tile latency of resource combinations with respect to their target tail latency. 85.1% of resource combinations' actual tail latency fall in the boundary of targeted latency SLO, which implies our asymmetric hüber loss is taking effect as we wanted. In Figure 17, we can also witness that measured 99%-tile latency points are densely located near targeted tail latency, which means our configuration solver tightly minimizes total CPU resources in terms of latency SLO. Also, the gradient descent algorithm's 90%-tile latency to reach the target tolerance threshold takes about 6.7 seconds, fast enough to make resource allocation decisions synchronously in microservices.

Modules	AWS EC2 Instance	Time (h)	Budget ($)
Load Generator	CPU (c4.large)	208.3	20.83
Worker Node	CPU (c4.2xlarge)	208.3	82.92
Model Training	GPU (g4dn.xlarge)	16	8.42

Table 3: Expected budgets for collecting 50k samples and training the latency prediction model for Online Boutique.

Scaling workload. The resource controller scale down the workload moderately to fit in the region where GNN is trained. After, configuration solver finds resource combination, the resource controller scale up resource combination with the scaled ratio of the workload. This design is chosen under the assumption that workloads are evenly distributed among deployed instances. During the evaluation, GRAF shows consistent behavior under various workloads settings with the resource controller module. Figure 18 shows experiment results with varying simulated number of users in the online boutique application. GRAF's showed dominance over tuned K8s autoscaler, achieving the same tail latency while distancing the proportional amount of the saved instances as workload increase.

5.3 End-to-end Performance Evaluation

Resource saving. We show how much GRAF can save CPU resources compared to K8s autoscaler. Since K8s autoscaler has no functionality to make resource optimization according to latency SLO, we hand-tuned the resource utilization threshold of K8s autoscaler to meet latency SLO. One global resource utilization threshold is empirically found according to the latency SLO, and then applied to every microservice in the application.

As shown in Figure 15 and 16, GRAF saves 14-19% more total CPU resources at runtime compared to fine-tuned K8s autoscaler while achieving the same tail latency performance. GRAF finds optimal resource configuration by allocating more CPU resources to latency-sensitive microservices while saving from others. For example, GRAF allocates more CPU resources to MS5 (recommendation microservice) and MS6 (shipping microservice) and save from others compared to K8s autoscaler as depicted in Figure 15.

In addition, we conduct the cost-benefit analysis of GRAF for Online Boutique. Figure 19 shows profitable regions by applying GRAF according to microservices update period and workload magnitude. We calculate the cost for the sample collection and training of GRAF, and the benefit by following the pricing plan of AWS EC2 [15]. A total of $112.17 cost is expected for collecting 50k samples from Online Boutique application and training the latency prediction model. The specific details of instance type and borrowing time are illustrated in Table 3. Total time for collecting samples is driven

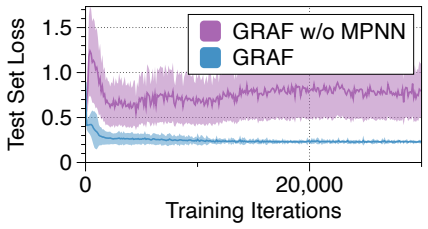


Figure 11: Learning curve comparison between GRAF and GRAF without MPNN.

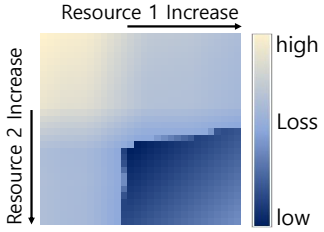


Figure 12: Heatmap of gradient descent optimization loss value according to two microservices' resources.

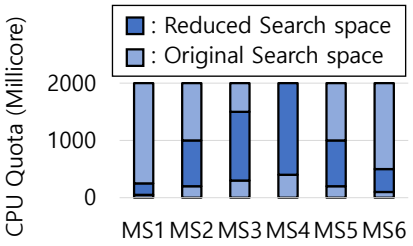


Figure 13: Reduced search space compared to orginal search space in Online Boutique application.

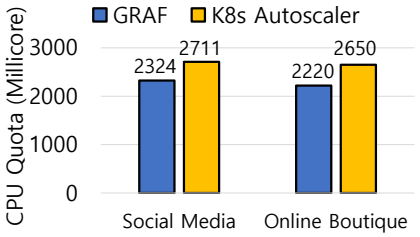


Figure 14: Total CPU resources when using GRAF and Kubernetes autoscaler according to target latency SLO.

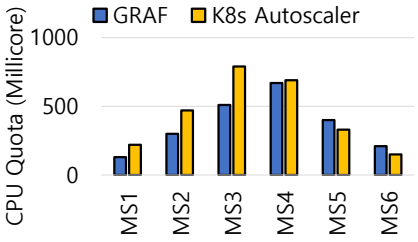


Figure 15: Online Boutique: Each microservice's CPU resource comparison when using GRAF and Kubernetes autoscaler.

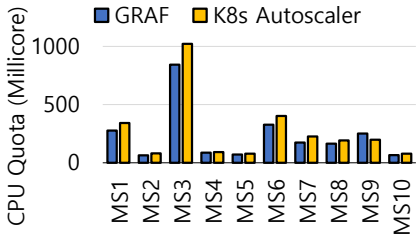


Figure 16: Social Network: Each microservice's CPU resource comparison when using GRAF and Kubernetes autoscaler.

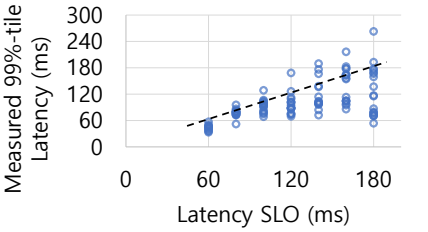


Figure 17: Measured tail latency of resource configuration targeting various latency SLO.

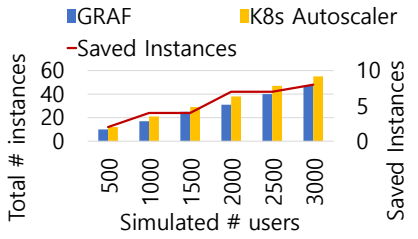


Figure 18: Total CPU instances saved by GRAF in the various workload settings.

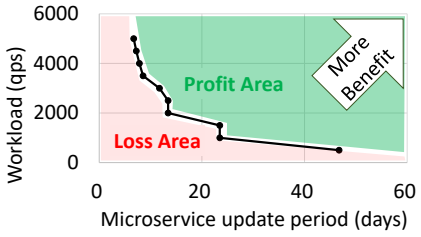


Figure 19: Cost-benefit analysis according to update period and workload magnitude.

by multiplying 50k samples and 15 seconds per sample. The model training time with GPU is referred from our actual time spent to train the model. Note that sample collection can be processed in parallel to save time while spending the same budget. Moreover, it is a one-time cost for the sample collection and training unless the microservices application is updated.

The expected profit is also calculated by converting the number of saved instances to the saved $ per day according to the pricing plan of AWS EC2. The more workloads an application handles, the more benefits one can get from GRAF as shown in Figure 18. Also, the less frequently a microservice is updated, the more benefits one receives because the cost is constant regardless of the workload.

Real workload demonstration. We demonstrate GRAF under Online Boutique [25] using real workload trace data in Figure 20. To the best of our knowledge, published granular time-series workload traffic data of web-service applications do not exist. Hence, we use AzurePublicDatasetV2 [56] which contains real-world time series data of functions invocations per minute. To convert functions invocations data to our experimental environment, the total number of functions invocations per minute is abstracted as the input workloads per minute. To mimic the real input workloads, the Locust [23] spawns the appropriate number of user threads at every minute. Both GRAF and K8s autoscaler achieve approximately 180ms 95%-tile latency measured by response time statistics of Locust API, while GRAF utilizes less number of total instances than K8s autoscaler during the most of 1900s demonstration time window.

Furthermore, we can observe GRAF makes scaling decisions according to the workload variation, efficiently scaling up and down immediately according to the frontend workload changes. On the other hand, K8s autoscaler suffers from cascading effects when workload increases and scale-downs slowly when the workload

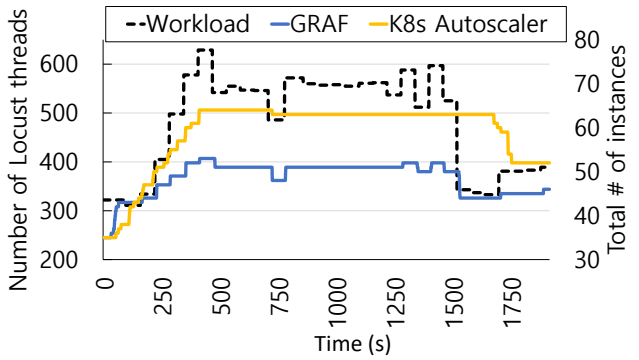


Figure 20: Total number of microservice instances under real time-series workload. Locust spawns threads according to the Azure function invocation data set [56].

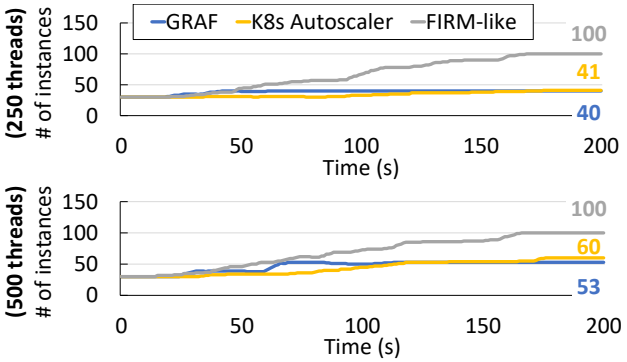


Figure 21: Total number of microservice instances when traffic surges. Each graph represents results when a load generator called Locust [23] spawns 250 threads and 500 threads respectively.

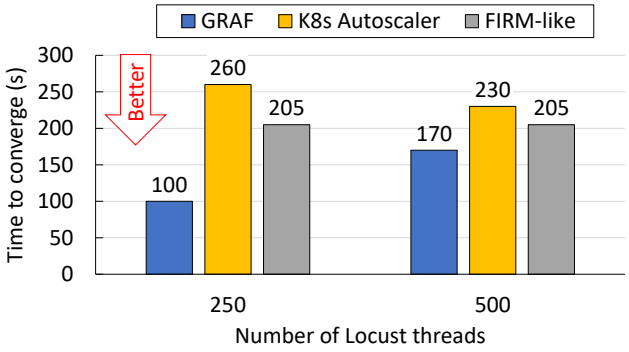


Figure 22: Time for end-to-end tail latency to converge after traffic surge.

decreases. In the default setting, K8s autoscaler records the scale recommendations of the past 5 minutes and chooses the highest one. This means that scale-downs occur gradually, smoothing out the impact of fluctuating CPU utilization metrics. Accordingly, after a sharp workload decrease at 1500s, K8s autoscaler scale down slowly after 5 minutes whereas GRAF makes scaling decisions according to the workload variation. As a result, GRAF utilizes 21% less number of net instances on average than K8s-autoscaler.

Handling traffic surge. We compare GRAF with K8s autoscaler and FIRM [53]-like algorithm when traffic surges using Online Boutique [25] application. As mentioned in Section 2.1, unlike GRAF, K8s autoscaler and FIRM do not consider the microservice chain in their scaling decision so they face the cascading effect when traffic surges. We implement FIRM-like algorithm that increases the CPU quota of a microservice when a ratio between median and 95%-tile latency for the microservice exceeds a pre-determined threshold. To generate traffic surge, we use Locust [23], which spawns multiple user threads, each of the users sends various types of requests in a predefined order. The Locust thread randomly waits for up to 5 seconds before it sends the next request to mimic the actual user behavior. To increase workload abruptly, we create from 250 to 500 Locust threads.

Figure 21 and 22 show the results. GRAF creates 13-60% less number of microservice instances while taking up to 2.6x faster time to achieve end-to-end tail latency settlement compared to existing approaches. In Figure 21, during the first 160 seconds the two methods incrementally increase the number of instances while GRAF creates the required instances concurrently at around 50 seconds.

6 DISCUSSION AND FUTURE WORK

Scalability of GRAF. GRAF has complete scalability toward handling multiple chains of different request types at once. This is possible due to our workload analyzer and node embedding procedure in the system design. Workload analyzer analyzes frontend multiple APIs workloads and distributes information to graph nodes states. Then, the node embedding procedure captures the state of a graph and predicts the end-to-end tail latency of the overall microservices application. Also, GRAF is scalable toward the size of the workload as the resource controller handles the size of input fed into the configuration solver.

However, the latency prediction model of GRAF isn't scale-free to the microservices size. To meet the goal of satisfying end-to-end tail latency and resource optimization of overall microservices, GRAF's design incorporates every microservices' state concurrently. Accordingly, the readout phase's neural network input node dimension is linearly dependent on the number of microservices in an application. Although GRAF has the capability in that it easily copes with decent size microservices including Social Network, GRAF's performance may degrade when applied to applications composed of hundreds to thousands of microservices. Graph partitioning algorithms might reduce the burden on the latency prediction model's scalability by partitioning the microservices and training separately.

Actively removing contention anomalies. GRAF optimizes resource usage for given traffic while satisfying latency SLO. However, microservices sometimes suffer from latency SLO violations even when the proper amount of resources are allocated. It is frequently observed that latency spikes occur in microservices caused by unexpected contention in resources. Overprovisioning resources for microservices is a possible option that would probably enhance the ability to prevent unexpected latency spikes, therefore reducing SLO violations caused by anomalies. Unfortunately, this approach conflicts with GRAF, since GRAF tries to minimize as many resources as possible within the target SLO latency. An algorithm

that actively removes contentions in microservices should take place in order to fully utilize the capabilities of GRAF while meeting SLO latency at all times.

Integer Optimization for instances scaling. Latency prediction model and resource configuration solver in GRAF works with real numbers. State input features for the latency prediction model are not an integer, nor the optimal configuration found with gradient descent algorithm. On the other hand, in the microservices environment, the CPU resources are scaled by the number of instances, where a CPU unit is pre-determined for each instance. In this paper, we have rounded up the real numbers from GRAF modules to the number of instances. As the result, GRAF is overprovisioning resources in every microservices, yet bounded by the CPU resource unit for an instance. Although integer optimization is a well-known NP-hard problem, if one can perform integer optimization considering an instance's CPU unit, there is slight improvement room for GRAF to save more resources.

7 RELATED WORKS

Autoscaling. Numerous works apply autoscaling to cloud applications [41, 46, 53, 55, 62, 63]. To the best of our knowledge, none of the existing autoscalers have the ability to optimize the resources of an entire microservice application targeting adjustable latency SLOs. Also, they suffer from traffic surges due to cascading effect. FIRM [53] is a reinforcement learning (RL)-based resource management framework. FIRM uses the support vector machine to identify microservices critical to latency SLO violations and adjust multiplex resources with the deep deterministic policy gradient algorithm. FIRM focuses on finding and removing contention in the microservice that is the critical cause of latency SLO violation. Although FIRM can achieve latency SLO, it does not handle subsequent microservices in the chain which possibly lead to falling in sub-optimal states. MIRAS [62] is another RL-based autoscaler. MIRAS learns a policy that behaves to allocate more resources to the microservices with longer request queues. However, MIRAS does not target end-to-end tail latency and does not consider the influence of other microservices with short request queues. ATOM [41] is a model-driven autoscaler that runs a genetic algorithm over queuing model. ATOM adjusts resources for microservices at once considering the entire structure, but they optimize the system in the perspective of throughput. Throughput can be a candidate of service level indicator, however, tail-latency and availability are key indicators for latency-sensitive applications. RAMBO [46] leverages multi-objective Bayesian optimization to allocate resources and meet performance/cost goals. RAMBO chooses a design that achieves sample efficiency from an enormous search space. RAMBO claims to explore and find Pareto optimal points in performance and cost relationship, where performance is chosen as the maximum throughput under latency constant. Considering the traffic to microservice applications is often an uncontrollable external factor, RAMBO yet faces a lack of design to automatically find optimums under dynamic traffic workload.

Container startup latency. The main cause that develops cascading effect of microservices into a severe problem is container startup latency. S. Fu et al [39] suggest a scheduling scheme to reduce container startup time by utilizing dependencies between layers of the container images. S. Fu et al [39]'s scheme has been adopted by Kubernetes, but Kubernetes still suffers from a slow startup time of about 15 seconds in 90%-tile latency. Slacker [43] reduces container startup latency by lazily pulling the container images. Yet, Slacker can not be easily deployed because of requirements to use a proprietary NFS server and modify the Linux kernel. There are other approaches [1, 44, 61] to reduce container startup delay by reusing containers. The side effect of these approaches is an expensive cost in cloud services [39] due to excessive use of resources such as memory [64].

Combinatorial Optimization. Combinatorial optimization[49] problems have been explored in many research for decades. Bayesian optimization [51], one of the de-facto standards in the area of global optimizations, has shown its capability of sample-efficient optimization in non-convex, and heterogeneous functions [29, 32, 57, 59]. Recent researches improve scalability, handle constraints, and multi-objective with Bayesian optimization [28, 37], making the algorithms more practical to combinatorial optimization. The reinforcement learning approach also has been widely investigated in the use of solving combinatorial optimization problems [30, 31, 33, 36]. However, off-the-shelf optimization algorithms can not fit into microservices' resource optimization problems because of constantly changing external factors (e.g., workload) in the environment, narrow decision-making time window, and costly or even fatal exploration. GRAF tackles the combinatorial optimization problem in microservices by using asynchronous latency model training, and fast convergence algorithm. GRAF utilizes the convexity of the loss function achieved in nature of microservice latency characteristics, model-based differential loss function, and integration of the constraints into loss function.

8 CONCLUSION

We present GRAF, a GNN-based proactive resource allocation framework that minimizes overall CPU resources in the microservice chain while satisfying latency SLO. Despite the wide usage of microservices, existing resource allocation algorithms do not interpret the microservice chain which results in sub-optimal behaviors. Our main objective is to present an efficient microservices resources allocation framework that considers the characteristics of the microservice chain. GRAF utilizes GNN and gradient descent algorithm to make CPU resource allocation for every microservices considering the internal relationship of the microservice chain. We show the effectiveness of our approach by greatly outperforming existing autoscalers in end-to-end evaluations. By applying GRAF, service providers can minimize the total usage of CPU resources while satisfying latency SLO without additional human interventions.

ACKNOWLEDGMENTS

We thank our shepherd Marco Canini and anonymous reviewers for their valuable comments that improved the paper. We appreciate Youngmok Jung and Yechan Kim for their thorough feedback on writing. This work was partially supported by Toyota Motor Corporation.

REFERENCES

[1] 2014. Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/.

[2] 2015. Microservices at Amazon. https://www.slideshare.net/apigee/i-love-apis-2015-microservices-at-amazon-54487258.

[3] 2017. Airbnb, From Monolith to Microservices: How to Scale Your Architecture. https://www.youtube.com/watch?v=N1BWMW9NEQc.

[4] 2018. Examples and types of microservices. https://www.itrelease.com/2018/10/examples-and-types-of-microservices/.

[5] 2018. Horizontal Pod Autoscaler in AWS. https://docs.aws.amazon.com/eks/latest/userguide/horizontal-pod-autoscaler.html.

[6] 2018. Stan's Robot Shop by Instana. https://www.instana.com/blog/stans-robot-shop-sample-microservice-application/.

[7] 2019. Borg cluster workload traces. https://github.com/google/cluster-data.

[8] 2019. Cloud Waste To Hit Over 14 Billion in 2019. https://devops.com/cloud-waste-to-hit-over-14-billion-in-2019/.

[9] 2019. Multi-Tenancy Kubernetes on Bare Metal Servers. https://deview.kr/data/deview/2019/presentation/[231]+Multi-Tenancy+Kubernetes+on+Bare+Metal+Servers.pdf (16p).

[10] 2020. Microservice architecture growing in popularity, adopters enjoying success. https://www.itproportal.com/news/microservice-architecture-growing-in-popularity-adopters-enjoying-success/.

[11] 2020. Microservices Adoption in 2020. https://www.oreilly.com/radar/microservices-adoption-in-2020/.

[12] 2020. The Story of Netflix and Microservices. https://www.geeksforgeeks.org/the-story-of-netflix-and-microservices/.

[13] 2020. Vegeta: a versatile HTTP load testing tool. https://github.com/tsenart/vegeta.

[14] 2020. Wasted Cloud Spend to Exceed 17.6 Billion in 2020, Fueled by Cloud Computing Growth. https://jaychapel.medium.com/wasted-cloud-spend-to-exceed-17-6-billion-in-2020-fueled-by-cloud-computing-growth-7c8f81d5c616.

[15] 2021. Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[16] 2021. Bookinfo Application by Istio. https://istio.io/latest/docs/examples/bookinfo/.

[17] 2021. cAdvisor software on Github. https://github.com/google/cadvisor.

[18] 2021. Configuring horizontal Pod autoscaling in GKE. https://cloud.google.com/kubernetes-engine/docs/how-to/horizontal-pod-autoscaling.

[19] 2021. Horizontal Pod Autoscaler of Kubernetes. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[20] 2021. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/.

[21] 2021. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/.

[22] 2021. Linkerd: The world's lightest, fastest service mesh. https://linkerd.io/.

[23] 2021. Locust: An open source load testing tool. https://locust.io/.

[24] 2021. Microservices - Netflix Techblog. https://netflixtechblog.com/tagged/microservices.

[25] 2021. Online Boutique by Google. https://github.com/GoogleCloudPlatform/microservices-demo.

[26] 2021. Prometheus - Monitoring system & time series database. https://prometheus.io/.

[27] 2021. Scale applications in Azure Kubernetes Service (AKS). https://docs.microsoft.com/en-us/azure/aks/tutorial-kubernetes-scale.

[28] Setareh Ariafar, Jaume Coll-Font, Dana Brooks, and Jennifer Dy. 2019. ADMMBO: Bayesian Optimization with Unknown Constraints using ADMM. *Journal of Machine Learning Research* 20, 123 (2019), 1–26. http://jmlr.org/papers/v20/18-227.html

[29] Maximilian Balandat, Brian Karrer, Daniel R Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. 2019. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. *arXiv preprint arXiv:1910.06403* (2019).

[30] Thomas Barrett, William Clements, Jakob Foerster, and Alex Lvovsky. 2020. Exploratory combinatorial optimization with reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 3243–3250.

[31] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. 2016. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940* (2016).

[32] Eric Brochu, Vlad M Cora, and Nando De Freitas. 2010. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599* (2010).

[33] Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre Cire. 2020. Combining reinforcement learning and constraint programming for combinatorial optimization. *arXiv preprint arXiv:2006.01610* (2020).

[34] Byungkwon Choi, Jinwoo Park, Chunghan Lee, and Dongsu Han. 2021. pHPA: A Proactive Autoscaling Framework For Microservice Chain. In *5th Asia-Pacific Workshop on Networking (APNet 2021)*. Association for Computing Machinery, Inc.

[35] Chris Jones,John Wilkes,Niall Murphy,Cody Smith. [n.d.]. Service Level Objectives. https://sre.google/sre-book/service-level-objectives/.

[36] Hanjun Dai, Elias B Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665* (2017).

[37] David Eriksson and Matthias Poloczek. 2021. Scalable Constrained Bayesian Optimization. In *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 130)*, Arindam Banerjee and Kenji Fukumizu (Eds.). PMLR, 730–738. http://proceedings.mlr.press/v130/eriksson21a.html

[38] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[39] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. 2020. Fast and efficient container startup at the edge via dependency scheduling. In *3rd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 20)*.

[40] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. 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*. 3–18.

[41] Alim Ul Gias, Giuliano Casale, and Murray Woodside. 2019. ATOM: Model-driven autoscaling for microservices. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1994–2004.

[42] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 1263–1272. http://proceedings.mlr.press/v70/gilmer17a.html

[43] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Slacker: Fast distribution with lazy docker containers. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 181–195.

[44] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. 445–451.

[45] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[46] Qian Li, Bin Li, Pietro Mercati, Ramesh Illikkal, Charlie Tai, Michael Kishinevsky, and Christos Kozyrakis. 2021. RAMBO: Resource Allocation for Microservices Using Bayesian Optimization. *IEEE Computer Architecture Letters* 20, 1 (2021), 46–49. https://doi.org/10.1109/LCA.2021.3066142

[47] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. 2018. Combinatorial optimization with graph convolutional networks and guided tree search. *arXiv preprint arXiv:1810.10659* (2018).

[48] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. 270–288.

[49] Christos H Papadimitriou and Kenneth Steiglitz. 1998. *Combinatorial optimization: algorithms and complexity*. Courier Corporation.

[50] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[51] Martin Pelikan, David E Goldberg, Erick Cantú-Paz, et al. 1999. BOA: The Bayesian optimization algorithm. In *Proceedings of the genetic and evolutionary computation conference GECCO-99*, Vol. 1. Citeseer, 525–532.

[52] Issaret Prachitmutita, Wachirawit Aittinonmongkol, Nasoret Pojjanasuksakul, Montri Supattatham, and Praisan Padungweang. 2018. Auto-scaling microservices on IaaS under SLA with cost-effective framework. In *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI)*. 583–588. https://doi.org/10.1109/ICACI.2018.8377525

[53] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu

[54] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the third ACM symposium on cloud computing*. 1–13.

[55] Krzysztof Rzadca, Paweł Findeisen, Jacek Świderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Paweł Krzysztof Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: Workload Autoscaling at Google Scale. In *Proceedings of the Fifteenth European Conference on Computer Systems*. https://dl.acm.org/doi/10.1145/3342195.3387524

[56] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[57] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. *arXiv preprint arXiv:1206.2944* (2012).

[58] Jacopo Soldani, Damian Tamburri, and Willem-Jan Heuvel. 2018. The Pains and Gains of Microservices: A Systematic Grey Literature Review. *Journal of Systems and Software* 146 (09 2018). https://doi.org/10.1016/j.jss.2018.09.082

[59] Jost Tobias Springenberg, Aaron Klein, Stefan Falkner, and Frank Hutter. 2016. Bayesian optimization with robust Bayesian neural networks. *Advances in neural information processing systems* 29 (2016), 4134–4142.

[60] Arunchandar Vasan, Anand Sivasubramaniam, Vikrant Shimpi, T. Sivabalan, and Rajesh Subbiah. 2010. Worth their watts? - an empirical study of datacenter servers. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–10. https://doi.org/10.1109/HPCA.2010.5463056

[61] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 133–146.

[62] Z. Yang, P. Nguyen, H. Jin, and K. Nahrstedt. 2019. MIRAS: Model-based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 122–132. https://doi.org/10.1109/ICDCS.2019.00021

[63] Guangba Yu, Pengfei Chen, and Zibin Zheng. 2019. Microscaler: Automatic scaling for microservices with an online learning approach. In *2019 IEEE International Conference on Web Services (ICWS)*. IEEE, 68–75.

[64] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. 2019. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–10.