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*
{goli, nmahmoud, oardakan}@ualberta.ca, hkh@yorku.ca

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

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.

The rest of this paper is organized as follows. Section 2 reviews related work on autoscaling and Section 3 provides a motivating scenario. Section 4 presents the proposed machine learning-based performance models for microservice applications. Section 5 describes the design of Waterfall autoscaler. Section 6 evaluates the proposed autoscaling technique, and Section 7 concludes the paper.

2 RELATED WORK

Autoscaling is a widely used and well-known concept in cloud computing, mainly due to the elasticity and pay-as-you-go cost model of cloud services. With the shift in the runtime environment of microservice applications from bare-metal servers to more fine-grained environments, such as virtual machines and containers in the cloud, autoscaling has become an indispensable part of microservice applications. 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, search-based optimization methods, and machine learning-based methods.

Rule-based autoscalers act based on a set of predefined rules to scale and estimate the amount of necessary resources for provisioning. This type of autoscalers is common in the industry and usually serves as the baseline (Qu et al., 2018). Products such as Amazon AWS Autoscaling service (Amazon, 2020b) and Kubernetes Horizontal Pod Autoscaler (HPA) (Kubernetes, 2020) fall into this group. Wong et al. (Kwan et al., 2019) proposed two rule-based autoscalers similar to Kubernetes HPA for microservices, namely HyScale$_{\text{CPU}}$ and HyScale$_{\text{CPU+Mem}}$. HyScale$_{\text{CPU}}$ uses both horizontal and vertical scaling to scale each microservice in the target application separately based on CPU utilization. It gives priority to vertical scaling and applies horizontal scaling only if the required amount of resources cannot be acquired using vertical scaling. HyScale$_{\text{CPU+Mem}}$ operates similarly except that it considers memory utilization in addition to CPU utilization for making the scaling decision. Although rule-based autoscalers are easy to implement, they typically need expert knowledge about the underlying application for tuning the thresholds and defining the scaling policies (Qu et al., 2018).

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, 2020a). 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 metaheuristic 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. Machine learning techniques can be divided into regression and reinforcement learning methods. Regression-based methods usually find the relationship between a set of input variables and an output variable such as resource demand or a performance metric. 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. Jindal et al. (Jindal et al., 2019) used a regression model to estimate the microservice capacity (MSC) for each service in a microservice application. MSC is the maximum number of requests that a microservice with a certain number of replicas can serve per second without violating the service level objective (SLO). They obtained this value by sandboxing and stress-testing each service for several configuration deployments and then fitting a regression model to the collected data. In reinforcement learning approaches, an agent tries to find the optimal scaling policy for each state of the system (without assuming prior knowledge) through interaction with the system. Iqbal et al. (Iqbal et al., 2015) leveraged reinforcement learning to learn autoscaling policies for a multi-tier web application under different workloads. They identified the workload pattern from access logs and learned the appropriate resource allocation policy for a specific workload pattern so that SLO is satisfied and resource utilization is minimized. The drawback of reinforcement learning methods is the poor performance of autoscalers at the early stages of deployment because it takes some time for the reinforcement learning model to learn the optimal policy. 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.

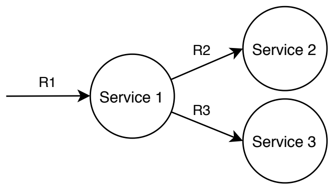

Figure 1: Interaction of services in an example microservice application.

3 MOTIVATING SCENARIO

A microservice application usually consists of multiple services interacting with each other to accomplish their job. The rate at which a service sends requests to the downstream services depends on the rate at which it receives requests and the amount of resources available for processing these requests. Thus, scaling a service that may invoke a group of other services might subsequently change the load on those services. 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 performance 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 2 shows services in the Teastore application and the relationships between them. The solid lines show the dependencies between services, and dashed lines indicate that the

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

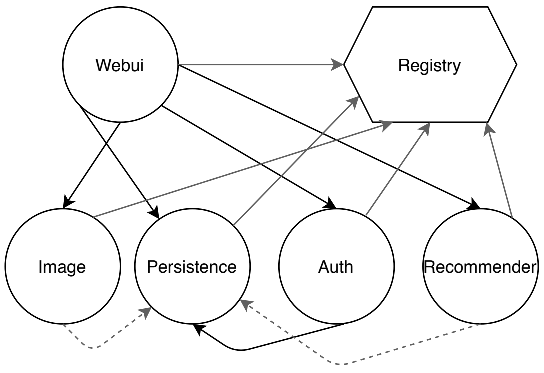

Figure 2: Architecture of the Teastore application.

service call happens only once at startup time. Teastore includes five primary services: Webui, Auth, Persistence, Recommender, and Image. Webui is the front-end service that users interact with and is responsible for rendering the user interface. Auth stands for authentication; it verifies the user's credentials and session data. The Persistence service interacts with the database and performs create, read, update, and delete (CRUD) operations. The Recommender service predicts the user preference for different products and recommends appropriate products to users using a collaborative filtering algorithm. The Image service provides an image of products in different sizes. In addition to main services, Teastore has another component named Registry, which is responsible for service registration and discovery.

As can be seen in Figure 2, 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 3 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 the 95% confidence interval. Table 1 shows the request rate of Webui and its downstream rate to each

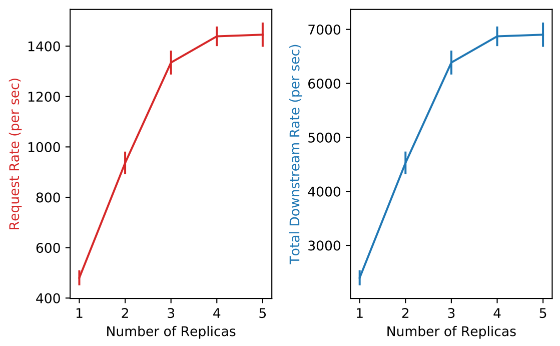

Figure 3: Request rate and total downstream rate of Webui under the same load intensity for different numbers of replicas.

Table 1: Request Rate (RR) and Downstream Rate (DR) of the Webui service to each service.

Monitored Metric	1	2	3	4	5
RR(Webui)	480	936	1334	1438	1445
DR(Webui,Persistence)	1083	2108	3005	3239	3253
DR(Webui,Auth)	482	938	1337	1441	1447
DR(Webui,Image)	562	1094	1559	1680	1688
DR(Webui,Recommender)	121	235	334	360	362

service for the different number of replicas. 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.

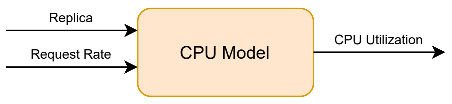

Figure 4: Input features and the predicted value of the CPU model.

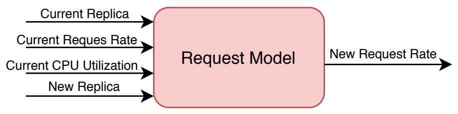

Figure 5: Input features and the predicted value of the request model.

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

As Figure 4 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 5, 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.

As we discussed in Section 3, any changes in the request rate of a service in a microservice application might lead to changes in the downstream rate of that service to other services. However, we observed that under the same workload intensity, when we scale a service, the downstream rate of that service to another service changes linearly with respect to its request rate. For instance, we used the results from Section 3 in Table 1 and divided the downstream rate of Webui service to other services by its request rate and got the values in Table 2. Consequently, when we scale a service, if we have the new request rate after scaling, we can calculate its new downstream rate to other services. We achieve this goal through Request Model. For example, according to Table 1 for one replica *RR(Webui)* ≈ *480* and *DR(Webui,Persistence)* ≈ *1083*. Moreover, from Table 2 we know that for all replica counts, *DR(Webui,Persistence) / RR(Webui)* ≈ *2.25*. Therefore, if we scale out the Webui service to two replicas and have the new value for *RR(Webui)* as *936*, we can estimate the new *DR(Webui,Persistence)* by multiplying the new *RR(Webui)* by *2.25* which will be *936 * 2.25* ≈ *2106*. The reason for the difference between the calculated value (2106) and the real value (2108) for the new *DR(Webui,Persistence)* is that numbers in Table 1 and Table 2 are rounded due to lack of space.

Table 2: The ratio of Downstream Rate (DR) values of Webui service to its Request Rate (RR) for different number of replicas under the same workload intensity.

DR/RR	1	2	3	4	5
DR(Webui,Persistence)/ RR(Webui)	2.25	2.25	2.25	2.25	2.25
DR(Webui,Auth)/ RR(Webui)	1.00	1.00	1.00	1.00	1.00
DR(Webui,Image)/ RR(Webui)	1.17	1.17	1.17	1.17	1.17
DR(Webui,Recommender)/ RR(Webui)	0.25	0.25	0.25	0.25	0.25

4.3 Data Collection

To train CPU Model and Request Model for each microservice, we needed to collect two datasets per microservice. The data collection for each microservice is performed independent of other services. We deploy enough number of replicas from other services to avoid any limitations imposed by other services on the target service for data collection.

The dataset for CPU Model includes three metrics: the number of replicas, the request rate per second, and the average CPU utilization of replicas. Each data point results from applying a workload with a fixed number of threads for 12 minutes to the front-end service. At the end of each run, we collect each metric's values during this period and use their mean as the value of the metric for that data point. Note that we ignored data values for the first and last minute of each run to exclude the warm-up and cool-down periods. We consider a different number of replicas for the target service, and for each number of replicas, we change the number of threads to increase the number of requests until we reach the saturation point for that specific number of replicas. For instance, for one replica of an example service, we apply the workload with 1, 2, 3, 4, and 5 threads, resulting in five data points.

The Request Model dataset contains five metrics: the current number of replicas, the current request rate per second, the current average CPU utilization of replicas, the new number of replicas, and the new request rate per second. Each data point for this dataset results from the merging of two runs with the same number of threads but a different number of replicas. For example, we merge the result for the run with one replica and five threads with the result for two replicas and five threads to generate a data point for the Request Model dataset. More specifically, we get the current replica, current CPU utilization, and current request rate from the first run and the new replica and new request rate from the second run.

Figure 6 shows an example data point for CPU Model and Request Model datasets. The data point for Request Model is a combination of two runs that have n threads with x and x' replicas, respectively.

4.4 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), mean squared error (MSE), root mean squared error (RMSE), and R^2 score. Table 3 and Table 4 show the results for CPU Model and Request Model of each microservice, respectively. As can be seen from the results, Support Vector Regressor and Random Forest provide lower MAE, MSE, RMSE, and higher R^2 score for CPU Model and Request Model compared to Linear Regression. Currently, we use offline learning to train machine learning models, but our approach can be adapted to leverage online learning as well.

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 7 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. For instance, consider the following vertex (V) and edge (E) sets for an example microservice graph:

$$V = \{A, B, C\}, \quad E = \{(A, B), (A, C)\} \qquad (1)$$

This microservice graph contains three services and two edges. A, B, and C are three different services. The edges (A, B) and (A, C) show that service A calls services B and C respectively. 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:

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

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

$$Downstream\ Rate\ Ratio(m,n) = \frac{DR(m,n)}{RR(m)} \qquad (3)$$

We calculate these weights for each edge and populate the graph using the monitoring data. Figure 8 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

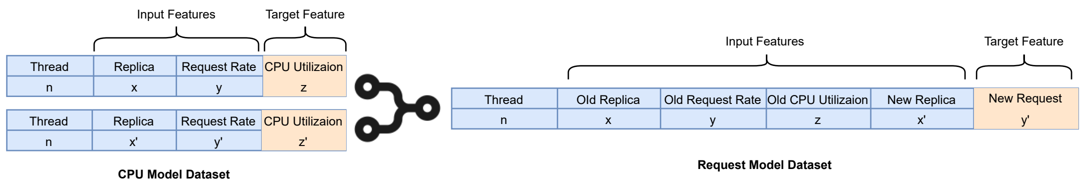

Figure 6: The construction of datasets for CPU Model and Request Model. Request Model dataset is built by merging data points from the CPU Model dataset.

Table 3: The accuracy and R^2 score of CPU Model for different services using Linear Regression (LR), Random Forest (RF), and Support Vector Regressor (SVR).

Service	Linear Regression				Random Forest				SVR			
	MAE	MSE	RMSE	Score	MAE	MSE	RMSE	Score	MAE	MSE	RMSE	Score
Webui	4.97	45.32	6.73	92.21	3.67	18.57	4.31	96.81	1.43	3.07	1.75	99.47
Persistence	4.12	27.55	5.25	94.03	3.26	17.02	4.13	96.31	0.88	1.91	1.38	99.59
Auth	4.40	37.39	6.11	94.82	4.26	34.45	5.87	95.23	1.73	6.45	2.54	99.11
Recommender	2.62	12.42	3.52	92.94	1.39	4.23	2.06	97.60	1.38	5.00	2.23	97.16
Image	3.81	20.12	4.49	96.87	3.61	21.09	4.59	96.72	1.54	3.45	1.86	99.50

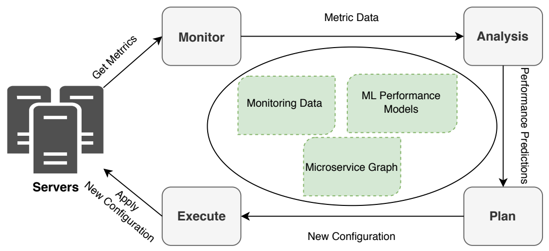

Figure 7: Architecture of Waterfall autoscaler.

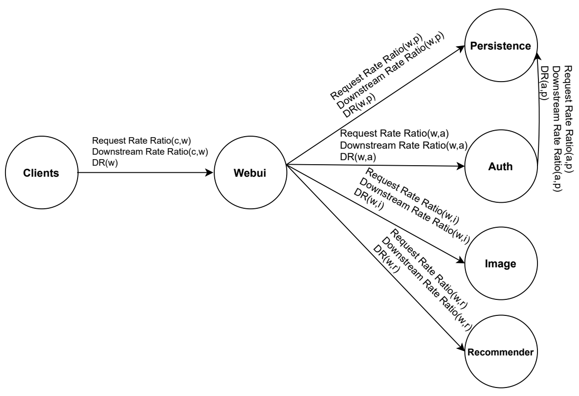

Figure 8: Teastore microservice graph.

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 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.

In case that the CPU utilization is higher than the threshold, it calls the *scaleOut* function. This function increases the service replicas and predicts the new request rate of the service using Request Model. After predicting the new request rate, it uses CPU Model to predict the new CPU utilization with the new number of replicas and the new request rate. If the new predicted CPU utilization is below the threshold, it considers the new replica as the new configuration for the service. Afterwards, it updates the microservice request rate using the *updateReqRate* function. As Algorithm 3 indicates, function *updateReqRate* updates the *DR* value on all edges ending to this microservice based on the $Request\ Rate\ Ratio$ value on each edge.

If the CPU utilization is less than the threshold, it calls the *scaleIn* function. This function reduces the number of service replicas and predicts the new request rate of the service using Request Model. It then feeds the new request rate and new replica to CPU Model to predict the new CPU utilization. If the new CPU utilization is still below the threshold, it considers the new replica as the new configuration for service and updates the microservice request rate using the *updateReqRate* function. Otherwise, it keeps the current replica as the configuration of the service.

Table 4: The accuracy and R^2 score of Request Model for different services using Linear Regression (LR), Random Forest (RF), and Support Vector Regressor (SVR).

Service	Linear Regression				Random Forest				SVR			
	MAE	MSE	RMSE	Score	MAE	MSE	RMSE	Score	MAE	MSE	RMSE	Score
Webui	50.01	3568.55	59.74	97.83	25.67	1596.37	39.95	99.02	32.01	2134.50	46.20	98.70
Persistence	71.21	9708.55	98.53	99.50	34.94	2717.49	52.13	99.86	39.36	3041.56	55.15	99.84
Auth	79.23	11158.89	105.64	96.35	47.34	3857.84	62.11	98.74	39.57	3611.02	60.09	98.82
Recommender	31.22	1258.56	35.48	94.26	24.49	911.24	30.19	95.84	20.27	620.22	24.90	97.17
Image	71.45	8137.23	90.21	98.72	72.48	7328.20	85.60	98.85	42.99	3642.93	60.36	99.43

If the node that is being processed has any children, the algorithm goes to the next step which is applying the effect of change in service replica number on downstream services by calling the *updateDownstreamRate* function. As Algorithm 3 shows, this function updates the *DR* value on all edges starting from the current node and ending at child nodes based on the *Downstream Rate Ratio* value on each edge.

After this step, the algorithm continues the BFS search by the next node and repeats the steps mentioned above. After searching the whole graph and inferring the new configuration for each service, the search is over and the algorithm returns the new scaling configuration.

As lines 8-11 show, if the request rate of the service in the current node has been changed in the graph in previous steps, the CPU utilization in the monitoring data is not valid anymore, and we should estimate the new CPU utilization using CPU Model. The *getRequestRate* function calculates the request rate of a node by summing the *DR* value on all edges ending to this node.

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 $cpu\_util < THRESH \land curr\_replica > 1$ 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 $G[service].hasChild() \land req\_rate\_updated$ then
27         $updateDownstreamRate(G, service, pred\_req\_rate)$
28     for each $v \in G[service].adjacent()$ do
29         $queue.append(v)$
```

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 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.

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. The incoming traffic is distributed in a round-robin fashion between pods that belong to a deployment. We imposed con-

[2]Kubernetes: https://kubernetes.io

[3]Compute Canada Cloud: https://computecanada.ca

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 $(new\_replica + 1, new\_req\_rate)$
```

straints on the amount of resources available to each pod using the resource request and limit mechanism in Kubernetes. The resource request is the amount of resources guaranteed for a pod, and the resource limit is the maximum amount of resources that a pod can have in the cluster. We used the same value for both resource request and limit to decrease the variability in pods' performance. Table 5 shows the details of CPU and memory configuration for each pod. We configured the startups, readiness, and liveness probes for each pod to measure the exact number of ready pods at any time in the system and also have a recovery mechanism in place for unhealthy pods. We used the Kubernetes API to query or change the number of pods in a deployment.

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.

[4]Jmeter: https://jmeter.apache.org

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             $G[m][n]['DR'] \longleftarrow new\_req\_rate * G[m][n]['DownstreamRateRatio']$
```

Table 5: Resource request and limit of Teastore services.

Service Name	CPU	Memory
Webui	1200mCore	512MB
Persistence	900mCore	512MB
Auth	900mCore	512MB
Recommender	800mCore	512MB
Image	1100mCore	512MB

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 9-13 show the average CPU utilization and replica count for each service in the Teastore application throughout the experiment. The red 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

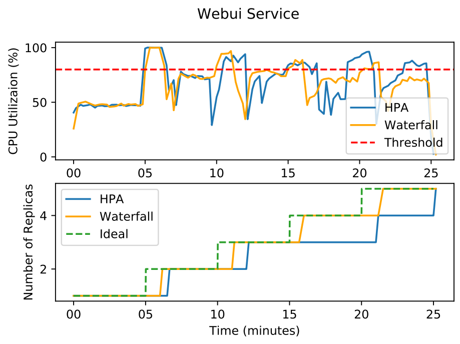

Figure 9: The CPU utilization and number of replicas for the Webui service.

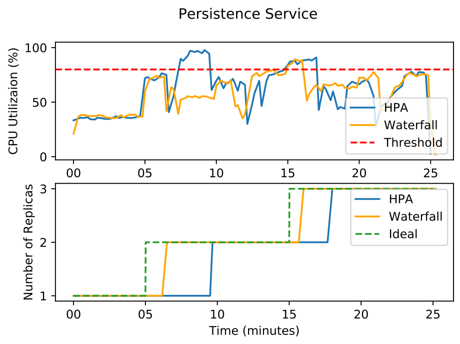

Figure 10: The CPU utilization and number of replicas for the Persistence service.

incoming load and keep the CPU utilization of the service below the threshold. According to Figures 9-13, 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. Therefore, when Waterfall scales a service while its CPU utilization is below the threshold, it must be due to the predicted performance degradation of the service as a result of scaling of another service(s).

As Figure 9 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 8, 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 10, 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 10, 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 service after the 15th minute is based on CPU utilization.

Results for the Auth service shown in Figure 11

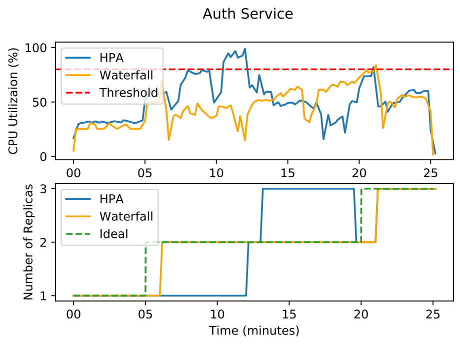

Figure 11: CPU utilization and number of replicas for Auth service.

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 12, 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 12 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 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.

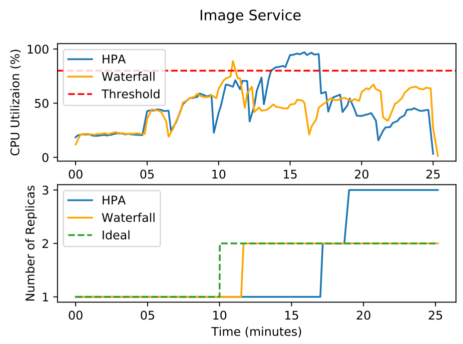

Figure 12: The CPU utilization and number of replicas for the Image service.

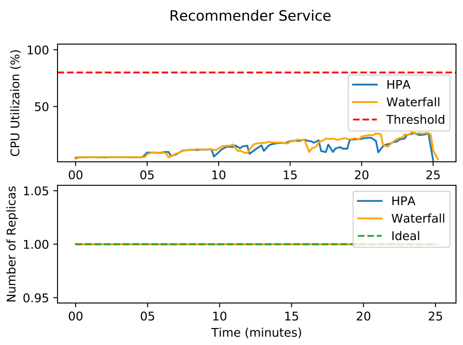

Figure 13: The CPU utilization and number of replicas for the Recommender service.

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 13 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 9, 10, and 11 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 14 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.

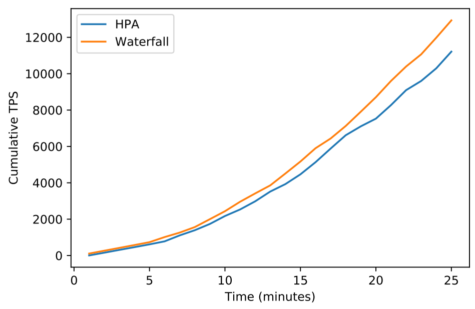

Figure 14: Cumulative Transaction Per Second (TPS) of Waterfall and HPA autoscalers.

Table 6: Comparison of Waterfall and HPA autoscalers in terms of performance metrics.

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

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 6 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 the following metrics for both autoscalers and presented them in Table 7:

- CPU>Threshold time: The percentage of time that CPU utilization of the service is above the threshold.
- Underprovision time: The percentage of time that the number of service replicas is less than the ideal state.
- Overprovision time: The percentage of time that the number of service replicas is more than the ideal state.

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.

Table 7: Comparison of Waterfall and HPA in terms of CPU>Threshold(T), overprovision, and underprovision time.

Service	CPU >T		Underprovision		Overprovision	
	HPA	Waterfall	HPA	Waterfall	HPA	Waterfall
Webui	∼31%	∼16%	∼54%	∼15.33%	0%	0%
Persistence	∼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%
Recommender	0%	0%	0%	%0	0%	0%

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

Amazon (2020b). Aws auto scaling. https://aws.amazon.com/autoscaling/. 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 tradeoff 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.

Iqbal, W., Dailey, M. N., and Carrera, D. (2015). Unsupervised learning of dynamic resource provisioning policies for cloud-hosted multitier web applications. *IEEE Systems Journal*, 10(4):1435–1446.

Jindal, A., Podolskiy, V., and Gerndt, M. (2019). Performance modeling for cloud microservice applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 25–32.

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). Auto-scaling 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.