pHPA: A Proactive Autoscaling Framework for Microservice Chain

Byungkwon Choi
KAIST
South Korea

Chunghan Lee
Toyota Motor Corporation
Japan

Jinwoo Park
KAIST
South Korea

Dongsu Han
KAIST
South Korea

ABSTRACT
Microservice is an architectural style that breaks down monolithic applications into smaller microservices and has been widely adopted by a variety of enterprises. Like the monolith, autoscaling has attracted the attention of operators in scaling microservices. However, most existing approaches of autoscaling do not consider microservice chain and severely degrade the performance of microservices when traffic surges. In this paper, we present pHPA, an autoscaling framework for the microservice chain. pHPA proactively allocates resources to the microservice chains and effectively handles traffic surges. Our evaluation using various open-source benchmarks shows that pHPA reduces 99%-tile latency and resource usage by up to 70% and 58% respectively compared to the most widely used autoscaler when traffic surges.

1 INTRODUCTION
Microservice is a software architectural style, which is widely adopted across various enterprises including Netflix [19], Amazon [21], and Airbnb [42]. According to the survey report from O’Reilly in 2020 [18, 20], 77% of 1,502 respondents who took on a technical role in the company said that they have introduced microservices to their businesses. The microservice architecture breaks traditional monolithic applications into multiple small components, which are called microservices. Microservices communicate with one another and form complex relationships. A request to the service goes through a chain of microservices to execute the request. It is reported that commercial applications have hundreds of microservices that exchange messages to each other [9, 32].

At the same time, resource autoscaling gains popularity for balancing the quality of service (QoS) and operating costs. Traffic dynamically changes due to various reasons. When traffic changes, allocating resources to cloud applications promptly is critical. If resources are not provisioned enough in time, a service outage occurs [5, 11, 22, 38, 39]. For example, Amazon had faced a service outage due to the failure of provisioning additional servers on Prime Day in 2018 [11]. Microsoft and Zoom had experienced an outage due to the traffic surge stemming from the COVID-19 pandemic [22, 38]. To avoid such outages, some service operators rely on provisioning resources, but this essentially leads to high operating costs and low utilization [6, 37, 54]. To balance operating costs and service quality, operators usually apply autoscaling in production [1, 2, 11, 23, 33]. Resource autoscaling detects changes in workload or resource usage and adjusts resource quota without human intervention. Autoscaling is promising, however, existing approaches suffer from limitations.

Most existing autoscalers [40, 53, 58] provision resources only after issues (e.g., long-tail latency, high resource utilization) happen. This approach becomes problematic when scaling microservice chains because resource provisioning takes time. Creating a microservice instance involves accessing a file system to fetch system libraries and data files and takes up to tens of seconds. Google Borg revealed that the median container startup latency is 25 seconds [56]. Recent work also reports that 90%-tile latency of microservice startup is about 15 seconds even with a state-of-the-art scheduler [44].

This provisioning time is propagated down through the microservice chain. When a microservice receives a request, for example, it subsequently transmits related requests to the following microservices in the same chain. If a microservice lacks resources, the incoming traffic to the microservice would be dropped and the microservice would not transmit requests to the following microservices. With the existing autoscalers, the further a microservice is located at the back of the chain, the slower it will perceive changes in the workload. Until the rearmost microservice in the chain does not experience a resources shortage, the service would continue to respond improperly. To avoid such disruption, some service operators manually provision resources even though they are using autoscaling schemes [11, 23]. Some operators also use schedule-based autoscaling to hide the resource provisioning time based on traffic history [17, 27–29], but this cannot deal with sudden traffic surges.

To overcome the limitation of the existing autoscalers, we present pHPA, a proactive autoscaling framework for microservice chain. pHPA first identifies the appropriate amount of resources for each microservice by using graph neural network (GNN) and gradient
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 increases abruptly. As described in Section 6, most existing autoscalers do not consider the microservice chain and face the cascading effect. In this section, we describe the cascading effect based on Kubernetes (K8s) autoscaler [40] that is widely used [8, 10, 23, 26]. Then, we show how to avoid it.

K8s autoscaler operates with a pre-determined resource utilization threshold. When a microservice’s resource utilization reaches the threshold, the autoscaler creates more instances of the microservice to keep the utilization below a certain level \(^1\). Service operators can control the threshold to adjust how quickly the autoscaler reacts to the changes in resource utilization. The autoscaler monitors the usage of the resources such as CPU or memory and independently creates instances for each microservice.

Cascading effect. Instance creation time is accumulated and propagated down through the microservice chains and we call this phenomenon the cascading effect. Figure 1 shows the time it takes to create instances \(^2\). On average across microservices, it takes 5.5 seconds to create a single instance. When multiple instances are created at once, the creation time increases. Furthermore, in production settings, service operators set an interval (e.g., 15 seconds) of how often the autoscaler works to prevent the number of instances from being fluctuated. This control interval makes the autoscaler much slower to perceive changes in resource utilization.

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 [24]. This microservice chain is to get a cart page of the service. When ‘Frontend’ receives a request from an end-user, it first sends a request to the following microservice called ‘Currency’. ‘Frontend’ then sequentially sends a request to the successive microservice ‘Cart’ and so forth. Assume that ‘Currency’ receives an excessive amount of requests and the autoscaler creates more instances for ‘Currency’. During that time, ‘Cart’ is not aware of the change in the workload. After the additional instances of ‘Currency’ are created, ‘Cart’ perceives the increase of the workload. This happens sequentially to the following microservices.

We can observe this phenomenon in experiments. Figure 5 (upper graph) 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 [36]. 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 workloads even further later at 155s. A naive approach to reducing

---

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

\(^2\)We create instances of microservices in [24] on a single worker node and ignore the network delay to download the container images.
Our goal is to automatically identify the appropriate number of instances, we can develop an autoscaler that avoids the cascading effect. 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, this approach ‘Proactive’ reduces the 99%-tile latency by 27.8 to 17.2 seconds but the total number of instances dramatically increases from 51 to 258.

Opportunity. When we create the instances for all microservices in a chain at once, we could avoid the cascading effect. We first transmit the cart queries and then manually create the heuristically-determined number of instances for each microservice. As shown in Figure 2 and 3, this approach ‘Proactive’ reduces the 99%-tile latency by 8.6 times compared to the 10% threshold setting of K8s autoscaler. However, it is non-trivial because of the multimodality of distribution from which latency of microservices is drawn and the complexity of the microservice chain. GNN is known to be scalable when modeling graph-structured workloads like microservice chain. By leveraging GNN, we model tail latency of microservices. In addition, the search space is also very large because there are tens [24, 31, 45] or hundreds [9, 32] of microservices.

Formulation. To overcome this challenge, we model tail latency by using a graph neural network (GNN). \( L(\bar{r}, w) \) is a complex black-box function and structured in the form of a graph. Estimating \( L(\bar{r}, w) \) by assuming latency is drawn from a certain distribution is infeasible because of the multimodality of distribution from which latency of microservices is drawn and the complexity of the microservice chain. GNN is known to be scalable when modeling graph-structured workloads [43, 51, 52] like microservice chain. By leveraging GNN, we model \( L(\bar{r}, w) \). With the trained latency model, we solve the formula by using a gradient method. To apply gradient method, we transform the formulation.

\begin{align}
\min_{\bar{r}} \sum_{r \in \bar{r}} r & \quad \text{(1)} \\
\text{s.t. } L(\bar{r}, w) \leq \text{Latency SLO} & \quad \text{(2)}
\end{align}

where \( \bar{r} \) is the number of instances for each microservice; \( w \) is workload (e.g., queries per sec); and \( L(\bar{r}, w) \) is the end-to-end tail latency of microservices. We must solve the formula in real-time because workload changes continuously. However, it is non-trivial because of the search space which is very large.

pHPA design. Figure 6 (left side) illustrates an overview of the pHPA framework. In the beginning, pHPA exploits an existing approach such as K8s autoscaler until training is done. Then, pHPA periodically solves the formulation by using the trained model and gradient method.

It consists of two modules. First, Latency Model Learning module collects samples from a microservice management framework and trains a latency model using GNN. A sample consists of traffic that end-users transmit to an application, the number of microservice instances, and tail latency (e.g., 90%-tile). The architecture of the GNN model is constructed based on the microservice chain structure. The GNN model takes workload and the number of instances of microservices as input and infers tail latency. Latency Model Learning module continuously scrapes samples and trains the GNN model until the accuracy of GNN exceeds a certain threshold, as formulated as:

\begin{align}
\min_{\bar{r}} \sum_{r \in \bar{r}} r & \quad \text{(1)} \\
\text{s.t. } L(\bar{r}, w) \leq \text{Latency SLO} & \quad \text{(2)}
\end{align}

where \( \bar{r} \) is the number of instances for each microservice; \( w \) is workload (e.g., queries per sec); and \( L(\bar{r}, w) \) is the end-to-end tail latency of microservices. We must solve the formula in real-time because workload changes continuously. However, it is non-trivial because of the multimodality of distribution from which latency of microservices is drawn and the complexity of the microservice chain. GNN is known to be scalable when modeling graph-structured workloads [43, 51, 52] like microservice chain. By leveraging GNN, we model \( L(\bar{r}, w) \). With the trained latency model, we solve the formula by using a gradient method. To apply gradient method, we transform the formulation.
shown in Figure 6 (right side). This module divides the collected samples by training and test sets and evaluates the accuracy of the model. When training is done, pHPA stops exploiting the existing autoscaler and runs the next module.

*Formulation Solver* identifies the appropriate number of instances for each microservice by using the trained GNN model and gradient method. It first scrapes the frontend workload from the management framework. We combine Equation 1 and 2 to apply gradient method as follows:

$$\min_{r \in R} \sum_{r \in R} r - \lambda \times \max(L(r, w) - SLO, 0)$$  \hspace{1cm} (3)

where $\lambda$ is a penalty term that is applied when latency SLO is violated. We use a smooth max function [30] here and $L(r, w)$ is modeled by using GNN so that Equation 3 is end-to-end differentiable and gradient method can be applied. *Formulation Solver* module solves the transformed formulation by using gradient method and identifies the appropriate number of instances for each microservice. Finally, the solution is applied to microservice applications.

**pHPA prototype.** We implement a prototype that calculates the number of instances based on CPU usage. It is because the work is underway to implement and evaluate the design described in Section 3. The prototype predicts CPU usage for each microservice by using Parametric Predictive Gaussian Process Regression (PPGPR) [48] and calculates the number of microservice instances by using ceil(CPU usage / (quota ÷ (1-margin))). The prototype estimates CPU usage of each microservice based on the frontend workload and creates the instances proactively to avoid the cascading effect. PPGPR is known to be suitable at modeling stochastic data such as CPU usage [48]. $\text{quota}$ indicates resource quota of a single microservice instance. $\text{margin}$ is a ratio of how much CPU quota margin to give. We leverage Prometheus [25] and Linkerd [15] to collect CPU usage and workload and Jaeger [12] for the information of microservice chains. We implement PPGPR using GPytorch [35]. The prototype is implemented in 3.2K lines of Python code. We call this prototype pHPA in the next section.

### 4 PRELIMINARY EVALUATION

We evaluate pHPA using three open-source applications including *Bookinfo* [4], *Robot Shop* [31], and *Online Boutique* [24]. Microservice chain of the applications is illustrated in Figure 4 and 7, respectively. All microservices are deployed using Docker containers. We use Kubernetes [13] as the underlying container orchestration framework. We run Kubernetes clusters on 7 machines with two Intel E5-2650 CPUs and 128GB of memory. We use one machine for Kubernetes master node and 6 for worker nodes. We run the load generators on a separate machine. Unless specifically noted, we allocate 0.3 CPU quota per microservice instance and set margin of pHPA and the CPU utilization threshold of K8s autoscaler to 0.5 and 25%, respectively.

#### 4.1 Resource Usage Prediction Analysis

We evaluate how accurately pHPA predicts CPU usages. We run the applications and collect pairs of workload and CPU usage for each microservice while varying workload from 50 to 300 qps using Vegeta [36]. We divide it by train(80%) and test(20%) set and train the resource usage model for each microservice using PPGPR. We use the combination of linear and RBF for PPGPR kernel function. We compare the results of PPGPR with that of linear regression. We use the upper bound of 95% confidence interval as the estimation value. PPGPR takes 6 µs on average to predict CPU usage.

Table 1 shows the result. PPGPR underestimates CPU usage only for up to 15% of data points across the applications while linear regression underestimates for 50 to 58% of data points. Underestimating resource usage is critical because the lack of resources would lead to severe degradation in performance. When we increase the intercept value of the y-axis for linear regression to make the underestimation ratio to be the same as that of PPGPR, linear regression overestimates CPU usage by 18 to 150% on average across the applications while PPGPR overestimates by 15 to 23% on average.

#### 4.2 End-to-end Evaluation

We compare pHPA with K8s autoscaler. Each microservice has a single instance at the beginning. The control interval of both pHPA and K8s autoscaler is set to 10 seconds. By using Vegeta [36], we send 1000qps, 200qps, and 300qps of workload to *Robot shop*, *Bookinfo*, and *Online Boutique*, respectively. We pre-train PPGPR of pHPA. We count the total number of microservice instances every second and measure the end-to-end latency of requests.

Figure 8 and 9 show the total number of instances and end-to-end latency, respectively. pHPA reduces 99%-tile latency by 36 to 70% while creating 1.1 to 2.4 times less number of instances at the end across the applications, compared to K8s autoscaler. The longer a microservice chain is, the more pHPA improves latency. pHPA reduces 99%-tile latency of *Online Boutique* by 70% while reducing that of *Robot Shop* by 50%. As shown in Figure 8 (b), the total number of instances created by K8s autoscaler increases to 165 in the middle
and then decreases. It is because one of the microservices in Bookinfo is implemented in Java and consumes a lot of CPU cycles in the beginning to convert Java bytecodes to machine code [7]. pHPA also experiences the same, but the conversion occurs in parallel and pHPA delivers better latency even with a smaller number of instances.

**Evaluation with dynamic workload.** We evaluate pHPA in a more realistic environment. We use Locust [16] to generate workload. Locust spawns multiple user threads, each of which sends various types of requests in a predefined order. The Locust thread randomly waits for up to 3 seconds until it sends the next request to mimic the actual user behavior. We create 10 threads every second until the total number of threads reaches 500. As shown in Figure 10 and 11, pHPA reduces 99%-tile latency by 43% while creating 1.4 times less number of instances at the end compared to K8s autoscaler.

**Cost vs QoS.** We compare pHPA with K8s autoscaler in terms of how well it can balance the tradeoff between cost and service quality. We vary the CPU utilization threshold of K8s autoscaler from 25 to 50% and margin of pHPA from 0.5 to 0.75 while sending workload by using Locust. We calculate the operating cost by following the pricing plan of AWS Fargate [3]. As shown in Figure 12, pHPA
controls the balance between cost and latency better than K8s autoscaler.

5 DISCUSSION AND FUTURE WORK

Considering multiple microservice chains. pHPA optimizes resource usage for a single microservice chain. However, there exist multiple chains in an application. For example, an open-source benchmark called Online Boutique exposes six APIs and each API composes a different microservice chain. And also each chain shares some microservices.

To optimize resource usage for multiple chains, one can naively apply the pHPA approach by adding more max terms to Equation 3. As more terms are added, however, the objective function of Equation 3 gets more complex and gradient method becomes more likely to be stuck in the local optimum. The problem is that many optimization algorithms such as gradient method are devised under the assumption of convexity [14], however, it cannot be guaranteed that the objective function is convex. Recently, some automation approaches [41, 50] have been introduced to overcome the limitation.

The core idea of those works is that they learn how to iteratively update point in the domain by leveraging reinforcement learning. We leave the adoption of it to optimize for multiple microservice chains as future work.

6 RELATED WORKS

Autoscaling. There have been a number of frameworks that autoscale cloud applications [46, 53, 55, 58, 59]. To the best of our knowledge, none of the existing autoscalers could avoid the cascading effect. FIRM [53] and MIRAS [58] are an RL-based autoscaling framework. Both works handle only microservices that experience resource shortages and this faces the cascading effect. FIRM [53] uses SVM to identify microservices critical to latency SLO violations and adjusts the quota of various resources through a policy learned by RL algorithm. Because FIRM allocates more resources to microservices that cause latency SLO violations, FIRM does not handle subsequent microservices in a chain until the queue is long enough. ATOM [46] uses queuing model and genetic algorithm. ATOM adjusts resources for microservices at once, however, ATOM still suffers from traffic surges because of the long control interval caused by expensive genetic algorithm.

Container startup latency. Container startup latency is the main cause of developing the cascading effect into a severe problem. S. Fu et al [44] present a scheduling scheme that leverages dependencies between layers of the container images to reduce container startup time. Their scheduler has been adopted into the mainline Kubernetes codebase, however, it still delivers slow startup time (90%-tile startup latency is about 15 seconds). Slacker [47] lazily pulls the contents of the container image and improves startup latency, however, it requires modifications to the linux kernel and the use of a proprietary NFS server, which are non-trivial. Some approaches [34, 49, 57] reuse containers to reduce startup delay. However, such strategies would bring about the excessive use of resources such as memory (90%-tile size of container images in Docker Hub is 1.3GB [60]) and the expensive charge of cloud services [44].

7 CONCLUSION

The role of autoscaling is to save resource usage while satisfying service quality requirement without human intervention. Existing autoscalers suffer from the cascading effect and deliver poor performance in scaling microservice chains when traffic surges. In this paper, we argue that an autoscaler for microservice chain must be designed to operate proactively. We present pHPA, an autoscaling framework that proactively allocates resources for microservice chains. Our preliminary evaluation using the prototype of pHPA demonstrates that pHPA reduces 99%-tile latency by 43% and resource usage by 27% at the same time when traffic changes, compared to the autoscaler that is most widely used in production.

REFERENCES

[7] [n. d.]. Compiling and running a Java program. https://books.google.co.kr/books?id=8kD8j3FCk5c&pg=PR20. ([n. d.]).