A Holistic Machine Learning-based Autoscaling Approach for Microservice Applications

Alireza Goli[1][a], Nima Mahmoudi[1][b], Hamzeh Khazaei[2][c] and Omid Ardakanian[1][d]
[1] *University of Alberta, Edmonton, AB, Canada*
[2] *York University, Toronto, ON, Canada*

Keywords: Autoscaling, Microservices, Performance, Machine Learning.

Abstract: Microservice architecture is the mainstream pattern for developing large-scale cloud applications as it allows for scaling application components on demand and independently. By designing and utilizing autoscalers for microservice applications, it is possible to improve their availability and reduce the cost when the traffic load is low. In this paper, we propose a novel predictive autoscaling approach for microservice applications which leverages machine learning models to predict the number of required replicas for each microservice and the effect of scaling a microservice on other microservices under a given workload. Our experimental results show that the proposed approach in this work offers better performance in terms of response time and throughput than HPA, the state-of-the-art autoscaler in the industry, and it takes fewer actions to maintain a desirable performance and quality of service level for the target application.

1 INTRODUCTION

Microservice is the most promising architecture for developing modern large-scale cloud software systems (Dragoni et al., 2017). It has emerged through the common patterns adopted by big tech companies to address similar problems, such as scalability and changeability, and to meet business objectives such as reducing time to market and introducing new features and products at a faster pace (Nadareishvili et al., 2016). Traditional software architectures, such as monolithic architecture, are not capable of accommodating these needs efficiently (Dragoni et al., 2017). Companies like SoundCloud, LinkedIn, Netflix, and Spotify have adopted the microservice architecture in their organization in recent years and reported success stories of using it to meet their non-functional requirements (Calçado, 2014; Ihde and Parikh, 2015; Mauro, 2015; Nadareishvili et al., 2016).

In the microservice paradigm, the application is divided into a set of small and loosely-coupled services that communicate with each other through a message-based protocol. Microservices are autonomous components which can be deployed and scaled independently.

One of the key features of the microservice architecture is *autoscaling*. It enables the application to handle an unexpected demand growth and continue working under pressure by increasing the system capacity. While different approaches have been proposed in the literature for autoscaling of cloud applications (Kubernetes, 2020; Fernandez et al., 2014; Kwan et al., 2019; Lorido-Botran et al., 2014; Qu et al., 2018), most related work is not tailored for the microservice architecture (Qu et al., 2018). This is because a holistic view of the microservice application is not incorporated in most related work; hence each service in the application is scaled separately without considering the impact this scaling could have on other services. To remedy the shortcoming of existing solutions, a more effective and intelligent autoscaler can be designed for microservice applications, a direction we pursue in this paper.

We introduce Waterfall autoscaling (hereafter referred to as Waterfall for short), a novel approach to autoscaling microservice applications. Waterfall takes advantage of machine learning techniques to model the behaviour of each microservice under different load intensities and the effect of services on one another. Specifically, it predicts the number of required replicas for each service to handle a given load

[a] https://orcid.org/0000-0002-0376-9750
[b] https://orcid.org/0000-0002-2592-9559
[c] https://orcid.org/0000-0001-5439-8024
[d] https://orcid.org/0000-0002-6711-5502

Goli, A., Mahmoudi, N., Khazaei, H. and Ardakanian, O.
A Holistic Machine Learning-based Autoscaling Approach for Microservice Applications.
DOI: 10.5220/0010407701900198
In *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*, pages 190-198
ISBN: 978-989-758-510-4

and the potential impact of scaling a service on other services. This way, Waterfall avoids shifting load or possible bottlenecks to other services and takes fewer actions to maintain the application performance and quality of service metrics at a satisfactory level. The main contributions of our work are as follows:

- We introduce data-driven performance models for describing the behaviour of microservices and their mutual impacts in microservice applications.
- Using these models, we design Waterfall which is a novel autoscaler for microservice applications.
- We evaluate the efficacy of the proposed autoscaling approach using Teastore, a reference microservice application, and compare it with a state-of-the-art autoscaler used in the industry.

2 RELATED WORK

The autoscalers can be categorized based on different aspects from the underlying technique to the decision making paradigm (e.g., proactive or reactive) and the scaling method (e.g., horizontal, vertical, or hybrid) (Qu et al., 2018). Based on the underlying technique, autoscalers can be classified into five categories: rule-based methods, application profiling methods, analytical modelling methods, and machine learning-based methods.

Application profiling methods measure the application capacity with a variety of configurations and workloads and use this knowledge to determine the suitable scaling plan for a given workload and configuration. For instance, Fernandez et al. (Fernandez et al., 2014) proposed a cost-effective autoscaling approach for single-tier web applications using heterogeneous Spot instances (Amazon, 2020). They used application profiling to measure the processing capacity of the target application on different types of Spot instances for generating economical scaling policies with a combination of on-demand and Spot instances.

In autoscalers with analytical modelling, a mathematical model of the system is used for resource estimation. Queuing models are the most common analytical models used for performance modelling of applications in the cloud. In applications with more than one component, such as microservice applications, a network of queues is usually considered to model the system. Gias et al. (Gias et al., 2019) proposed a hybrid (horizontal+vertical) autoscaler for microservice applications based on a layered queueing network model (LQN) named ATOM. ATOM uses a genetic algorithm in a time-bounded search to find the optimal scaling strategy. The downside of modelling microservice applications with queuing network models is that finding the optimal solution for scaling is computationally expensive. Moreover, in queueing models, measuring the parameters such as service time and request mix is non-trivial and demands a complex monitoring system (Qu et al., 2018).

Search-based optimization methods use a meta-heuristic algorithm to search the state space of system configuration for finding the optimal scaling decision. Chen et al. (Chen and Bahsoon, 2015) leveraged a multi-objective ant colony optimization algorithm to optimize the scaling decision for a single-tier cloud application with respect to multiple objectives.

Machine learning-based autoscalers leverage machine learning models to predict the application performance and estimate the required resources for different workloads. Wajahat et al. (Wajahat et al., 2019) proposed a regression-based autoscaler for autoscaling of single-tier applications. They considered a set of monitored metrics to predict the response time of the application, and based on predictions, they increased or decreased the number of virtual machines assigned to the application on OpenStack. Moreover, machine learning has been used for workload prediction in proactive autoscaling. These methods use time series forecasting models to predict the future workload and provision the resources ahead of time based on the prediction for the future workload. Coulson et al. (Coulson et al., 2020) used a stacked LSTM (Hochreiter and Schmidhuber, 1997) model to predict the composition of the next requests and scale each service in the application accordingly. Abdullah et al (Abdullah et al., 2020) introduced a proactive autoscaling method for microservices in fog computing micro data centers. They predict the incoming workload with a regression model using different window sizes and identify the number of containers required for each microservice separately. The main problem with these methods is that they can lead to dramatic overprovisioning or underprovisioning of resources (Qu et al., 2018) owing to the uncertainty of workload arrivals, especially in the news feed and social network applications.

3 MOTIVATING SCENARIO

Consider the interaction between three services in an example microservice application depicted in Figure 1. Service 1 calls Service 2 and Service 3 to complete some tasks. If Service 1 is under heavy load (R1), scaling Service 1 would cause an increase in the load observed by Service 2 (R2) and Service 3 (R3). If we predict how scaling Service 1 degrades the per-

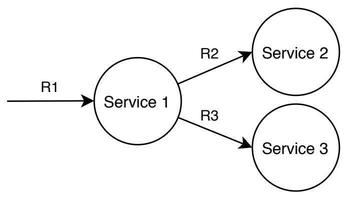

Figure 1: Interaction of services in an example microservice application.

formance of Service 2 and Service 3, we can avoid the shift in the load and a possible bottleneck from Service 1 to Service 2 and Service 3 by scaling Service 2 and Service 3 proactively at the same time as Service 1.

To further examine the cascading effect of scaling a service in a microservice application on other services, we conducted an experiment using an example microservice application called Teastore (von Kistowski et al., 2018). Teastore[1] is an emulated online store for tea and tea-related products. It is a reference microservice application developed by the performance engineering community to provide researchers with a standard microservice application that can be used for testing and evaluating research in different areas such as performance modelling, cloud resource management, and energy efficiency analysis (von Kistowski et al., 2018). Figure 6 shows services in the Teastore application and the relationships between them. Teastore includes five primary services: Webui, Auth, Persistence, Recommender, and Image.

As can be seen in Figure 6, depending on the request type, the Webui service may invoke Image, Persistence, Auth, and Recommender services. We generate a workload comprising different types of requests so that Webui service calls all of these four services. Keeping the same workload intensity, we increased the number of replicas for the Webui service from 1 to 5 and monitored the request rate of Webui in addition to the downstream rate of the Webui service to other services that each has one replica. For the two services m and n, we define the request rate of service m, denoted by $RR(m)$, as the number of requests it receives per second, and the downstream rate of service m to service n, denoted by $DR(m,n)$, as the number of requests service m sends to service n per second. For instance, in Figure 1, *RR(Service 1)* is equal to $R1$ and *DR(Service 1, Service 2)* is equal to $R2$.

Figure 2 shows the results of our experiment. The left plot and right plot show the request rate and total downstream rate of the Webui service for different number of replicas, respectively. Error bars indicate

[1]https://github.com/DescartesResearch/TeaStore

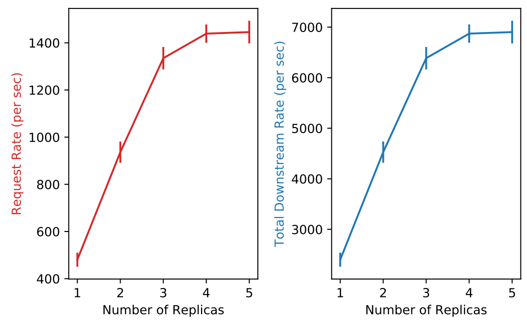

Figure 2: Request rate and total downstream rate of Webui under the same load intensity for different numbers of replicas.

the 95% confidence interval. As can be seen, scaling the Webui service leads to an increase in its request rate, which in turn increases the downstream rate of the Webui service to other services. Therefore, under heavy load, scaling the Webui service increases the load on the other four services.

The cascading effect of microservices on each other motivates the idea of having an autoscaler that takes this effect into account and takes action accordingly. Autoscalers that consider and scale different services in an application independently are unaware of this relationship, thereby making premature decisions that could lead to extra scaling actions and degradation in the quality of service of the application. In this work, we introduce a novel autoscaler to address the deficiencies in these autoscalers.

4 PREDICTING PERFORMANCE

This section presents machine learning models adopted for performance modelling of microservice applications. These models are at the core of our autoscaler for predicting the performance of each service and possible variations in performance as a result of scaling another service. Hence, we utilize two machine learning models for each microservice which are described in the following sections.

4.1 Predictive Model for CPU Utilization

The *CPU Model* captures the performance behaviour of each microservice in a microservice application in terms of CPU utilization. CPU utilization is a good proxy for estimating the workload of a microservice (Gotin et al., 2018). Therefore, we use the average CPU utilization of the microservice replicas as the performance metric for scaling decisions. Depending

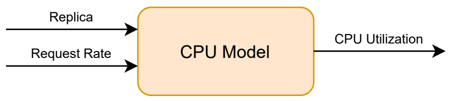

Figure 3: Input features and the predicted value of the CPU model.

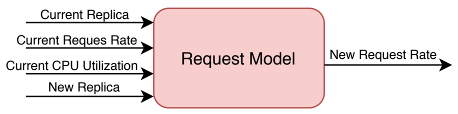

Figure 4: Input features and the predicted value of the request model.

on the target performance objective, this metric can be replaced with other metrics, such as response time and message queue metrics.

As Figure 3 demonstrates, the CPU Model takes the number of service replicas and the request rate of service as input features and predicts the service's average CPU utilization. In other words, this model can tell us what would be the average CPU utilization of service under a specific load.

4.2 Predictive Model for Request Rate

The *Request Model* predicts the new request rate of a microservice after scaling and changing the number of service replicas. As shown in Figure 4, we feed the current number of service replicas, the current average CPU utilization of service, the current request rate of service, and the new number of service replicas as input features to the Request Model to predict the new request rate for the service. The current replica, current CPU utilization, and current request rate describe the state of the service before scaling. The new replica and new request rate reflect the state of the service after scaling. We use the output of the Request Model for a given service to calculate the new downstream rate of that service to other services. Thus, the Request Model helps us predict the effect of scaling a service on other services.

4.3 Model Training Results

We trained CPU Model and Request Model for all microservices in the Teastore application using datasets created from collected data. Each dataset was split into training and validation sets. The training sets and validation sets contain 80% and 20% of data, respectively. We used Linear Regression, Random Forest, and Support Vector Regressor algorithms for the training process and compared them in terms of mean absolute error (MAE) and R^2 score. Table 1 and Table 2 show the results for CPU Model and Request Model of each microservice, respectively. As can be

Table 1: The accuracy and R^2 score of CPU Model.

Service	Linear Regresson		Random Forest		SVR	
	MAE	Score	MAE	Score	MAE	Score
Webui	4.97	92.21	3.67	96.81	1.43	99.47
Persist	4.12	94.03	3.26	96.31	0.88	99.59
Auth	4.40	94.82	4.26	95.23	1.73	99.11
Recom	2.62	92.94	1.39	97.60	1.38	97.16
Image	3.81	96.87	3.61	96.72	1.54	99.50

Table 2: The accuracy and R^2 score of Request Model.

Service	Linear Regresson		Random Forest		SVR	
	MAE	Score	MAE	Score	MAE	Score
Webui	50.01	97.83	25.67	99.02	32.01	98.70
Persist	71.21	99.50	34.94	99.86	39.36	99.84
Auth	79.23	96.35	47.34	98.74	39.57	98.82
Recom	31.22	94.26	24.49	95.84	20.27	97.17
Image	71.45	98.72	72.48	98.85	42.99	99.43

seen from the results, Support Vector Regressor and Random Forest provide lower MAE and higher R^2 score for CPU Model and Request Model compared to Linear Regression.

5 WATERFALL AUTOSCALER

In this section, we present the autoscaler we designed using the performance models described in Section 4. We first outline the architecture of Waterfall and discuss its approach to abstracting the target microservice application. Finally, we elaborate on the algorithm that Waterfall uses to obtain the scaling strategy.

5.1 Architecture and Abstraction

Figure 5 shows the architecture of Waterfall, which is based on the MAPE-K control loop (Brun et al., 2009; Kephart et al., 2003; Kephart and Chess, 2003) with five elements, namely monitor, analysis, plan, execute, and a shared knowledge base.

Waterfall abstracts the target microservice application as a directed graph, which is called microservice graph, hereafter. In the microservice graph, vertexes represent services, and edges show the dependencies between services. The direction of an edge determines which service sends request to the other one. In addition, we assign the following three weights to each directed edge (m,n) between two microservices m and n:

- **DR(m,n)** which is defined in Section 3.
- **Request Rate Ratio(m,n)** which is defined for two services m and n as:

$$Request\ Rate\ Ratio(m,n) = \frac{DR(m,n)}{RR(n)} \qquad (1)$$

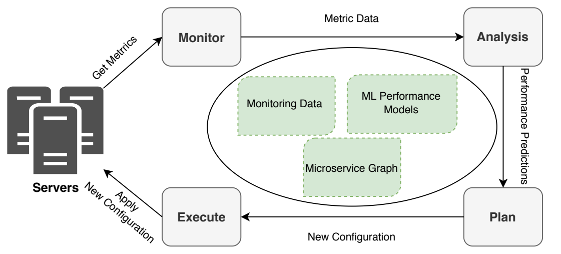

Figure 5: Architecture of Waterfall autoscaler.

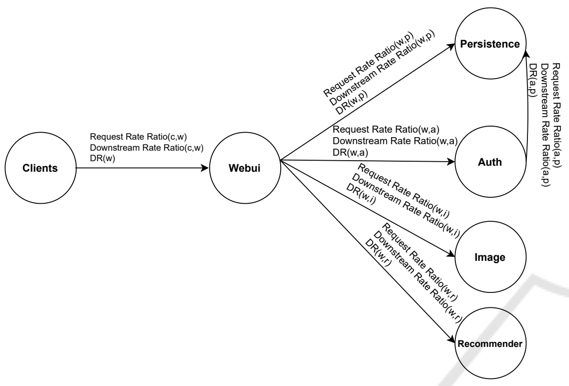

Figure 6: Teastore microservice graph.

- **Downstream Rate Ratio(m,n)** which is defined for two services m and n as:

$$Downstream\ Rate\ Ratio(m,n) = \frac{DR(m,n)}{RR(m)} \quad (2)$$

We calculate these weights for each edge and populate the graph using the monitoring data. Figure 6 shows the microservice graph for the Teastore application. The microservice graph for small applications can be derived manually according to service dependencies. There are also tools (Ma et al., 2018) for extracting the microservice graph automatically.

5.2 Scaling Algorithm

Our proposed algorithm for autoscaling of microservices leverages machine learning models to predict the number of required replicas for each service and the impact of scaling a services on the load of other services. This way, we provide a more responsive autoscaler that takes fewer actions to keep the application at the desired performance.

At the end of each monitoring interval, Waterfall initializes the microservice graph weights using monitoring data and runs the scaling algorithm to find the new scaling configuration. The steps in the Waterfall scaling algorithm are summarized in Algorithm 1. The algorithm takes the microservice graph, start node, and monitoring data as input and provides

Algorithm 1: Autoscaling Algorithm.

```
   Input: Microservice Graph G, Start Node S, Monitoring Data M
   Output: New Scaling Configuration New_Config
1  $New\_Config \longleftarrow$ initilize with current config
2  $queue \longleftarrow []$
3  $queue.append(S)$
4  while queue is not empty do
5      $service \longleftarrow queue.pop(0)$
6      $req\_rate\_updated \longleftarrow False$
7      $req\_rate \longleftarrow getReqRate(G, service)$
8      if $M[service]['Req\_Rate'] == req\_rate$ then
9          $cpu\_util \longleftarrow M[service]['CPU\_Util']$
10     else
11         $cpu\_util \longleftarrow$
             $CPU\_Model(service, new\_config[service], req\_rate)$
12     $curr\_req\_rate \longleftarrow req\_rate$
13     $curr\_cpu\_util \longleftarrow cpu\_util$
14     $curr\_replica \longleftarrow new\_config[service]$
15     if $cpu\_util >= THRESH$ then
16         $(new\_replica, pred\_req\_rate) \longleftarrow$
             $scaleOut(curr\_replica, curr\_cpu\_util,$
             $curr\_req\_rate)$
17         $updateReqRate(G, service, pred\_req\_rate)$
18         $new\_config[service] \longleftarrow new\_replica$
19         $req\_rate\_updated \longleftarrow True$
20     else if $cpu\_util < THRESH \wedge curr\_replica > 1$ then
21         $(new\_replica, pred\_req\_rate) \longleftarrow$
             $scaleIn(curr\_replica, curr\_cpu\_util, curr\_req\_rate)$
22         if $new\_replica \neq curr\_replica$ then
23             $updateReqRate(G, service, pred\_req\_rate)$
24             $new\_config[service] \longleftarrow new\_replica$
25             $req\_rate\_updated \longleftarrow True$
26     if $G[service].hasChild() \wedge req\_rate\_updated$ then
27         $updateDownstreamRate(G, service, pred\_req\_rate)$
28     for each $v \in G[service].adjacent()$ do
29         $queue.append(v)$
```

the new scaling configuration as the output. In the beginning, it initializes the New_Config with the current configuration of the system using monitoring data and starts finding the new configuration.

It traverses the microservice graph using the Breadth-First Search (BFS) algorithm and starts the search from the start node. The start node is usually the front-end service, which is the users' interaction point with the application. At each node, the algorithm checks whether the CPU utilization of the service is above or below the target threshold and scales out or scales in the service accordingly.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of Waterfall autoscaler by comparing Waterfall with HPA, which is the de facto standard for autoscaling in the

Algorithm 2: Scale Out and Scale In Functions.

```
1  Function scaleOut(curr_replica, curr_cpu_util, curr_req_rate):
2      $new\_replica \longleftarrow curr\_replica$
3      $pred\_cpu\_util \longleftarrow curr\_cpu\_util$
4      while $pred\_cpu\_util > THRESH$ do
5          $new\_replica \longleftarrow new\_replica + 1$
6          $pred\_req\_rate \longleftarrow$
             $Request\_Model(service, curr\_replica,$
             $curr\_cpu\_util, curr\_req\_rate, new\_replica)$
7          $pred\_cpu\_util \longleftarrow CPU\_Model(service, new\_replica,$
             $pred\_req\_rate)$
8      return $(new\_replica, pred\_req\_rate)$
9  Function scaleIn(curr_replica, curr_cpu_util, curr_req_rate):
10     $new\_replica \longleftarrow curr\_replica$
11     $pred\_cpu\_util \longleftarrow curr\_cpu\_util$
12     while $pred\_cpu\_util < THRESH$ do
13         $new\_replica \longleftarrow new\_replica - 1$
14         $pred\_req\_rate \longleftarrow$
             $Request\_Model(service, curr\_replica,$
             $curr\_cpu\_util, curr\_req\_rate, new\_replica)$
15         $pred\_cpu\_util \longleftarrow CPU\_Model(service, new\_replica,$
             $pred\_req\_rate)$
16         if $pred\_cpu\_util < THRESH$ then
17             $new\_req\_rate \longleftarrow pred\_req\_rate$
18     return $(new\_replica + 1, new\_req\_rate)$
```

Algorithm 3: Microservice Graph Helper Functions.

```
1  Function getReqRate(Microservice Graph G, Node service):
2      $req\_rate \longleftarrow 0$
3      for each $(m,n) \in G$ do
4          if $n == service$ then
5              $req\_rate \longleftarrow req\_rate + G[m][n]['DR']$
6      return $req\_rate$
7  Function updateReqRate(Microservice Graph G, Node service,
     new_req_rate):
8      for each $(m,n) \in G$ do
9          if $n == service$ then
10             $G[m][n]['DR'] \longleftarrow$
                 $new\_req\_rate * G[m][n]['ReqRateRatio']$
11 Function updateDownstreamRate(Microservice Graph G,
     Node service, new_req_rate):
12     for each $(m,n) \in G$ do
13         if $m == service$ then
14             $G[m][n]['DR'] \longleftarrow new\_req\_rate *$
                 $G[m][n]['DownstreamRateRatio']$
```

industry. First, we elaborate on the details of our experimental setup. After that, we present and discuss our experimental results for the comparison of Waterfall and HPA in terms of different metrics.

Table 3: Resource request and limit of Teastore services.

Service Name	CPU	Memory
Webui	1200mCore	512MB
Persist	900mCore	512MB
Auth	900mCore	512MB
Recom	800mCore	512MB
Image	1100mCore	512MB

6.1 Experimental Setup

6.1.1 Microservice Application Deployment

We created a Kubernetes[2] cluster as the container orchestration system with one master node and four worker nodes in the Compute Canada Arbutus Cloud[3]. Each node is a virtual machine with 16 vCPU and 60GB of memory running Ubuntu 18.04 as the operating system. We deployed each microservice in the Teastore application as a Kubernetes deployment exposed by a Kubernetes service. We imposed constraints on the amount of resources available to each pod. Table 3 shows the details of CPU and memory configuration for each pod.

6.1.2 Load Generation

We used Jmeter[4] , an open-source tool for load testing of web applications, to generate an increasing workload with a length of 25 minutes for the Teastore application. This workload is a common browsing workload that represents the behaviour of most users when visiting an online shopping store. It follows a closed workload model and includes actions like visiting the home page, login, adding product to cart, etc. Jmeter acts like users' browsers and sends requests sequentially to the Teastore front-end service using a set of threads. The number of threads controls the rate at which Jmeter sends requests to the front-end service. We deployed Jmeter on a stand-alone virtual machine with 16 vCPU and 60GB of memory running Ubuntu 18.04 as the operating system.

6.2 Results and Discussion

To compare the behaviour and effectiveness of Waterfall autoscaler with HPA, we applied the increasing workload described in the previous section to the front-end service of the Teastore application for 25 minutes. Figures 7-11 show the average CPU utilization and replica count for each service in the Teastore application throughout the experiment. The red

[2]Kubernetes: https://kubernetes.io/
[3]Compute Canada Cloud: https://computecanada.ca/
[4]Jmeter: https://jmeter.apache.org/

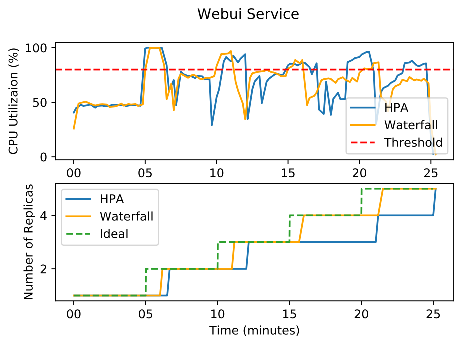

Figure 7: The CPU utilization and number of replicas for the Webui service.

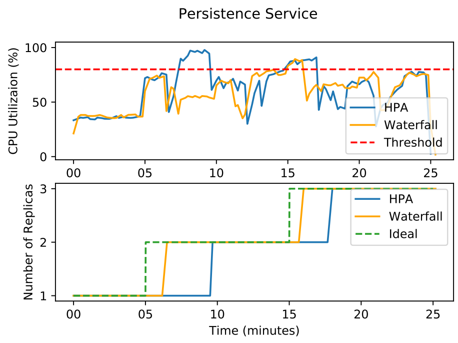

Figure 8: The CPU utilization and number of replicas for the Persistence service.

dashed line in CPU utilization plots denotes the CPU utilization threshold that both autoscalers use as the scaling threshold. The green dashed line in each service's replica count plot shows the ideal replica count for that service at each moment of the experiment. The ideal replica count is the minimum number of replicas for the service which is enough to handle the incoming load and keep the CPU utilization of the service below the threshold. According to Figures 7-11, HPA scales a service whenever the service's average CPU utilization goes above the scaling threshold. However, Waterfall scales a service in two different situations: 1) the CPU utilization of the service goes beyond the scaling threshold; 2) the predicted CPU utilization for the service exceeds the threshold due to scaling of another service.

As Figure 7 shows, for the Webui service, both autoscalers increase the replica count when the CPU utilization is above the threshold with some delay compared to the ideal state. According to Figure 6, as Webui is the front-end service and no other internal services depend on it, scaling of other services does not compromise the performance of the Webui service. Hence, all Waterfall's scaling actions for the Webui service can be attributed to CPU utilization.

As can be seen in Figure 8, we observe that Waterfall scales the Persistence service around the 6th minute, although the CPU utilization is below the threshold. We attribute this scaling action to the decision for scaling the Webui service in the same monitoring interval that leads to an increase in the CPU utilization of Persistence service as Webui service depends on Persistence service. In contrast, as we can see in Figure 8, the HPA does not scale the Persistence service at the 6th minute. Consequently, a short while after the 6th minute, when the second replica of Webui service completes the startup process and is ready to accept traffic, the CPU utilization of Persistence service increases and goes above the threshold. The other scaling action of Waterfall for Persistence

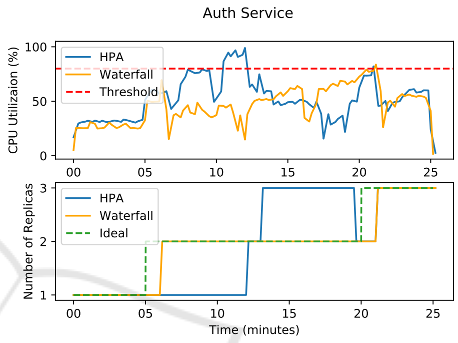

Figure 9: CPU utilization and number of replicas for Auth service.

service after the 15th minute is based on CPU utilization.

Results for the Auth service shown in Figure 9 suggest that the increase in the replica count of Auth around the 6th minute is based on the prediction for the impact of scaling of the Webui service, as the CPU utilization of Auth is below the threshold during this time. On the other hand, we can see that at 6th minute, the HPA does not increase the replica count for Auth service. Therefore, after adding the second replica of Webui, the CPU utilization of Auth reaches the threshold. The other scaling action of Waterfall for Auth after the 20th minute is based on the CPU utilization.

According to the Image service results in Figure 10, Waterfall scales the Image service around the 11th minute. This scaling action is due to scaling the Webui service that depends on Image service from two to three replicas in the same monitoring interval. However, HPA does not scale the Image service simultaneously with Webui causing an increase in the CPU utilization of the Image service. For Waterfall, as Figure 10 shows, there is a sudden increase in the CPU utilization of Image service right before the time that the second replica of Image service is ready to accept traffic. This sudden increase in CPU utilization

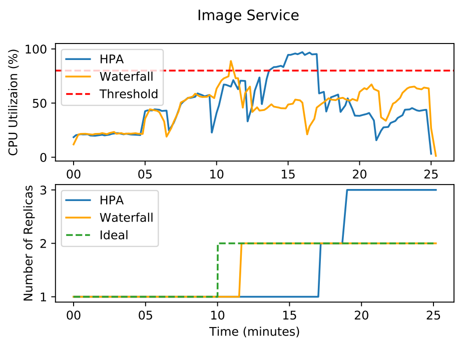

Figure 10: The CPU utilization and number of replicas for the Image service.

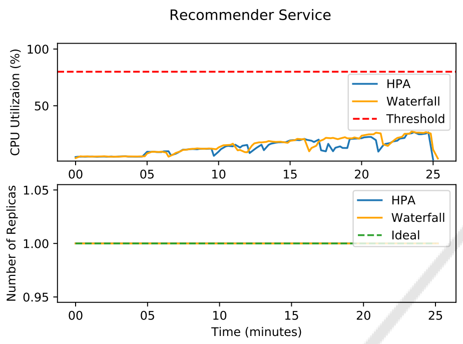

Figure 11: The CPU utilization and number of replicas for the Recommender service.

of Image service is because of the time difference between the time that Webui and Image services complete the startup process and reach the ready state. During the interval between these two incidents, the Webui service has three replicas; therefore, its downstream rate to Image service increases while the second replica of the Image service is not ready yet.

For the Recommender service, as Figure 11 illustrates, during the whole time of the experiment, the CPU utilization is below the threshold. Consequently, there is no scaling action for both autoscalers.

Putting the results of all services together, we can see that the Waterfall autoscaler predicts the effect of scaling a service on downstream services and scale them proactively in one shot if it is necessary. Therefore, it takes fewer actions to maintain the CPU utilization of the application below the threshold. For example, around the 6th minute, we can see from Figures 7, 8, and 9 that Waterfall autoscaler scales the Persistence and Auth services along with Webui in the same monitoring interval. However, HPA scales these services separately in different monitoring intervals.

To quantify the effectiveness of Waterfall compared to HPA, we evaluate both autoscalers in terms of several metrics. Figure 12 shows the total number of transactions executed per second (TPS) for Waterfall and HPA throughout the experiment. It can be seen that Waterfall has a higher cumulative TPS than HPA thanks to timely scaling of services.

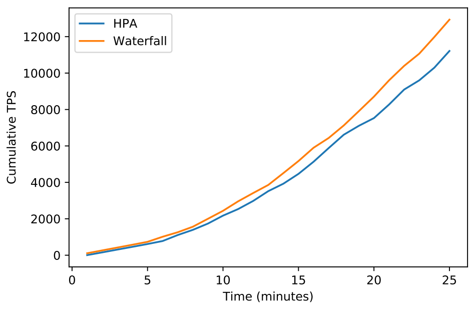

Figure 12: Cumulative Transaction Per Second (TPS) of Waterfall and HPA autoscalers.

Table 4: Comparison of Waterfall and HPA autoscalers in terms of performance metrics.

#	HPA	Waterfall
Total Request	727270.0 $\pm$ 12369.95	796867.4 $\pm$ 4594.77
TPS	484.55 $\pm$ 8.23	530.93 $\pm$ 3.06
Response Time	20.47 $\pm$ 0.36	18.67 $\pm$ 0.11

Table 5: Comparison of Waterfall (WF) and HPA in terms of CPU>threshold(T), overprovision, and underprovision time.

Service	CPU >T		Underprovision		Overprovision	
	HPA	WF	HPA	WF	HPA	WF
Webui	31%	16%	54%	15.33%	0%	0%
Persist	16%	4%	28.66%	7.33%	0%	0%
Auth	6.33%	0.33%	32%	8%	26%	0%
Image	13.33%	0.33%	28%	6%	24%	0%
Recom	0%	0%	0%	%0	0%	0%

We repeated the same experiment five times and calculated the average of the total number of served requests, TPS, and response time for both autoscalers over these runs. Table 4 shows the results along with the 95% confidence interval. It can be seen that TPS (and the total number of served requests) is 9.57% higher for Waterfall than HPA. The response time for Waterfall is also 8.79% lower than HPA.

Additionally, we have calculated CPU>threshold (T), underprovision, and overprovision time for both autoscalers and presented them in Table 5. It can be seen that for all services except the Recommender service, both autoscalers have a nonzero value for CPU>T. However, CPU>T is less for Waterfall in all services. Moreover, Waterfall yields a lower underprovision time and zero overprovision time for all services. Despite the overprovisioning of HPA for two services, we observe that Waterfall still provides a higher TPS and better response time; we attribute this to the timely and effective scaling of services by the Waterfall autoscaler.

7 CONCLUSION

We introduced Waterfall, a machine learning-based autoscaler for microservice applications. While numerous autoscalers consider different microservices in an application independent of each other, Waterfall takes into account that scaling a service might have an impact on other services and can even shift the bottleneck from the current service to downstream services. Predicting this impact and taking the proper action in a timely manner could improve the application performance as we corroborated in this study. Our evaluation results show the efficacy and applicability of our approach. In future work, we plan to explore the feasibility of adding vertical scaling to the Waterfall autoscaling approach.

REFERENCES

Abdullah, M., Iqbal, W., Mahmood, A., Bukhari, F., and Erradi, A. (2020). Predictive autoscaling of microservices hosted in fog microdata center. *IEEE Systems Journal*.

Amazon (2020). Amazon ec2 spot instances. https://aws.amazon.com/ec2/spot/. Accessed: 2020-10-25.

Brun, Y., Serugendo, G. D. M., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., and Shaw, M. (2009). Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems*, pages 48–70. Springer.

Calçado, P. (2014). Building products at soundcloud—part i: Dealing with the monolith. *Retrieved from: https://developers. soundcloud. com/blog/building-products-at-soundcloud-part-1-dealing-withthe-monolith*. Accessed: 2020-10-25.

Chen, T. and Bahsoon, R. (2015). Self-adaptive trade-off decision making for autoscaling cloud-based services. *IEEE Transactions on Services Computing*, 10(4):618–632.

Coulson, N. C., Sotiriadis, S., and Bessis, N. (2020). Adaptive microservice scaling for elastic applications. *IEEE Internet of Things Journal*, 7(5):4195–4202.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: yesterday, today, and tomorrow. In *Present and ulterior software engineering*, pages 195–216. Springer.

Fernandez, H., Pierre, G., and Kielmann, T. (2014). Autoscaling web applications in heterogeneous cloud infrastructures. In *2014 IEEE International Conference on Cloud Engineering*, pages 195–204. IEEE.

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

Gotin, M., Lösch, F., Heinrich, R., and Reussner, R. (2018). Investigating performance metrics for scaling microservices in cloudiot-environments. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 157–167.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Ihde, S. and Parikh, K. (2015). From a monolith to microservices + rest: the evolution of linkedin's service architecture. *Retrieved from: https://www.infoq.com/presentations/linkedin-microservices-urn/*. Accessed: 2020-10-25.

Kephart, J., Kephart, J., Chess, D., Boutilier, C., Das, R., Kephart, J. O., and Walsh, W. E. (2003). An architectural blueprint for autonomic computing. *IBM White paper*, pages 2–10.

Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Kubernetes (2020). Kubernetes hpa. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed: 2020-10-25.

Kwan, A., Wong, J., Jacobsen, H.-A., and Muthusamy, V. (2019). Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 80–90. IEEE.

Lorido-Botran, T., Miguel-Alonso, J., and Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592.

Ma, S.-P., Fan, C.-Y., Chuang, Y., Lee, W.-T., Lee, S.-J., and Hsueh, N.-L. (2018). Using service dependency graph to analyze and test microservices. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 81–86. IEEE.

Mauro, T. (2015). Adopting microservices at netflix: Lessons for architectural design. *Retrieved from https://www. nginx. com/blog/microservices-at-netflix-architectural-best-practices*. Accessed: 2020-10-25.

Nadareishvili, I., Mitra, R., McLarty, M., and Amundsen, M. (2016). *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.".

Qu, C., Calheiros, R. N., and Buyya, R. (2018). Autoscaling web applications in clouds: A taxonomy and survey. *ACM Computing Surveys (CSUR)*, 51(4):1–33.

von Kistowski, J., Eismann, S., Schmitt, N., Bauer, A., Grohmann, J., and Kounev, S. (2018). Teastore: A micro-service reference application for benchmarking, modeling and resource management research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 223–236. IEEE.

Wajahat, M., Karve, A., Kochut, A., and Gandhi, A. (2019). Mlscale: A machine learning based application-agnostic autoscaler. *Sustainable Computing: Informatics and Systems*, 22:287–299.