Graph Neural Network-Based SLO-Aware Proactive Resource Autoscaling Framework for Microservices

Jinwoo Park, Byungkwon Choi, Chunghan Lee, *Member, IEEE*, and Dongsu Han, *Member, IEEE*

***Abstract*— Microservice is an architectural style widely adopted in various latency-sensitive cloud applications. Similar to the monolith, autoscaling has attracted the attention of operators for managing the 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 SLO-aware proactive resource autoscaling 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 the 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. GRAF also handles a traffic surge with 36% fewer resources while achieving up to 2.6x faster tail latency convergence compared to the Kubernetes autoscaler. Moreover, we verify the scalability of GRAF on large-scale deployments, where GRAF saves 21.6% and 25.4% for CPU resources and memory resources, respectively.**

***Index Terms*— Microservices, resources optimization, graph neural networks, applied machine learning, cloud computing, autoscaler.**

I. 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 [2], [3] 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 [4]. 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 [5], Amazon [6], and Airbnb [7]. Considering applications' purpose and target, indicators that represent user experiences such as availability and tail latency are often set as service level objectives (SLOs) [8]. Latency SLO in these microservices applications is especially important. Therefore, many operators choose to overprovision resources on clouds [9], [10], [11]. Unfortunately, overprovisioning comes with the cost, about \$6.6 billion are wasted in the cloud because of overprovisioning [12], [13], [14]. 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 § II-A), which severely degrade microservices performance when traffic surges. The root causes of cascading effect are; inevitable deployment overhead (i.e., 15s startup latency [15]), 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 [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27] 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 and Sinan [17], [24], yet FIRM falls in sub-optimal configuration (discussed in § V-C) and SINAN exploits only subset of potential resource allocations during online deployment. Existing autoscalers target objectives such as CPU utilization or performance (e.g., throughput, profit/cost). For example,

Manuscript received 21 April 2023; revised 25 September 2023 and 28 February 2024; accepted 25 March 2024; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor P. P. C. Lee. Date of publication 3 May 2024; date of current version 20 August 2024. This work was supported in part by the Institute for Information & Communications Technology Promotion (IITP) funded by Korean Government (Ministry of Science and ICT) under Grant 2024-00398157. This submission is an extended version of a CoNEXT 2021 paper [DOI: 10.1145/3485983.3494866]. *(Corresponding author: Dongsu Han.)*

Jinwoo Park, Byungkwon Choi, and Dongsu Han are with the School of Electrical Engineering, Korea Advanced Institute of Science Technology, Daejeon 34141, South Korea (e-mail: jinwoo520528@kaist.ac.kr; nfsp3k@gmail.com; dhan.ee@kaist.ac.kr).

Chunghan Lee is with Toyota Motor Corporation, Tokyo 100-0004, Japan (e-mail: lch@toyota-tokyo.tech).

Digital Object Identifier 10.1109/TNET.2024.3393427

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 [16], [17], [18], [19], [21] control resources of each microservice individually, which cannot 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 the possible combinations across every microservices within a 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 SLO-aware proactive resource autoscaling framework for 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. Last, GRAF provides an effective additional resource-distributing functionality for better robustness. GRAF estimates the end-to-end performance impact of temporal instance failures at each microservice and then distributes additional resources accordingly.

To the best of our knowledge, GRAF is the first work to optimize microservice resources proactively addressing dynamic frontend workloads 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.

II. 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 [27]. Specifically, we observed that they experience a phenomenon called cascading effect and severely degrade the performance when traffic changes abruptly (§ II-A) and also inefficiently optimize resources for microservices (§ II-B). In this section, we describe our observations in detail and investigate the opportunity behind them to effectively optimize resources for microservices.

A. 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 VII, most existing autoscalers do not consider the microservice chain and face the cascading effect. We describe the cascading effect based on K8s autoscaler [31] that is widely used [11], [32], [33], [34].

K8s autoscaler operates with a pre-determined resource utilization threshold. When a microservice's resource utilization

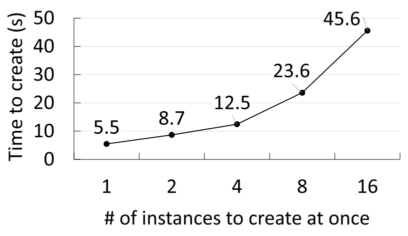


Fig. 1. Time to create microservice instances.

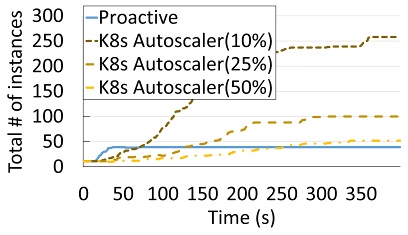


Fig. 2. Total number of microservice instances when traffic surges.

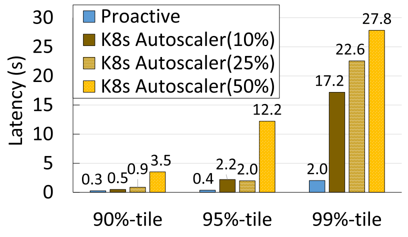


Fig. 3. End-to-end latency when traffic surges.

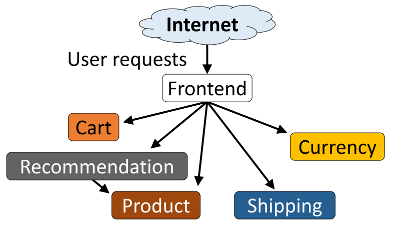


Fig. 4. A microservice chain of an open source benchmark called online boutique [28].

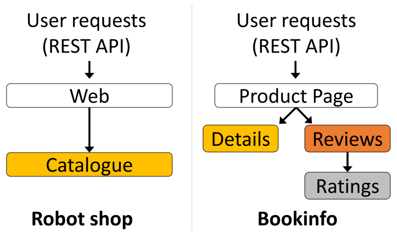


Fig. 5. Microservice chain of robot shop [29] and Bookinfo [30].

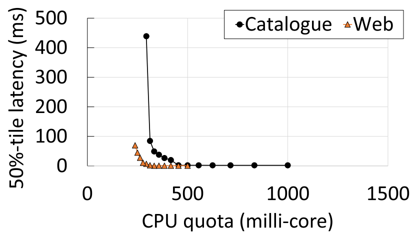


Fig. 6. Latency curves of each microservice in online boutique application.

reaches the threshold, the autoscaler creates more instances of the microservice to keep the utilization below a certain level.[1] The autoscaler monitors the usage of the resources such as CPU and memory to independently create instances for each microservice.

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 [28]. 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.

Experimental result. 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 [35]. 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 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, we can develop a resource allocation framework that avoids the cascading effect.

[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 [21].

[2]We create instances of microservices in [28] on a single worker node and ignore the network delay to download the container images.

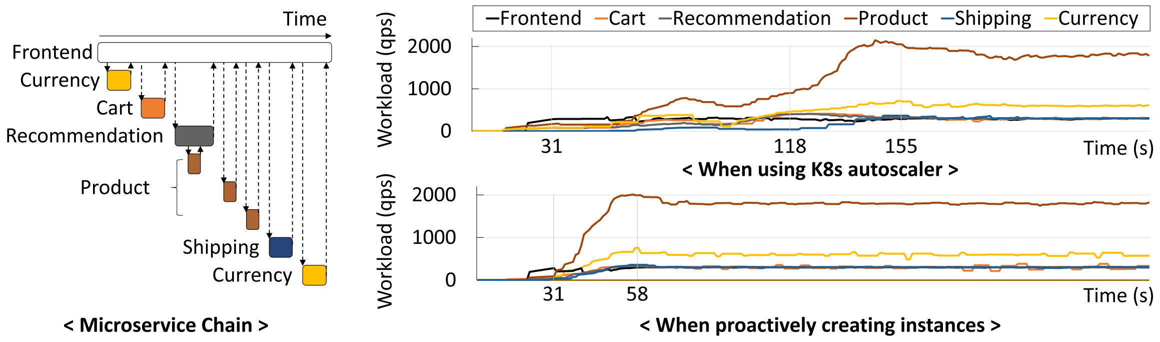


Fig. 7. Microservice chain and workload that each microservice perceives when traffic surges.

B. Microservice Latency Characteristics

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 [29] 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 [30]. '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 nontrivial. In our case, it is feasible because the target function has convexity as latency of every microservice monotonically decreases.

III. 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, 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 to hundreds [36], [37] 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 [38], [39], [40].

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) [41] with edge information constructed from microservices tracing data. For proactive resource allocation, we precisely restricted state features of our GNN model to use only available information at the frontend. 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.

A. GRAF Overview

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

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

2. Workload analyzer (§ III-C), analyzes front-end workload into distributed workloads to each microservices in the system. Such distributed workloads capture the state of the

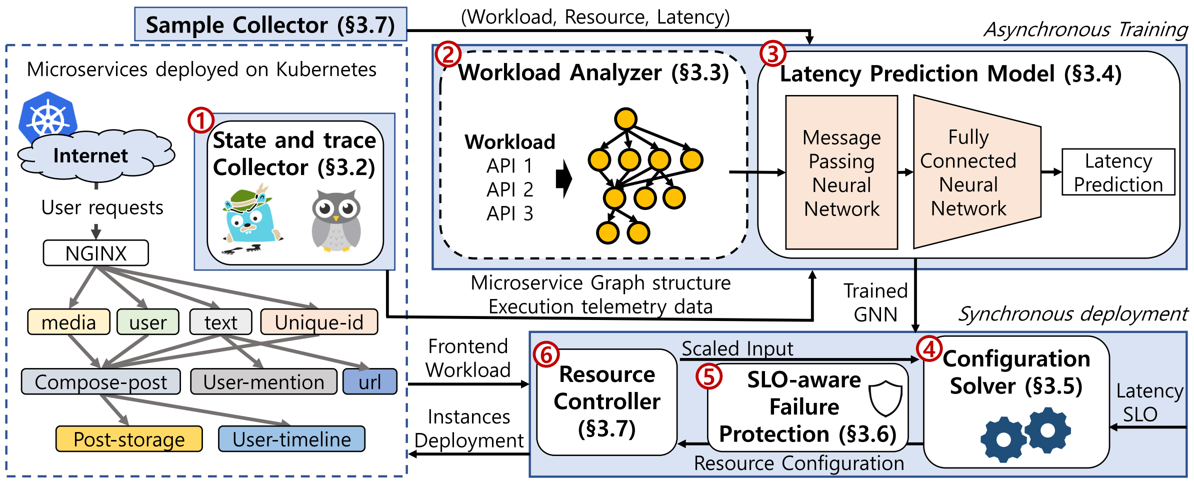


Fig. 8. GRAF: Design overview.

graph, representing a microservices dependency. The reshaped workload is fed to the latency prediction model as an input feature.

3. Latency prediction model (§ III-D) 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 (§ III-E) 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, SLO-aware failure protection (§ III-F) analyzes the end-to-end performance impact for each microservice in case of temporal instance failures, and then effectively distributes resources within a predetermined budget. Finally, the resource controller (§ III-G) calculates corresponding instances for each microservice and makes a scaling decision to the cluster.

5. For the training process, the sample collector (§ III-H) 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 is further explained in detail throughout the following sections.

B. State and Trace Collector

The state and trace collector monitors the current states of microservices and collects the history of trace data. cAdvisor [42] is used to monitor each microservice's current CPU usage and CPU utilization. Jaeger [43] 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.

C. Workload Analyzer

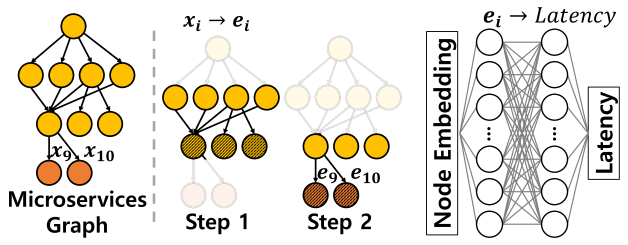
Capturing the microservice's graphical features. The state collector observes the frontend workload for each API, which we have annotated as $\vec{w}$. The workload analyzer gets frontend workloads as input and output distribution of workloads for each microservice in the application. With a distributed tracing called Jaeger [43], the workload analyzer collects the request history of 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 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.

D. 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.

Graph neural network. Our GNN is composed of MPNN [41] 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



(a) Node Embedding (b) Readout

Fig. 9. Graph neural network composed of node embedding and readout phase to predict latency.

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. $N(i)$ 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 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.

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

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.

E. 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.

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

$$\phi(x, y) = max(0, x - y) \tag{6}$$

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 (§ V-B). We use ADAM [44] for optimization of our loss function.

F. SLO-Aware Failure Protection

Until now, GRAF aims to minimize resources to the limit as long as it meets the target SLO. While it is very effective at achieving the original goal of resource-saving, temporal instance failures expose microservices to possible immediate SLO violations. We additionally design an SLO-aware failure protection system, which aims to prevent possible SLO violations caused by temporal instance failures at critical microservices. The goal of this component is to estimate each microservice's vulnerability toward instance failures and effectively distribute resources within a predetermined budget. We analyze each microservices by the expected severity of SLO violations when they suffer temporal instance failures. We distribute resources proportional to the expected severity of SLO violations between microservices under instance failures. The resource budget is given as a hyperparameter that the operator can decide.

SLO-critical services analysis. GRAF estimates the magnitude of end-to-end percentile latency elongation in case of temporal instance failures by utilizing the latency prediction model (§ III-D). To analyze each microservice's vulnerability toward instance failures, we feed the latency prediction model with the resource configuration result of (§ III-E) but with fewer resources at the microservice. By comparing the outputs of the latency prediction model with input of the original resource configuration and with input of the resource configuration assuming instance failures at the microservice, we can estimate the elongated end-to-end percentile latency when instance failures take place. Recall that our latency prediction model outputs end-to-end tail latency with the input of every microservice's node features, composed of workload and allocated CPU resources.

Fault tolerant resource preparation. Given the predetermined resource budget, we distribute the resources to each microservice according to the magnitude of the negative impact on end-to-end percentile latency followed by instance failure at the microservice. Specifically, we distribute resources among microservices following the ratio of the result below for the microservice m.

$$L(\vec{w}, \vec{r}) - L(\vec{w}, \vec{r} - r_m) \tag{7}$$

where, L, $\vec{w}$, $\vec{r}$, and r_m represent the latency prediction model, workload, resources configuration, and temporal instance failure at microservice m, respectively. The degree of instance failure is a hyperparameter controlled by the operator, presuming the number of failed instances. The distributed resources configuration is delivered to the resource controller to be added to the final scaling decisions.

G. 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 microservice.

$$Instances = Ceil(CPUquota/CPUunit) \tag{8}$$

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 8. In the presence of additional resources budget for SLO-aware Failure Protection, the result of additional instances configuration is added to the final result. Finally, the scaling decisions are handed over to the Kubernetes cluster and each microservice's CPU resources are adjusted.

H. Sample Collector

Building a big enough training dataset is necessary to train GNN for 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.

The sample collector collects samples from the reduced search space identified below. An upper bound for each microservice is found by collocating large CPU resources for every microservice and 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 the end-to-end latency of the microservices application, the corresponding CPU resource is set as the lower bound. Note that, the latency SLO threshold for the sample collecting process can be set with an extra margin to train the latency prediction model in broader latency regions considering various latency SLO requirements.

IV. Implementation

To collect CPU usage and workload data for each microservice, we use Prometheus [45], Linkerd [46], and

TABLE I
LATENCY PREDICTION MODEL (§ III-D) TRAINING PARAMETERS

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)

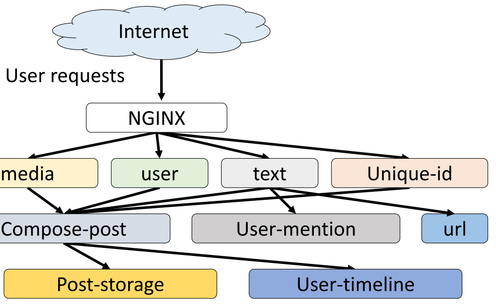


Fig. 10. A microservice chain of a post-compose request in social network [49].

cAdvisor [42]. For tracing information about the microservice chain, we leverage Jaeger [43]. We implemented our machine learning model using Pytorch [47] and torch geometric [48] 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 [47] and torch geometric [48] 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 I.

V. 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 [28] and Social Network [49]. Graph representation of the controlled microservices for Online Boutique and Social Network is depicted in Figure 4 and 10, respectively. Kubernetes [16] is used for underlying container orchestration. We deploy Kubernetes clusters on 7 machines equipped 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 [50] and Vegeta [35] 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. 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.

TABLE II
AVERAGE ABSOLUTE PERCENTAGE ERROR OF MODEL PREDICTION ACCORDING TO SAMPLED 99%TILE-LATENCY REGION

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

A. End-to-End Tail Latency Prediction

Latency prediction accuracy. Table II 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 II 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 to GRAF neural network without MPNN and GRAF neural network with CNN. GRAF neural network with CNN is the model made by replacing MPNN layers with CNN layers. The trained model from GRAF showed better performance than the model from GRAF without MPNN and GRAF with CNN as shown in Figure 11. On the other hand, GRAF without MPNN's training loss converged faster than GRAF

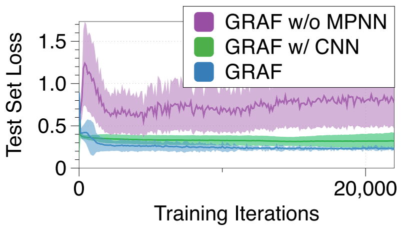


Fig. 11. Learning curve comparison between GRAF without MPNN, GRAF with CNN, and original GRAF.

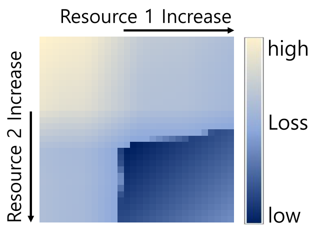


Fig. 12. Heatmap of gradient descent optimization loss value according to two microservices' resources.

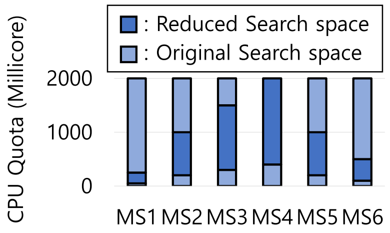


Fig. 13. Reduced search space compared to orginal search space in online boutique application.

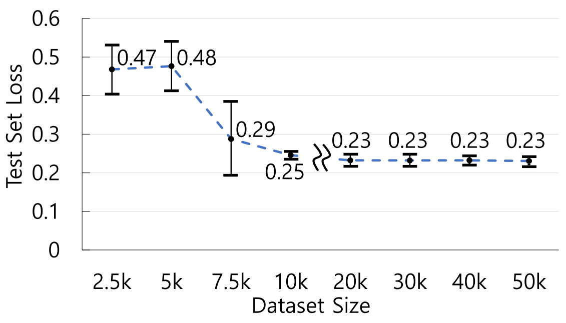


Fig. 14. Model performance according to the size of dataset used for training.

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 performed 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. The importance of the model's ability to interpret neighbor's information can be further seen through evaluating the learning curve of GRAF with CNN. GRAF with CNN achieves better performance than GRAF without MPNN as it incorporates adjacent neighbors' information through CNN. However, GRAF still shows more stable performance while training and better performance by achieving 32.3% smaller test set loss than GRAF with CNN. This is because MPNN better utilizes neighbors' information by explicitly following the application's microservice graph structure during the node embedding phase.

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 a result, in the Online Boutique application, our sample collector exploration is done in 0.00027x reduced search space compared to the original search space.

Model performance toward dataset size. To evaluate the number of samples to train our latency prediction model, we measure the model performance while varying the dataset size. From 50k samples, 6k samples are randomly chosen to make up for the test dataset, and then the rest of the samples are randomly chosen to meet the desired size of dataset. Within the dataset, we split the dataset into the training set and the validation set in proportion of 0.8 and 0.2, respectively. Note that, for 50k data point, only 44k are used for training as the remaining 6k data are designated as the test set. During the training, we checkpoint the model that shows the best performance with the validation set and later use the model for performance evaluation in the test set. The model performance according to the size of the dataset is shown in Figure 14, we plot the average test set loss and the error bar indicates the standard deviation. We observe that the model starts to achieve almost maximum prediction performance starting from 7.5k data samples although the model performance varies a lot for each trial. In Online Boutique application, the latency prediction model is stably trained with more than 20k samples used for the training.

B. 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

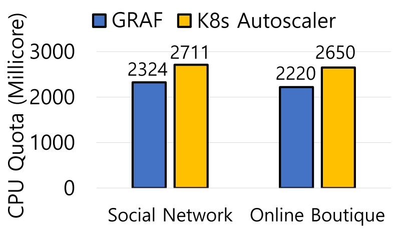


Fig. 15. Total CPU resources when using GRAF and Kubernetes autoscaler according to target latency SLO.

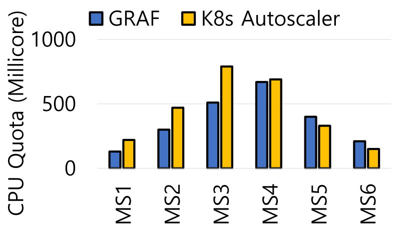


Fig. 16. Online boutique: Each microservice's CPU resource comparison when using GRAF and Kubernetes autoscaler.

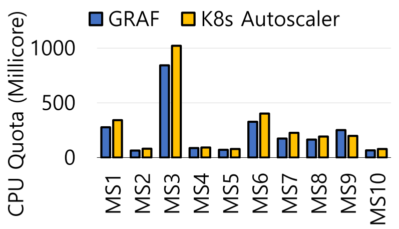


Fig. 17. Social network: Each microservice's CPU resource comparison when using GRAF and Kubernetes autoscaler.

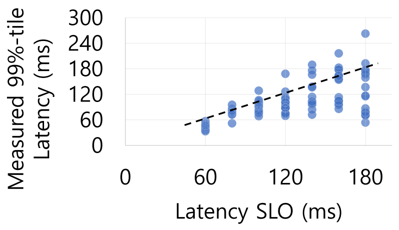


Fig. 18. Measured tail latency of resource configuration targeting various latency SLO.

the combinations. Figure 18 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 18, 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.

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

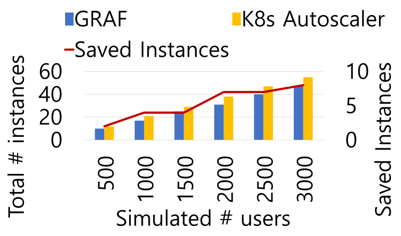


Fig. 19. Total CPU instances saved by GRAF in the various workload settings.

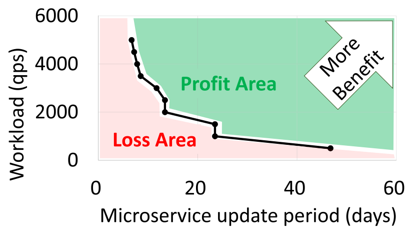


Fig. 20. Cost-benefit analysis according to update period and workload magnitude.

TABLE III
EXPECTED BUDGETS FOR COLLECTING 50K SAMPLES AND TRAINING THE LATENCY PREDICTION MODEL FOR ONLINE BOUTIQUE

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

with the resource controller module. Figure 19 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.

C. 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 16 and 17, 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 saves from others compared to K8s autoscaler as depicted in Figure 16.

In addition, we conduct the cost-benefit analysis of GRAF for Online Boutique. Figure 20 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 [51]. A total of $112.17 cost is expected for collecting 50k samples from Online Boutique

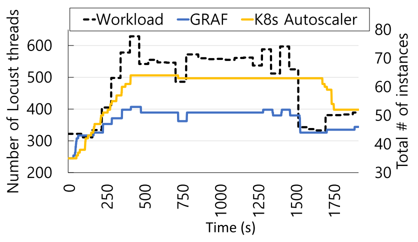


Fig. 21. Total number of microservice instances under real time-series workload. Locust spawns threads according to the Azure function invocation data set [52].

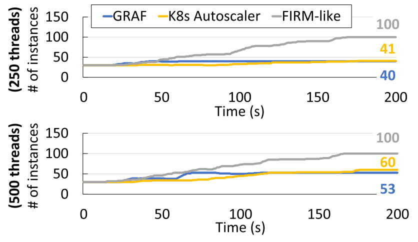


Fig. 22. Total number of microservice instances when traffic surges. Each graph represents the results when a load generator called Locust [50] spawns 250 threads and 500 threads respectively.

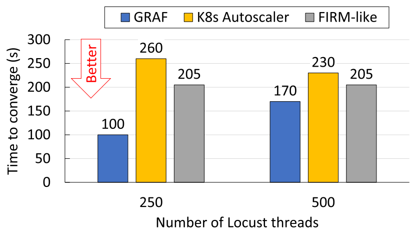


Fig. 23. Time for end-to-end tail latency to converge after traffic surge.

application and training the latency prediction model. The specific details of instance type and borrowing time are illustrated in Table III. The total time for collecting samples is driven 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 19. 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 [28] using real workload trace data in Figure 21. To the best of our knowledge, published granular time-series workload traffic data of web-service applications do not exist. Hence, we use AzurePublicDatasetV2 [52] 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 [50] spawns the appropriate number of user threads at every minute. Both GRAF and K8s autoscaler achieve approximately

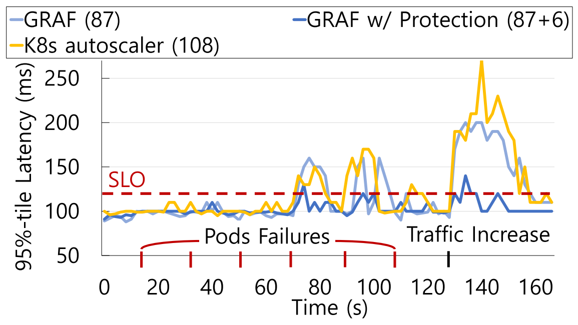


Fig. 24. 95%-tile latency of GRAF, GRAF with failure protection, K8s autoscaler during temporal pods failures at each microservices and traffic increase. The number in the bracket indicates the total number of pods allocated.

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 the workload increases and scale-downs slowly when the workload 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 [17]-like algorithm when traffic surges using Online Boutique [28] application. As mentioned in Section II-A, 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 predetermined threshold. To generate traffic surge, we use Locust [50], which spawns multiple user threads, and 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 22 and 23 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 22, 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.

D. Tolerating Instances Failures With GRAF

Until now we have shown the effectiveness of GRAF in achieving SLO with much fewer resources compared to the

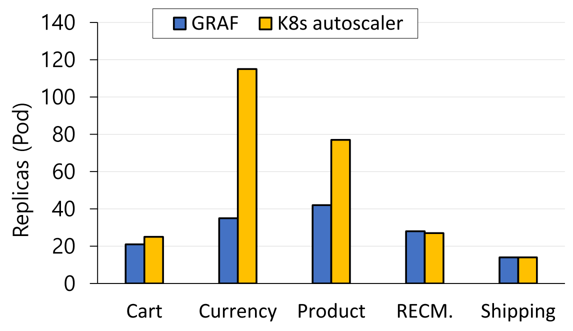


Fig. 25. CPU resource quota allocation for each microservice comparison between GRAF and K8s autosclaer in online boutique.

baselines. However, temporal instance failures aren't very rare in microservices and they can easily lead to SLO violations. In this section, we evaluate how effectively GRAF endures instances failures with additional instances allocated from SLO-aware Failure Protection (§ III-F).

Experimental Setup. We use Online Boutique application, and the resource configuration following evaluation in (§ V-B). During the experiment, the traffic is simulated with 600 application users generated from the locust. To demonstrate the instances failures, at a predetermined time step, we have failed two instances for each microservices. Failed instances are automatically respawned by Kubernetes. The time taken for instances to respawn is measured around 10 seconds which varies slightly among microservices. In specific at time 10s, we failed 2 instances at MS1, at time 30s failed 2 instances at MS2, at time 50s failed 2 instances at MS3, and so on. Moreover, we also tested an increase in the input traffic volume by 10% for 20 seconds at time 130s. For the predetermined resource budget for SLO-aware Failure Protection, we choose 6 instances. Following the results from Equation 7, SLO-aware Failure Protection allocates 1 instance to MS2, 2 instances to MS4, 2 instances to MS5, and 1 instance to MS6. As a result, throughout the experiment, GRAF, GRAF with Failure Protection, and Kuberenetes autoscaler utilize in total 86, 93, and 108 instances, respectively.

Enduring instances failures. Figure 24 shows the timeline of end-to-end percentile latency during instance failures and traffic increases taking place at the microservices. In the absence of failures, all the resource-allocating frameworks satisfy the latency SLO set as 120ms. However, during the temporal failures at MS4, MS5, and MS6, GRAF violates latency SLO for 18 seconds and K8s autoscaler violates latency SLO for 22 seconds. In contrast, GRAF with Failure Protection violates latency SLO for only 2 seconds during the instance failures throughout the experiment. Also, during the 10% increase in traffic volume, GRAF with Protection experiences slight SLO violations for 2 seconds while GRAF experiences SLO violations for 26 seconds and K8s autoscaler experiences SLO violations for 28 seconds. By distributing resources to SLO-critical microservices analyzed through SLO-aware Failure Protection (§ III-F), the microservices become more robust toward internal failures and the external environment. Note that, GRAF with Failure Protection still saves 13.9% resources compared to the K8s autoscaler baseline.

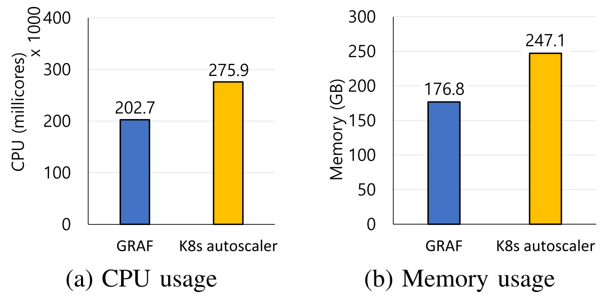


Fig. 26. Microservice application computing resource usage comparison between GRAF and K8s autoscaler.

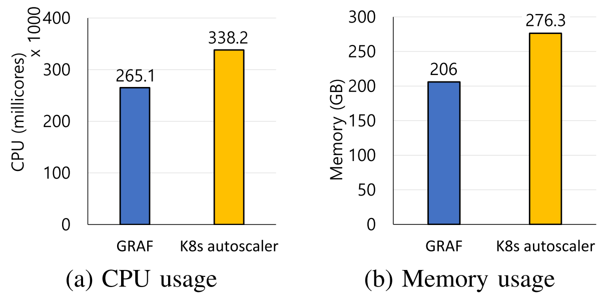


Fig. 27. Total computing resource usage in microservices clusters of GRAF and K8s autoscaler.

E. GRAF in Scale

The consistent performance of GRAF in a large-scale cloud environment is highly plausible from the previous evaluations including the results from Figure 19. Nevertheless, questions about the performance of GRAF in scale remained. In this subsection, we show empirical results of GRAF, operating in large-scale experiment settings.

Experimental Setup. For the overall experiment, one master and seven worker nodes are used, composing total 360 CPU cores and 989.9 Gigabytes of memory. A node for master and workers consists of two Intel Xeon (E5-2560) CPUs with 128GB of memory. The nodes are inter-connected via Mellanox ConnectX-5 which supports 100 Gbps. Locust [50] is run on 24 CPU cores to generate users sending out HTTP requests to the microservice application. The traffic surge scenario simulates a sudden peak of application users where 5,100 users are generated from the locust at once which invokes approximately a thousand requests per second. The gain of GRAF is demonstrated under the microservice application, Online Boutique [28]. The target SLO is set as 180 milliseconds in 95%-tile latency.

Strategy and Gain from GRAF. Figure 25 depicts the resource allocation strategy of GRAF compared to the K8s autoscaler. As expected from the experiment shown in Figure 19, GRAF maintained its resource allocation behavior in the large-scale experiment setting. GRAF allocates more replicas to *recommendation* microservice and *shipping* microservice in return for fewer replicas allocated for the other microservices. Such strategy was also shown in the previous Figure 16, where *MS5* and *MS6* corresponds to *recommendation* and *shipping* microservice, respectively.

The total gain of the GRAF is demonstrated in Figure 26 and Figure 27. During the experiments of GRAF in scale, we additionally collected memory resources metrics along with the CPU resources. The evaluation shows that GRAF saves 73200 millicores of CPU resources, corresponding to

73.2 cores of CPU. In addition, GRAF saves 70.3 GB of memory resources. Application-wise, GRAF achieves 26.5% reduction in CPU usage and 28.5% in memory usage. The number of resources saving is still significant in terms of total cluster-wise, where GRAF saves 21.6% and 25.4% for CPU resources and memory resources respectively.

Note that, GRAF achieves 160ms in 95%-tile latency throughout the large-scale evaluation while K8s autoscaler achieves 180ms in 95%-tile latency. In other words, the resource savings from the GRAF is made while serving better user experiences. GRAF achieves better 95%-tile latency in results of its Hüber loss function in the latency prediction model, which encourages conservative behavior to meet the target SLO with a margin. Meanwhile, the K8s autoscaler achieves 180ms 95%-tile latency as we hand-tuned one global resource utilization threshold parameter that satisfies the target SLO (§ V-C).

VI. 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.

Robustness toward temporal instance failures. 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 or temporal instance failures. Overprovisioning resources for microservices is a possible option that would probably enhance the ability to prevent unexpected latency spikes, therefore reducing SLO violations. While naïve overprovisioning conflicts with the resource minimization goal of GRAF, we suggest an additional system design that distributes additional resources effectively to SLO-critical microservices. We quantify the vulnerability of microservices toward temporal instance failures by predicting the end-to-end percentile latency when fewer instances are allocated compared to the GRAF's resource configuration decision. Within a predetermined budget, we distribute resources among microservices according to the magnitude of expected end-to-end percentile latency elongation. In our evaluation, GRAF with SLO-aware failure protection avoids SLO violations by 91.1% and 92% compared to the vanilla GRAF and K8s autoscaler, respectively, during failure scenarios. The above robustness is achieved by utilizing 6.9% more resources than vanilla GRAF but still 13.9% less resources than K8s autoscaler.

Integer Optimization for instances scaling. Latency prediction model and resource configuration solver in GRAF works with real numbers. 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 a 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 a slight improvement in room for GRAF to save more resources.

Adaptation of GRAF methodology. Through end-to-end performance evaluations and thorough experiments on the latency prediction model, GRAF's choice of GNN model has proven to perform well in a microservices environment. Especially, the MPNN for node embedding neighbors' information is shown to be the key to achieving a better prediction model than CNN or FC networks. The model performance ranked in order of MPNN, CNN, and FC in accordance with their network structure on how well they interpret neighbors' information. In other graph-structured environments, where downstream nodes are highly affected by the upstream nodes, MPNN would be a possible candidate to accurately model the system's behavior.

VII. Related Works

Autoscaling. Numerous works apply autoscaling to cloud applications [1], [17], [18], [19], [20], [21], [22], [24], [25], [26], [27]. Most 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 [17] 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 leads to falling into sub-optimal states. Sinan [24] is an ML-based QoS-aware resource management framework. Sinan utilizes a convolutional neural network (CNN) to capture the dependencies between microservices and predict near-future tail latency and boosted trees (BT) model to evaluate the probability of QoS violation in the future. During the deployment, Sinan predicts the outcome of the next resource allocation decisions using the trained CNN and BT models. However, in order to maintain scalability, Sinan only evaluates a subset of resource allocations followed by a set of heuristics which limits Sinan from finding the optimal resource configuration along entire microservices.

MIRAS [18] 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 [20] 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 [22] leverages multi-objective Bayesian optimization to allocate resources and meet performance/cost goals. RAMBO claims to explore and find Pareto optimal points in the performance and cost relationship, where performance is chosen as the maximum throughput under latency constant. Considering that user traffic is often an uncontrollable external factor, RAMBO yet faces a lack of design to automatically find optimums under a dynamic traffic workload. Rahman and Lama [53] suggests a design of predicting end-to-end tail latency and utilizing the model to calculate the desired CPU utilization for microservices. Yet, it doesn't consider where there are multiple workflows in combination, triggering different APIs. Also, the microservice autoscale when desired CPU utilization is violated, making the system vulnerable to cascading effects.

This work is an extension of the GRAF [1]. SLO-aware failure protection (§ III-F) design component has been added to the existing framework for enhancing robustness toward possible internal instance failures. With our latency prediction model, we estimate the magnitude of the end-to-end performance impact of each microservices under instance failures. Within a pre-determined resources budget, we distribute the additional resources among microservices following the ratio of expected end-to-end performance impact. Our evaluation (§ V-D) shows that GRAF with SLO-aware failure protection effectively tolerates various instance failure scenarios and traffic increases with much fewer SLO violations compared to the vanilla GRAF and K8s autoscaler. While showing enhanced robustness, GRAF with failure protection still saves 13.8% of resources compared to the K8s autoscaler. In addition, the scalability of the end-to-end system design toward a real-world cloud environment is evaluated (§ V-E), showing empirical results of GRAF operating in large-scale experiment settings. Also, the benefit of GRAF is largely investigated in terms of CPU and memory resources. The added comparison evaluation on CNN adds an additional empirical background for the design choice of GNN to predict the end-to-end performance in microservices. The cost for the sampling process is revisited by executing a sensitivity test of model performance over the size of the training dataset.

Container startup latency. The main cause that develops cascading effect of microservices into a severe problem is container startup latency. Fu et al [15] suggest a scheduling scheme to reduce container startup time by utilizing dependencies between layers of the container images. Fu et al [15]'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 [54] reduces container startup latency by lazily pulling the container images. Yet, Slacker can not be easily deployed because of the requirements to use a proprietary NFS server and modify the Linux kernel. There are other approaches [55], [56], [57] to reduce container startup delay by reusing containers. The side effect of these approaches is an expensive cost in cloud services [15] due to excessive use of resources such as memory [58].

SLO-aware system performance prediction. There have been numerous works [59], [60], [61], [62], [63], [64] on predicting system performance in cloud services to meet SLOs regarding a variety of terms. For example, Redy [59] provides a high-performance cache using RDMA-accessible remote memory, which can customize the performance of each cache with latency and throughput SLOs. To offer SLO-driven configuration, Redy performs offline modeling that maps RDMA configurations to throughput and latency with five-level tree and proceeds with online search by traversing the tree. Harvest VMs [60] predict SLOs for its new class, Harvest VMs, similar to spot instances yet more flexible and efficient. It utilizes Random Forest model [61] to predict SLOs for Harvest VMs in terms of survival rate and the average number of cores that can be harvested. The users can estimate the SLOs of Harvest VMs and decide the desired number to achieve their requirements. AWARE [62] suggests workload autoscaling with reinforcement learning in production cloud systems that incorporates meta-learning and RL life-cycle management to improve resource utilization while avoiding workload SLO violations. Murphy [63] is an automated performance diagnosis system for distributed cloud applications. Murphy learns loosely defined associations between entities like microservices with a learning algorithm based on Markov Random Field. Zhao et.al [64] proposes a method for estimating flow-level tail latency for large-scale datacenter networks by decomposing the problem into a large number of link simulations, and mechanically combining link-level delay distributions to produce flow-level estimates.

Combinatorial Optimization. Combinatorial optimization [65] problems have been explored in many research for decades. Bayesian optimization [66], a standard in global optimizations, has shown its capability of sample-efficient optimization in non-convex, and heterogeneous functions [67], [68], [69], [70]. Recent researches improve scalability, handle constraints, and multi-objective with Bayesian optimization [71], [72], 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 [39], [73], [74], [75]. 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.

VIII. Conclusion

We present GRAF, a GNN-based proactive resource autoscaling 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 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. GRAF can minimize the total usage of resources while satisfying latency SLO without additional expert interventions.

IX. Acknowledgment

The authors would like to thank their anonymous reviewers for their constructive feedback.

References

[1] J. Park, B. Choi, C. Lee, and D. Han, "GRAF: A graph neural network based proactive resource allocation framework for SLO-oriented microservices," in *Proc. 17th Int. Conf. Emerg. Netw. Exp. Technol.*, 2021, pp. 154–167.

[2] (2020). *Microservices Adoption in 2020*. [Online]. Available: https://www.oreilly.com/radar/microservices-adoption-in-2020/

[3] (2020). *Microservice Architecture Growing in Popularity, Adopters Enjoying Success*. [Online]. Available: https://www.itproportal.com/news/microservice-architecture-growing-in-popularity-adopters-enjoying-success/

[4] J. Soldani, D. A. Tamburri, and W.-J. Van Den Heuvel, "The pains and gains of microservices: A systematic grey literature review," *J. Syst. Softw.*, vol. 146, pp. 215–232, Dec. 2018.

[5] (2021). *Microservices—Netflix Techblog*. [Online]. Available: https://netflixtechblog.com/tagged/microservices

[6] (2015). *Microservices At Amazon*. [Online]. Available: https://www.slideshare.net/apigee/i-love-apis-2015-microservices-at-amazon-54487258

[7] (2017). *Airbnb, From Monolith To Microservices: How To Scale Your Architecture*. [Online]. Available: https://www.youtube.com/watch?v=N1BWMW9NEQc

[8] C. Jones, J. Wilkes, N. Murphy, and C. Smith. *Service Level Objectives*. Accessed: Apr. 28, 2024. [Online]. Available: https://sre.google/sre-book/service-level-objectives/

[9] A. Vasan, A. Sivasubramaniam, V. Shimpi, T. Sivabalan, and R. Subbiah, "Worth their watts?—An empirical study of datacenter servers," in *Proc. HPCA 16th Int. Symp. High-Perform. Comput. Archit.*, Jan. 2010, pp. 1–10.

[10] (2019). *Borg Cluster Workload Traces*. [Online]. Available: https://github.com/google/cluster-data

[11] (2019). *Multi-Tenancy Kubernetes on Bare Metal Servers*. [Online]. Available: https://deview.kr/data/deview/2019/presentation/[231]+Multi-Tenancy+Kubernetes+on+Bare+Metal+Servers.pdf (16p)

[12] C. Reiss and A. Tumanov, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2021, pp. 1–13.

[13] (2019). *Cloud Waste to Hit Over 14 Billion in 2019*. [Online]. Available: https://devops.com/cloud-waste-to-hit-over-14-billion-in-2019/

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

[15] S. Fu, R. Mittal, L. Zhang, and S. Ratnasamy, "Fast and efficient container startup at the edge via dependency scheduling," in *Proc. 3rd USENIX Workshop Hot Topics Edge Comput. (HotEdge)*, 2020, pp. 126–132.

[16] (2021). *Kubernetes: Production-Grade Container Orchestration*. [Online]. Available: https://kubernetes.io/

[17] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices," in *Proc. 14th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2020, pp. 805–825. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/qiu

[18] Z. Yang, P. Nguyen, H. Jin, and K. Nahrstedt, "MIRAS: Model-based reinforcement learning for microservice resource allocation over scientific workflows," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 122–132.

[19] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, Jul. 2019, pp. 68–75.

[20] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2019, pp. 1994–2004.

[21] K. Rzadca et al., "Autopilot: Workload autoscaling at Google," in *Proc. 15th Eur. Conf. Comput. Syst.*, Apr. 2020, pp. 1–16, doi: 10.1145/3342195.3387524.

[22] Q. Li et al., "RAMBO: Resource allocation for microservices using Bayesian optimization," *IEEE Comput. Archit. Lett.*, vol. 20, no. 1, pp. 46–49, Jan. 2021.

[23] I. Prachitmutita, W. Aittinonmongkol, N. Pojjanasuksakul, M. Supattatham, and P. Padungweang, "Auto-scaling microservices on IaaS under SLA with cost-effective framework," in *Proc. 10th Int. Conf. Adv. Comput. Intell. (ICACI)*, Mar. 2018, pp. 583–588.

[24] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: ML-based and QoS-aware resource management for cloud microservices," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2021, pp. 167–181.

[25] Z. Wang et al., "DeepScaling: Microservices autoscaling for stable CPU utilization in large scale cloud systems," in *Proc. 13th Symp. Cloud Comput.*, 2022, pp. 16–30.

[26] S. Luo et al., "Erms: Efficient resource management for shared microservices with SLA guarantees," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, vol. 1, Dec. 2022, pp. 62–77.

[27] B. Choi, J. Park, C. Lee, and D. Han, "pHPA: A proactive autoscaling framework for microservice chain," in *Proc. 5th Asia–Pacific Workshop Netw. (APNet)*, Jun. 2021, pp. 65–71.

[28] (2021). *Online Boutique By Google*. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo

[29] (2018). *Stan's Robot Shop By Instana*. [Online]. Available: https://www.instana.com/blog/stans-robot-shop-sample-microservice-application/

[30] (2021). *Bookinfo Application By Istio*. [Online]. Available: https://istio.io/latest/docs/examples/bookinfo/

[31] (2021). *Horizontal Pod Autoscaler of Kubernetes*. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[32] (2021). *Configuring Horizontal Pod Autoscaling in GKE*. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/how-to/horizontal-pod-autoscaling

[33] (2018). *Horizontal Pod Autoscaler in AWS*. [Online]. Available: https://docs.aws.amazon.com/eks/latest/userguide/horizontal-pod-autoscaler.html

[34] (2021). *Scale Applications in Azure Kubernetes Service (AKS)*. [Online]. Available: https://docs.microsoft.com/en-us/azure/aks/tutorial-kubernetes-scale

[35] (2020). *Vegeta: A Versatile HTTP Load Testing Tool*. [Online]. Available: https://github.com/tsenart/vegeta

[36] (2018). *Examples and Types of Microservices*. [Online]. Available: https://www.itrelease.com/2018/10/examples-and-types-of-microservices/

[37] (2020). *The Story of Netflix and Microservices*. [Online]. Available: https://www.geeksforgeeks.org/the-story-of-netflix-and-microservices/

[38] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM Special Interest Group Data Commun.*, 2019, pp. 270–288.

[39] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," 2017, *arXiv:1704.01665*.

[40] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," 2018, *arXiv:1810.10659*.

[41] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proc. 34th Int. Conf. Mach. Learning*, vol. 70, 2017, pp. 1263–1272. http://proceedings.mlr.press/v70/gilmer17a.html

[42] (2021). *Cadvisor Software on Github*. [Online]. Available: https://github.com/google/cadvisor

[43] (2021). *Jaeger: Open Source, End-to-End Distributed Tracing*. [Online]. Available: https://www.jaegertracing.io/

[44] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.

[45] (2021). *Prometheus Monitoring System & Time Series Database*. [Online]. Available: https://prometheus.io/

[46] (2021). *Linkerd: The World's Lightest, Fastest Service Mesh.* [Online]. Available: https://linkerd.io/

[47] A. Paszke et al., "Automatic differentiation in PyTorch," in *Proc. NIPS Workshop Autodiff*, Long Beach, CA, USA, 2017. [Online]. Available: https://www.bibsonomy.org/bibtex/2d9d4911f0310e65b1d54ff4c13f11 aad/ross_mck and https://openreview.net/forum?id=BJJsrmfCZ

[48] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric," 2019, *arXiv:1903.02428*.

[49] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware–software implications for cloud & edge systems," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 3–18.

[50] (2021). *Locust: An Open Source Load Testing Tool*. [Online]. Available: https://locust.io/

[51] (2021). *Amazon EC2 On-Demand Pricing*. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/

[52] M. Shahrad et al., "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proc. USENIX Conf. Annu. Tech. Conf. (USENIX ATC)*, 2020, pp. 205–218.

[53] J. Rahman and P. Lama, "Predicting the end-to-end tail latency of containerized microservices in the cloud," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Jun. 2019, pp. 200–210.

[54] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy Docker containers," in *Proc. 14th USENIX Conf. File Storage Technolog. (FAST)*, 2016, pp. 181–195.

[55] (2014). *Understanding Container Reuse in AWS Lambda*. [Online]. Available: https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/

[56] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proc. Symp. Cloud Comput.*, Sep. 2017, pp. 445–451.

[57] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA, Jul. 2018, pp. 133–146.

[58] N. Zhao et al., "Large-scale analysis of the Docker hub dataset," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2019, pp. 1–10.

[59] Q. Zhang, P. A. Bernstein, D. S. Berger, and B. Chandramouli, "Redy: Remote dynamic memory cache," 2021, *arXiv:2112.12946*.

[60] P. Ambati et al., "Providing SLOs for resource-harvesting VMs in cloud platforms," in *Proc. 14th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2020, pp. 735–751.

[61] A. Liaw and M. Wiener, "Classification and regression by randomForest," *R News*, vol. 2, no. 3, pp. 18–22, 2002.

[62] H. Qiu et al., "$AWARE$: Automate workload autoscaling with reinforcement learning in production cloud systems," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2023, pp. 387–402.

[63] V. Harsh, W. Zhou, S. Ashok, R. N. Mysore, B. Godfrey, and S. Banerjee, "Murphy: Performance diagnosis of distributed cloud applications," in *Proc. ACM SIGCOMM Conf.*, Sep. 2023, pp. 438–451.

[64] K. Zhao, P. Goyal, M. Alizadeh, and T. E. Anderson, "Scalable tail latency estimation for data center networks," in *20th USENIX Symp. Networked Syst. Design Implement. (NSDI 23)*, 2023, pp. 685–702.

[65] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and Complexity*. Chelmsford, MA, USA: Courier Corporation, 1998.

[66] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz, "BOA: The Bayesian optimization algorithm," in *Proc. Genetic Evol. Comput. Conf. (GECCO)*, vol. 1, 1999, pp. 525–532.

[67] E. Brochu, V. M. Cora, and N. de Freitas, "A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," 2010, *arXiv:1012.2599*.

[68] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," 2012, *arXiv:1206.2944*.

[69] J. T. Springenberg, A. Klein, S. Falkner, and F. Hutter, "Bayesian optimization with robust Bayesian neural networks," in *Proc. Adv. neural Inf. Process. Syst.*, vol. 29, 2016, pp. 4134–4142.

[70] M. Balandat et al., "BoTorch: A framework for efficient monte-carlo Bayesian optimization," 2019, *arXiv:1910.06403*.

[71] D. Eriksson and M. Poloczek, "Scalable constrained Bayesian optimization," in *Proc. 24th Int. Conf. Artif. Intell. Stat. (AISTATS)*, 2021, pp. 730–738. [Online]. Available: http://proceedings.mlr.press/v130/eriksson21a.html

[72] S. Ariafar, J. Coll-Font, D. Brooks, and J. Dy, "ADMMBO: Bayesian optimization with unknown constraints using ADMM," *J. Mach. Learn. Res.*, vol. 20, no. 123, pp. 1–26, 2019. [Online]. Available: http://jmlr.org/papers/v20/18-227.html

[73] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," 2016, *arXiv:1611.09940*.

[74] Q. Cappart, T. Moisan, L.-M. Rousseau, I. Prémont-Schwarz, and A. Cire, "Combining reinforcement learning and constraint programming for combinatorial optimization," 2020, *arXiv:2006.01610*.

[75] T. Barrett, W. Clements, J. Foerster, and A. Lvovsky, "Exploratory combinatorial optimization with reinforcement learning," in *Proc. AAAI Conf. Artif. Intell.*, vol. 34, no. 4, 2020, pp. 3243–3250.

Jinwoo Park received the B.S. and M.S. degrees from the School of Electrical Engineering, KAIST, in 2019 and 2021, respectively, where he is currently pursuing the Ph.D. degree with the School of Electrical Engineering. His research interests include network systems and applied machine learning.

Byungkwon Choi received the master's and Ph.D. degrees from the School of Electrical Engineering, KAIST, in 2016 and 2021, respectively, under the supervision of Prof. Dongsu Han. He is currently a Staff Engineer with Samsung Electronics, where he leads a project focused on developing infrastructures for AI. His research interests include cloud computing and infrastructure for artificial intelligence.

Chunghan Lee (Member, IEEE) received the Ph.D. degree from the Department of Electronic and Information Engineering, Toyohashi University of Technology, Japan, in 2013. From 2013 to 2019, he was with Fujitsu Laboratories Ltd., Japan. He is currently a Senior Researcher with Toyota Motor Corporation, Japan. His research interests include SDN/NFV/cloud and networking/infrastructure for connected vehicles.

Dongsu Han (Member, IEEE) received the B.S. degree in computer science from KAIST in 2003 and the Ph.D. degree in computer science from Carnegie Mellon University in 2012. He is currently a Professor with the School of Electrical Engineering, KAIST, and Graduate School of Artificial Intelligence. His research interests include networking, distributed systems, and network/system security. He is an Associate Editor of IEEE/ACM TRANSACTIONS ON NETWORKING. For more information visit the link (http://ina.kaist.ac.kr).