Microscaler: Cost-effective Scaling for Microservice Applications in the Cloud with an Online Learning Approach

Guangba Yu, Pengfei Chen*, Zibin Zheng

Abstract—Recently, the microservice becomes a popular architecture to construct cloud native systems due to its agility. In cloud native systems, autoscaling is a key enabling technique to adapt to workload changes by acquiring or releasing the right amount of computing resources. However, it becomes a challenging problem in microservice applications, since such an application usually comprises a large number of different microservices with complex interactions. When the performance decreases due to an unpredictable workload peak, it is difficult to pinpoint the scaling-needed services which need to scale out and evaluate how many resources they need. In this paper, we present a novel system named *Microscaler* to automatically identify the scaling-needed services and scale them to meet the Service Level Agreement (SLA) with an optimal cost for microservice applications. *Microscaler* first collects the quality of service (QoS) metrics in the service mesh enabled microservice infrastructure. Then, it determines under-provisioning or over-provisioning service instances along the service dependency graph with a novel scaling-needed service criterion named *service power*. The service dependency graph could be obtained by correlating each request flow in the service mesh. By combining an online learning approach and a step-by-step heuristic approach, *Microscaler* can precisely reach the optimal service scale meeting the SLA requirements. The experimental evaluations in a microservice benchmark show that *Microscaler* achieves an average 93% precision in scaling-needed service determination and converges to the optimal service scale faster than several state-of-the-art methods. Moreover, *Microscaler* is lightweight and flexible enough to work in a large-scale microservice system.

Index Terms—Auto-scaling, Microservice, Service Mesh, Bayesian Optimization, Cloud Computing

1 Introduction

Driven by the promising features of cloud computing such as pay-as-you-go, elasticity, and on-demand, many modern enterprises have deployed their applications in the cloud [2]. In order to take advantage of cloud resources more efficiently and to accelerate the development and delivery process, those enterprises build or reconstruct their application architectures from monolithic to microservice. Nowadays, the microservice architecture has been applied widely in the modern IT industry. With this kind of architecture, an application is decoupled into many loosely distributed fine-grained services [3]. Each service has simple and independent functions following the Single Responsibility Principle (SRP) [4]. The microservice architecture allows applications to scale out/in for partial services in a fine granularity (e.g., container instance), which reduces the scaling cost compared with the conventional virtual machine scaling.

Human experts can approximately specify the resources that applications need under different workload patterns in the predictable scenarios where the workload patterns have seasonality or stability [5]. However, in many other cases, the workload of one application in the cloud is unexpected due to system noises and fluctuations, especially in a microservice environment. A practical approach to achieve elasticity in the production environment is autoscaling. It is crucial to automatically scale out/in with resources as few as possible on condition that meeting the SLA requirements. If there is no special statement, we treat scaling as horizontal scaling (i.e., the increase of running instances) rather than vertical scaling (i.e., the increase of resource for one single running instance). The under-provisioning of resources should be avoided during the scaling phase. On the other side, the over-provisioning leads to resource wasting and extra cost [6]. Therefore, the application provider desires a cost-optimal scaling mechanism without SLA violations.

Cloud computing provides a flexible resource allocation mechanism. But it heavily relies on the application owner to leverage the flexible infrastructure [7]. Autoscaling has been shipped by default in conventional cloud providers such as Amazon AWS Autoscaling [1]. However, an automatic scaling approach in microservice environments is notoriously difficult to implement due to the following challenges:

- **Service metric collection**. Most mainstream cloud platforms today do not provide adequate means to monitor applications in a fine granularity [8]. Therefore, it is difficult to obtain service-level performance metrics in the wild. Although the application-level instrumentation can achieve that, application devel-

- *Guangba Yu, Pengfei Chen, Zibin Zheng are with the School of Data and Computer Science, Sun Yat-sen University, Guangzhou 510006, China, and Zibin Zheng is also with the National Engineering Research Center of Digital Life, Sun Yat-sen University , Guangzhou 510006, China, E-mail: yubg5@mail2.sysu.edu.cn, {chenpf7, zhzibin}@mail.sysu.edu.cn*
This paper is an extended version of ICWS'19 paper Microscaler [1]

1. Amazon AWS Autoscaling [http://www.aws.amazon.com/cn/ec2-/autoscaling]

opers must be expertise to instrument their applications correctly if using it.

- **Scaling-needed service determination**. A large number of services co-exist in a microservice system and the interactions among them are complex. Hence, a performance anomaly may result in multiple anomalies in many services simultaneously. Therefore, it is non-trivial to determine the scaling-needed services quickly and accurately when a performance anomaly occurs.
- **Performance and cost trade-off**. An autoscaler should automatically scale the right amount of resource for services which are under-provisioning or over-provisioning to optimize the performance and cost of the application [9]. But there is not an explicit regulation model between the resource of each service and the request volume. The autoscaler needs to determine how much resource is required to scale effectively and efficiently.

Extensive methods have been proposed to solve auto-scaling problems for virtual machines in the cloud environment [9], [10]. However, a few methods like [1], [11], [12] can scale quantitatively in the dynamic microservice environment. Moreover, most existing methods need to modify the application source code to expose the performance metrics required by auto-scaling. To resolve the aforementioned problems and shortcomings of previous work, this paper proposes *Microscaler* to determine the scaling-needed services, and scale out/in quickly and optimally. It primarily comprises five procedures including service metric collection, service dependency graph construction, scaling-needed service determination, auto-scale decision and auto-scale action. *Microscaler* collects service metrics exposed by the service mesh enabled infrastructure continuously. Once an SLA violation of the front-end service is detected, the service call graph in the last five minutes is constructed based on the request execution paths recorded by service mesh. *Microscaler* leverages the correlation between services' latency metrics and *service power* to determine the scaling-needed services and to decide the number of service instances by combining a Bayesian Optimization model and a step-by-step heuristic model. In addition, *Microscaler* maintains a key-value map that keeps pairs of request volume and the optimal number of service instances to speed up the auto-scale decision procedure significantly. *Microscaler* works in a real-time mode to adapt to the dynamic workload. Moreover, we validate *Microscaler*'s effectiveness and efficiency in a production-class microservice benchmark, namely Hipster-shop [2], managed by Istio [3] (i.e., a popular service mesh) enabled Kubernetes [4] platform. The experimental evaluations show that *Microscaler* achieves an average 93% precision in scaling-needed services determination and converges to the optimal service scale faster than several state-of-the-art methods.

Generally speaking, the contributions of this paper are four-fold:

- We leverage the advantages of service mesh infrastructure to resolve the challenging problem of collecting service performance metrics for autoscaler in microservice systems.
- We propose a novel criterion named *service power* to confirm the scaling-needed services in microservice systems, which reduces unnecessary scaling actions.
- We combine an online learning approach and a step-by-step heuristic approach to build a cost model for auto-scaling. The model can obtain the optimal service scale with only a few iterations. In the meantime, we save the key-value between current workload and optimal service scale, this will accelerate the process of searching the optimal service scale when a similar workload appears.
- We design and implement a prototype, namely *Microscaler*, to evaluate the proposed autoscaling approach in a production-class microservice benchmark system. The experimental result shows that *Microscaler* is a promising approach.

The rest of this paper is organized as follows. The background and motivation are introduced in Section 2. Section 3 shows the basic idea and detailed design of *Microscaler*. In Section 4, we present our experimental setting and results. Finally, we discuss the related work in Section 5 and conclude this paper in Section 6.

2 Background and Motivation

2.1 Scaling in Microservice Applications

Scaling monolithic applications is challenging because there are lots of components and some of them may be executed more frequently than others. If some components need to be scaled when they are overloaded, the whole application will be scaled at the same time, which results in some under-loaded components that consume a large number of resources even they are not going to be scheduled [13].

To mitigate the difficulty of traditional monolithic applications in scaling, the microservice architecture is widely adopted. The concept of microservice can be roughly expressed as "one microservice performs one job, multiple service instances can work in parallel, and microservices are independent with each other" [14], which makes each microservice of one application can scale independently. Another benefit of the microservice architecture is its portability. One microservice is typically packaged in a container which includes the microservice and all its dependencies (e.g., libraries, databases). Thus, a microservice system is readily to scale horizontally if there are sufficient resources [15].

Although the microservice architecture provides some appealing features for scaling in the cloud environment, it is non-trivial to design a cost-effective auto-scaling system to adapt to the highly dynamic microservice environment. The challenges have been shown in Section 1. This paper tries to solve the above-mentioned problems in the following sections.

2.2 Service Mesh

We have known the inherent challenges associated with monolithic applications. The intuitive solution to these chal-

2. Hipster-shop [https://www.github.com/GoogleCloudPlatform-/microservices-demo]
3. Istio [http://www.istio.io]
4. Kubernetes [http://www.kubernetes.io]

lenges is to decompose them into microservices. While this simplifies the development of individual services, orchestrating and operating hundreds or even thousands of microservices are not so simple. Recently, a solution was to combine them together using customized scripts, libraries, and dedicated engineers tasked with managing these distributed systems. This increases maintenance costs with no doubt.

To make the development and orchestration of microservice systems easy, the concept of service mesh is proposed recently [16]. A service mesh is a dedicated infrastructure layer making service-to-service communication safe, reliable, observable and configurable. It provides a transparent and language-independent way to flexibly and easily automate networking, security, and telemetry functions. In essence, it decouples development and operations for services. Therefore, a developer can deploy new services as well as make changes to existing ones without worrying about how that will impact the operational properties of their distributed systems. Similarly, an operator can seamlessly modify operational controls across services without redeploying them or modifying their source code. This layer of infrastructure between services and their underlying network is what is usually referred to as a service mesh.

A service mesh consists of two high-level components, namely control plane and data plane. In modern service mesh driven microservice systems, the data plane is implemented as a network proxy (e.g., Envoy [5]) that is deployed alongside each service as a sidecar without the awareness of applications. Klein [17] states that within a service mesh, the data plane touches every packet/request in the system and is responsible for service discovery, health checking, routing, load balancing, authentication/authorization, and observability. A control plane takes control of all the individual instances of the data plane and turns them into a distributed system. The control plane does not touch any packets/requests in the system, and instead, it allows a human operator to provide policy and configuration for all of the running data planes in the mesh. The control plane also enables the data plane telemetry to be collected, centralized, and ready for consumption by an operator.

Istio is an open-source service mesh that sits transparently on existing distributed applications. As a promising service mesh, Istio offers a full-fledged solution to satisfy diverse requirements (e.g., metrics collecting, transaction tracing) of microservice applications [18]. To achieve that, Istio provides a high-performance proxy to mediate all inbound and outbound network traffics for all services. The service metrics provided by Istio will help us determine when to scale and how to scale. Istio leverages proxies to capture network traffics, where possible, automatically program the networking layer to route traffics through those proxies without any changes to the deployed applications. In Kubernetes, the proxies are injected into pods and network traffics are captured by programming iptables rules. Once the Istio proxies are deployed, Istio can mediate all traffics. In this paper, we construct our system on Istio. But the methods can also be applied to other service mesh that is driven microservice systems.

5. Envoy [https://www.envoyproxy.io/]

2.3 Motivation

Although some cloud platforms like Kubernetes have been equipped with horizontal autoscaling modules [6], they are mainly triggered by the exceeding of resource limits (e.g., higher than 80% CPU utilization). However, in microservice applications, the correlation between CPU or memory utilization and service QoS might be not strong. The deatils are shown in Section 3.2. Therefore, traditional approaches based on CPU or memory load will not perform well in microservice systems. The ultimate goal of autoscaling is to continually optimize the QoS (e.g., request latency or throughput) and cost. Thus their SLA and cost budget requirements can be better satisfied.

On the other hand, with the ever-growing scale and complexity of modern micro-service applications, if we cannot locate the microservices which are under-provisioning or over-provisioning, autoscaling cannot resolve performance problems. However, existing autoscaling methods towards Virtual Machine (VM) based cloud environments are only concerned about scaling the number of VMs rather than the above-mentioned localization. Moreover, making effective scale decision to utilize resources efficiently is challenging as well. Following these motivations, we propose *Microscaler* to determine scaling-needed services and conduct cost-optimal autoscaling.

3 System Design

3.1 System Overview

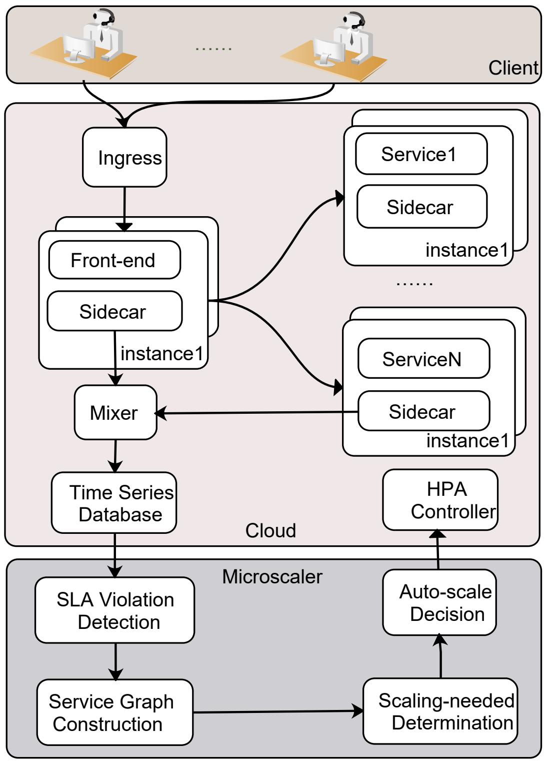


Fig. 1. System overview of *Microscaler*.

6. Kubernetes Autoscaler [http://www.kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale]

Fig. 1 shows the system overview of *Microscaler*. Microservice applications are deployed in a service mesh driven cloud platform. The issued request will go through the ingress of service mesh before visiting the *frontend* of the applications. At the service mesh's data plane, a network proxy is deployed as a sidecar to each relevant service instance (i.e., container). A sidecar cloud be recognized as an additional container which shares namespace with the primary container in order to enhance the functionality of this primary container. At the service mesh's control plane, a telemetry component (e.g., Mixer [18]) collects all network traffics from sidecar and stores them in a time-series database (e.g., Prometheus [7]).

Microscaler continually monitors the performance metrics of the frontend service within a sliding window. The gray background of Fig. 1 shows the process of how *Microscaler* reacts to the dynamic change of applications. When the *SLA Violation Detection* module detects an SLA violation, the *Service Graph Construction* module will construct the dependency graph of service instances in realtime. Then the *Scaling-needed Service Determination* process will be triggered to output the scaling-needed services based on the service dependency graph. After that, the *Auto-scale Decision* module will generate how many instances should be scaling out/in for each scaling-needed service. Finally, *Microscaler* issues a command to the cloud replica controller according to the results of *Auto-scale Decision* to perform horizontally scaling.

3.2 SLA Violation Detection

SLA provides information on the scope, the quality and the responsibilities of each microservice and its provider [19]. They include SLOs that define the detailed, measurable conditions. The service mesh infrastructure provides metrics about requests and responses and gives metrics about client and service workloads for each individual service within the mesh. Hence, *Microscaler* can easily obtain SLO metrics from service mesh for monitoring services and determining scaling-needed services.

Microservices are monitored periodically (e.g., per 10 seconds) to keep consistent with the SLA in face of fluctuated request volumes. The SLA of one service is specified by its providers before the service is deployed. A performance anomaly, then, is an event or a collection of events that cause SLA to be violated. In this paper, *Microscaler* selects the service request latency to denote the application performance rather than the resource utilization (e.g., CPU utilization) or the request volume since this metric directly expresses users' experience. The request latency is defined as the duration between request and response.

We deploy a microservice limited by 1-core CPU, 2GB of memory in a physical machine by Docker [8]. There are other processes that occupy most of the IO resources running on the same physical machine. We gradually increase the number of requests issued to this microservice from 10/s to 100/s. In Fig. 2, P90 is the average latency for the slowest 10 percent of requests over the last 30 seconds and the P50 is the average latency for the slowest 50 percent of requests

7. Prometheus [http://www.prometheus.io]
8. Docker [http://www.docker.com]

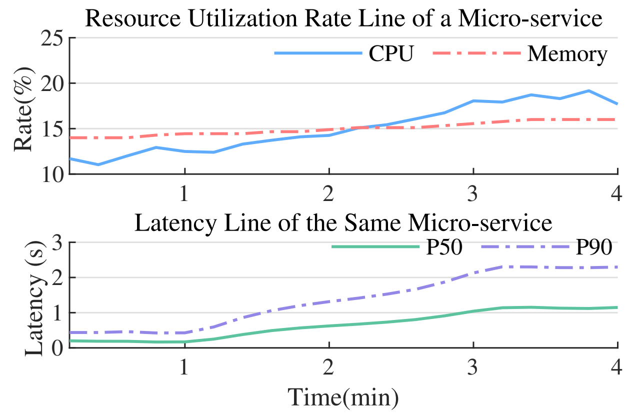
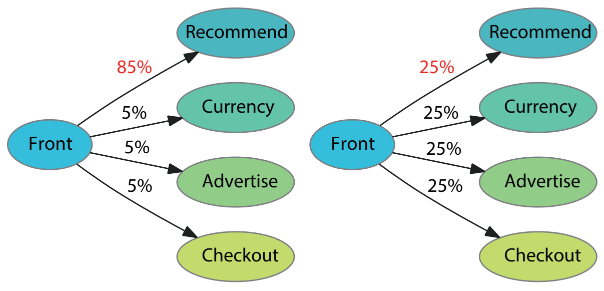


Fig. 2. A service that has low CPU and memory utilization but high service latency.



(a) Requests congest on the same service. (b) Requests are dispatched to different services evenly.

Fig. 3. Same requests to the front-end with different dispatch methods.

over the last 30 seconds. As shown in Fig. 2, although the microservice instance which has a low CPU load and memory utilization, it provides an unacceptable service request latency which exceeds 2 seconds in P90 because of the limited IO resources. Our previous work [1] also shows that a high CPU load only means that the microservice instance is fully utilized. But it can still provide an acceptable service request latency with no need to scale. Therefore, making the resource utilization as the trigger for autoscaling may be insufficient.

If taking the request workload as a trigger, it is not so sure when to conduct autoscaling as the requests will be dispatched to different down-streaming services. Considering that a small request volume congests on the same microservice like Fig. 3(a), even though the workload at that time does not trigger scaling, the request latency may still exceed SLA. On the contrary, if a huge workload volume could be distributed to different service instances in an appropriate manner such as shown in Fig. 3(b), it may still provide an acceptable SLA. In short, *Microscaler* chooses the service request latency of the front-end service to trigger autoscaling since it can avoid triggering unnecessary scaling actions [9].

The SLA-based anomaly detector of *Microscaler* allows service owners to specify SLA for deployed applications. An SLA comprises both an upper bound T_{max} and a lower bound T_{min} of the *frontend* service's request latency. *Microscaler* obtains the realtime latency and workload by querying time-series database and monitors the front-end service's request latency every 10 seconds. If the latency of the front-end service is higher than T_{max} or lower than T_{min}, *Microscaler* will consider that the SLA has been violated. And to prevent the detector from detecting a similar anomaly multiple times, we set the detection time window to 5 minutes to smooth the detection result. A side effect of this method is that the detector is unable to detect another violation until the time window is filled again which is also mentioned in [8]. Although this static threshold-based approach cannot adapt to the dynamic microservice environments such as normal updates, it works in this paper. Actually, our previous work has proposed an adaptive anomaly detection approach for microservice applications [20], which will be incorporated in future work.

Especially, this paper assumes that the instantaneous and minuscule requests which are abnormal may not be caused by the application itself. These extreme cases should not trigger scaling. In order to address this problem, *Microscaler* uses P90, namely the average latency for the slowest 10 percent of requests over the last 30 seconds, to trigger scaling. *Microscaler* can filter those extreme requests efficiently and avoid unnecessary autoscaling actions simultaneously. Moreover, a percentile-based KPI is validated to be robust in anomaly detection in the cloud environment mentioned in [21] which is consistent with our observation.

3.3 Service Dependency Graph Construction

Creating the service dependency graph before scaling-needed services determination is helpful both in understanding the whole application and determining scaling-needed services quicker. However, nowadays a web application of microservice architecture could be developed by many different teams and the number of microservices is large, the operators can hardly know the entire service dependency graph as a whole.

In previous studies, Sherlock [22] adopted tracing technology to acquire system components' dependency graph, but they cannot be easily deployed because it needs to instrument source code. CauseInfer [23] used the network connection information to construct a service causality graph. However, the direction of dependency between two service instances is determined by statistical methods, which introduces uncertainty. Other tracing based approaches like [24] and [25] need to intercept system calls and cannot obtain the service latency metric.

In this paper, *Microscaler* collects all service invocation links in realtime with the help of service mesh and constructs a service dependency graph to represent all links. Because the network proxies are injected into pods and network traffics are captured by programming iptables rules, for each request issued to one service instance, the injected proxy knows which service it is from or which service it will go to. Fig. 4 shows a request record collected by service mesh. In this example, the destination-app is *adservice* and the source-app is *frontend* which means that *frontend* calls *adservice*, so the *Microscaler* will get *frontend* $\rightarrow$ *adservice*.

Therefore, *Microscaler* can extract a bunch of dependencies that describe the invocation of the services from a microservice to callee microservice by querying the request records collected by service mesh and then combine all the dependencies to output the complete service dependency graph.

```
istio_requests_total {
  connection_security_policy = "none",
  destination_app = "adservice",
  destination_service_name = "adservice",
  destination_service_namespace = "hipster-shop",
  instance = "172.20.1.245:42422",
  job = "istio-mesh",
  request_protocol = "grpc",
  response_code = "200",
  source_app = "frontend",
}
```

Fig. 4. A record of request collected by service mesh.

When the *Service Graph Construction* module is triggered by the *SLA Violation Detection* module, it queries the request records collected by service mesh in the last 5 minutes and then extracts and combines service dependency from the records. Therefore, the service dependency graph can be updated dynamically, which will exclude some inactive microservices and improve the precision of scaling-needed services determination results.

3.4 Scaling-needed Service Determination

There may be hundreds of microservices in an application and the interactions among them are very complex. Hence, it is thorny to determine which is the microservice needed to scale when an SLA violation occurs. *Microscaler* introduces the scaling-needed service determination to overcome this problem. When an SLA violation emerged, the scaling-needed service determination module attempts to find, among all the dependent microservices, the services that are likely over-provisioning or under-provisioning. When the latency of the front-end service exceeds the upper bound T_{max}, the scaling-needed service determination algorithm starts to work as shown in Algorithm 1.

The scaling-needed service determination starts from the *frontend* service and calculates the correlation coefficient r between adjacent services in the service dependency graph by traversing the graph along the edges. The details are shown in Section 3.4.1. Unfortunately, whacking confidence intervals on an estimate of the correlation coefficient is not straightforward. Therefore, *Microscaler* converts each r to a Fisher'z [26] which is shown in Section 3.4.2 then judges whether they are correlated. If one service is correlated with its downstream services, *Microscaler* keeps searching other services along the service call graph. Otherwise, it terminates the search process. This will expedite the process of determining scaling-needed services. On the other hand, searching along the service call graph is more lightweight than the brute force searching. We consider the services which are correlated with its upstream services but uncorrelated with its all downstream services as the scaling-needed candidate services. Then the *Microscaler* further confirms scaling-needed services by a novel proposed criteria *service power* which is shown in Section 3.4.3.

Fig. 5 shows an example of scaling-needed service determination process and the *product* is the scaling-needed

Algorithm 1 Scaling-out-needed service determination algorithm

Input: An original abnormal service instance namely the front-end service *front*, a service dependency directed graph G, service power limit $\mathbb{P}_{out}$, the confidence level α.

Output: A list of scale-out services *scaleList*.

```
1:  queue ← Queue()
2:  scaleList ← List()
3:  visited ← Dict()
4:  queue.add(front)
5:  // Find scaling-needed service based on service dependency graph.
6:  while queue is not empty do
7:      node ← queue.pop()
8:      //flag is to identify whether all the downstream services of node are uncorrelated.
9:      flag ← false
10:     visited[node] ← true
11:     // The adj(G,X) represents the neighbors which are adjacent to X in G.
12:     for all neigh ∈ adj(G,node) && ! visited[neigh] do
13:         // Calculated the correlation between adjacent service.
14:         r ← Correlation(neigh, node)
15:         // Transform r → z using Fisher's Z-transform.
16:         $z \leftarrow arctanh(r)$
17:         $\zeta_u \leftarrow z + z_{critical}(1-\alpha/2) * \sqrt{\frac{1}{num-3}}$
18:         $\zeta_l \leftarrow z - z_{critical}(1-\alpha/2) * \sqrt{\frac{1}{num-3}}$
19:         $r_u \leftarrow \tanh(\zeta_u)$
20:         $r_l \leftarrow \tanh(\zeta_l)$
21:         if $r_u * r_l > 0$ then
22:             // IF neigh and node are correlated, keeping searching in this invocation list.
23:             queue.add(neigh)
24:             flag ← true
25:         end if
26:     end for
27:     if $node.P90/node.P50 > \mathbb{P}_{out}$ && ! flag then
28:         // If all neigh of node are uncorrelated with node and $node.P90/node.P50$ exceeds limit.
29:         scaleList.add(node)
30:     end if
31: end while
32: return scaleList
```

service. The search process begins at *front*, *Microscaler* calculates the correlation between *front* and *recommend*, *front* and *checkout*. It concludes that *recommend* is correlated with *front* and *checkout* is uncorrelated with *front*. So *Microscaler* will keep searching the *product* and *cart*, and it will ignore the *currency* and *payment*. Then *Microscaler* repeats the process in the *recommend*'s invocation list and find that *product* is correlated with *recommend* and *cart* is uncorrelated with *recommend*. There is no other service which is correlated with *product*, so if *product*'s $P90/P50$ exceeds $\mathbb{P}_{out}$ *Microscaler* will consider *product* as the scaling-needed service.

For the scaling-in-needed service, we only need to change the *service power* rule like

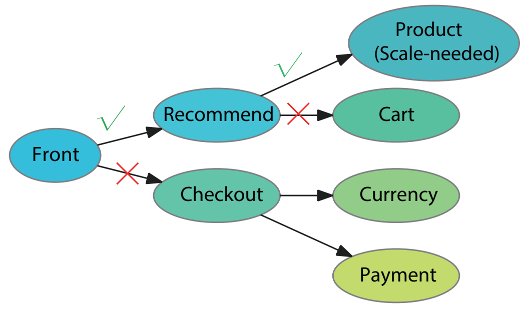


Fig. 5. Service dependency graph and the search process of scaling-needed service.

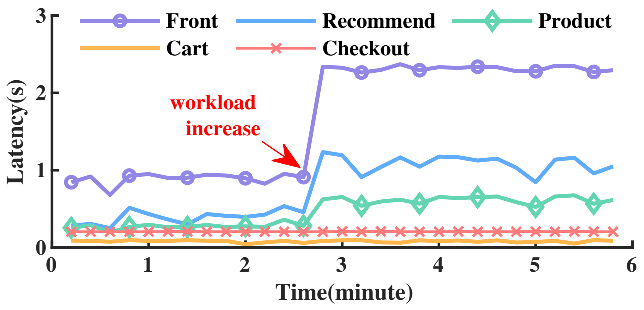


Fig. 6. Curve of request latency among the Fig. 5 services.

$$service_i.P90/service_i.P50 < \mathbb{P}_{in}.$$

3.4.1 *Correlation Coefficient*

According to our previous work Microscope [3], if two service instances have a strong dependent relationship, the curves of their request latency are similar. For example, Fig. 5 shows the service dependency graph and Fig. 6 shows the curves of service latency of five services in the last 5 minutes when the application's workload surged. In Fig. 6 we can observe that when the workload to *front* increases, the *front*'s latency rises up and the latency curves of *recommend* and *product* rise up too, but the *cart* and the *checkout* which is on the other invocation chain tend to be stable. Furthermore, we also can conclude that scaling-needed *product* causes the latency of *recommend* and *front* to increase. Therefore, *Microscaler* tries to find the most possible microservices that cause the anomaly of the front-end service by calculating the correlation coefficient of service request latency between the adjacent microservices in the service dependency graph.

There are many kinds of correlation coefficients that can express associations. Pearson product-moment correlation coefficients (PCC) and Spearman rank-order correlation coefficients (SCC) are the most commonly used measures of monotone association, with the latter usually suggested for no-normally distributed data [27]. In this work, we adopt the above two criteria to calculate the association between microservices. PCC is a parametric measure of association between two continuous variables. It is defined as the ratio of the covariance of the two variables to the product of their respective standard deviations [27]. The formula for the population PCC, denoted by ρ, is

$$\rho = \frac{\mathrm{E}((x-\mathrm{E}(x))(y-\mathrm{E}(y)))}{\sqrt{\mathrm{E}(x-\mathrm{E}(x))^2\mathrm{E}(y-\mathrm{E}(y))^2}}. \tag{1}$$

The population correlation, ρ, is usually not known. Therefore, the sample statistic, r_ρ, is used to estimate ρ and to carry out tests of hypotheses. The r_ρ can be obtained by plugging-in the covariance and the deviations into the above formula:

$$r_\rho = \frac{\sum_i\left(\left(x_i-\overline{x}\right)\left(y_i-\overline{y}\right)\right)}{\sqrt{\sum_i\left(x_i-\overline{x}\right)^2\sum_i\left(y_i-\overline{y}\right)^2}}, \tag{2}$$

where

$$\overline{x} = \frac{\sum_{i=1}^n x_i}{n}, \overline{y} = \frac{\sum_{i=1}^n y_i}{n}.$$

SCC is a rank-based version of the PCC [27]. Its sample correlation coefficient denoted by r_s can be written as follows:

$$r_s = \frac{\sum_i\left(\left(R_i-\overline{R}\right)\left(S_i-\overline{S}\right)\right)}{\sqrt{\sum_i\left(R_i-\overline{R}\right)^2\sum\left(S_i-\overline{S}\right)^2}}, \tag{3}$$

where R_i is the rank of x_i and S_i is the rank of y_i. And

$$\overline{R} = \frac{\sum_{i=1}^n R_i}{n}, \overline{S} = \frac{\sum_{i=1}^n S_i}{n}.$$

Both PCC and SCC take values between -1 and $+1$. The value of r_ρ or r_s describes the strength of the monotonic relationship. For example, when the correlation coefficient is positive (e.i., $r_\rho > 0$), it means that one variable increases with the other.

3.4.2 Fisher's z Transformation

Whacking confidence intervals on an estimate of PCC or SCC is not so straightforward. This is because PCC and SCC are bound between -1 and +1. Moreover, they are not normally distributed. We cannot make the decision only by these two correlation coefficients. That means we are unable to set one threshold based on SCC or PCC to determine whether two metrics are correlative. The significance of correlation coefficients needs to be tested. Therefore, we introduce Fisher's Z-transform to achieve that, shown as:

$$z_r = \tanh^{-1}(r) = \frac{1}{2}\log\left(\frac{1+r}{1-r}\right), \tag{4}$$

where r is the sample correlation coefficient. In PCC the r is r_ρ and in SCC the r is r_s.

The confidence limits for the correlation are derived through the confidence limits for the parameter ζ. The two-sided confidence limits for ζ are computed as

$$\zeta_l = z_r - z_{\mathit{critical}}(1-\alpha/2)\sqrt{\frac{1}{n-3}}, \tag{5}$$

$$\zeta_u = z_r + z_{\mathit{critical}}(1-\alpha/2)\sqrt{\frac{1}{n-3}}, \tag{6}$$

where z_{critical} an easily be obtained from the z-table for α given confidence level (e.g., 1.96 in the case of 95% confidence interval), n is the sample size.

These confidence limits of ζ_l and ζ_u are then transformed back to derive the confidence limits for the correlation:

$$r_l = \tanh\left(\zeta_l\right) = \frac{\exp\left(2\zeta_l\right)-1}{\exp\left(2\zeta_l\right)+1}, \tag{7}$$

$$r_u = \tanh\left(\zeta_u\right) = \frac{\exp\left(2\zeta_u\right)-1}{\exp\left(2\zeta_u\right)+1}. \tag{8}$$

If the confidence interval involves 0, we regard that the population correlation is not significantly different from zero, at a given level of confidence [28]. Thus, *Microscaler* considers that the adjacent services are uncorrelated if $r_l * r_u < 0$.

3.4.3 Service Power

In order to avoid doing unnecessary scaling in some situations that could not be solved by autoscaling (e.g., software bug), we propose a novel criterion to determine whether it is necessary to scale. *Microscaler* leverages the ratio between P_{50} and P_{90} to represent the *service power* denoted by $\mathbb{P}$, namely $\mathbb{P} = \frac{P_{90}}{P_{50}}$. $P50$, i.e., the average latency for the slowest 50 percent of requests over the last 30 seconds, roughly reflects the overall performance of the system. $P90$ is the average latency for the slowest 10 percent of requests over the last 30 seconds, it can reflect the performance outliers which the average value may mask, for example, that 10% of requests are not having a good experience. Here, we choose $P90$ rather than $P99$ or $P95$ to calculate $\mathbb{P}$ since this percentage can mitigate the impact of jitters brought by normal changes.

If P_{90} exceeds P_{50} too much (e.g., $\frac{P_{90}}{P_{50}} > \mathbb{P}_{out}$, $\mathbb{P}_{out} = 2$ in this paper), it means that the microservice could only handle a small part of requests. About 10 percent of requests to the microservice could not be processed in time. In other words, there have been many requests in the processing queue, which exceeds the normal *service power* at that moment. Therefore, the *service power* is low and the microservice needs to scaling-out. If P_{50} is close to P_{90} (e.g., $\frac{P_{90}}{P_{50}} < \mathbb{P}_{in}$, $\mathbb{P}_{in} = 1.3$ in this paper), it means that the microservice could handle most of requests. Therefore, the *service power* is strong and the microservice can be scaled-in to save cost.

Fig. 7 and Fig. 8 show service power $\mathbb{P}$ changes with workload increase and decrease in different services respectively. With the workload increasing, the percent of requests which could not be processed also increases. It can be observed that P_{90} increases faster than P_{50} at the beginning. Therefore, the line of $\mathbb{P}$ in Fig. 7 rises up first. Then it becomes stable when $\mathbb{P} \approx 2$, that is why we select $\mathbb{P}_{out} = 2$ to trigger scaling out in this paper. But with the workload increasing, P_{50} is close to P_{90} again. Thus, the line of $\mathbb{P}$ declines. At this time, using *service power* to confirm scaling-out-needed services will be unreasonable. In Fig. 8, the line of $\mathbb{P}$ decreases as the workload decreases. Finally, it keeps stable at about 1.3. That is why we select $\mathbb{P}_{in} = 1.3$ to determine to scale in in this paper. We admit that $\mathbb{P}_{in}$ and $\mathbb{P}_{in}$ may be different in different microservice systems. However, we can find appropriate values through the stressing test.

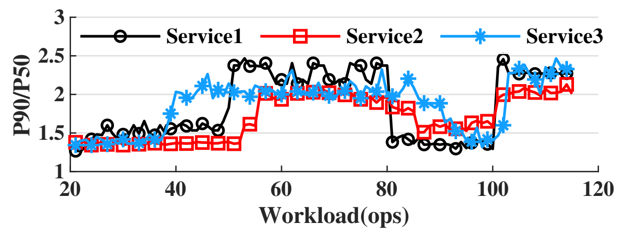


Fig. 7. Service power $\mathbb{P}$ changes along with workload increase.

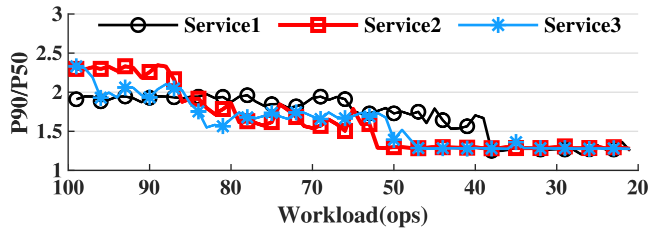


Fig. 8. Service power $\mathbb{P}$ changes along with workload decrease.

3.5 Auto-scale Decision

After finding out services which need to scale, it is of equal importance to determine how many instances should be scaled out/in precisely and quickly. Building a complex prediction model for the coming workload is at one end of the spectrum. However, a large amount of training data is required for building such a complex model which is not viable since one application may have hundreds of microservices. Moreover, even given enough data, the prediction performance still remains an issue because microservices may be updated frequently.

From the black-box perspective, *Microscaler* combines Bayesian Optimization (BO) [29] and a step-by-step heuristic approach to solve the above problem. *Microscaler* leverages BO to find a near-optimal result with only a few searching iterations then uses a step-by-step approach to reach the optimal result. In addition, *Microscaler* maintains a map-table between the workload and the optimal scale for each microservice to speed up the search process. The details are shown in Section 3.5.1.

Algorithm 2 shows the searching process of optimal service scales. If *Microscaler* finds that map-table has stored the current workload, it will directly select the replicas number corresponding to the workload. Otherwise, it begins with the replicas number whose corresponding workload is the closest to the current workload in the map-table. BO then dynamically picks the next service replicas number based on the model shown in Section 3.5.2. The performance data is fed into the performance model under new service scales. The searching process will stop and output the sub-optimal or optimal number of service instances when it exceeds the maximum iterations. Finally, *Microscaler* leverages a step-by-step approach to find the optimal number and put the key-value pair between workload and optimal replicas number in the map-table.

3.5.1 *Map-table*

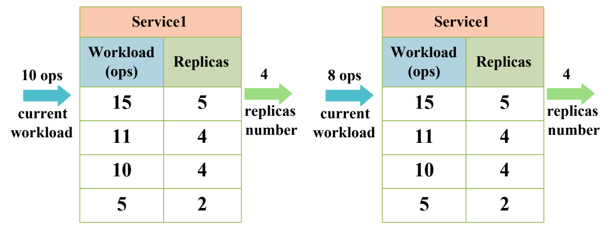
Microscaler maintains a map-table for each microservice. On one hand, if the map-table has involved the current workload of scaling-needed service, the map-table will output the corresponding number of replicas directly. For example in Fig. 9(a), the current workload of *service1* is 10 ops. We can find that *service1*'s map-table has included 10 ops and the corresponding 4 replicas of service instances. Therefore, *Microscaler* can obtain the target service scale quickly by a query operation.

On the other hand, we will select the workload which is the nearest to the current workload and output its corresponding replicas number for BO Initialization. In Fig. 9(b), the current workload of *service1* is 8 ops. There is not a corresponding workload towards 8 ops in this map-table. In

Algorithm 2 Auto-scale Decision Algorithm

Input: Scaling-needed service workload *Workload*, scaling-needed service hash-table *Table*, acquisition function *Acq*, surrogate model M.
Output: Optimal service replica number, *Num*.

```
1:  if $Workload \in Table$ then
2:      $Num = Table\ (Workload)$
3:  else
4:      $Num_0 := Initial(Workload, Table)$
5:      Obtain new $Latency_0$ after scaling instance number to $Num_0$
6:      $Cost_0 \leftarrow Num_0 * Latency_0$
7:      $D \cup (Num_0, Latency_0, Cost_0)$
8:      for i =1 ,2, ... do
9:          Update surrogate model $G \leftarrow FitModel(M, D)$
10:         $Num_i \leftarrow \arg\min_{Num} Acq(Num, G)$
11:         Obtain new $Latency_i$ after scaling instance number to $Num_i$
12:         $Cost_i \leftarrow Num_i * Latency_i$
13:         $D \cup (Num_i, Latency_i, Cost_i)$
14:         if meeting stopping cirteria then
15:             break
16:         end if
17:     end for
18:     $Num, Latency \leftarrow \arg\min_{Cost} D$
19:     for $Latency < T_{max}$ do
20:         $Num \leftarrow Num - 1$
21:         Obtain new Latency after scaling instance number to Num
22:     end for
23:     $Num \leftarrow Num + 1$
24: end if
25: Update Table (Workload,Num)
26: return Num
```



Service1	
Workload (ops)	Replicas
15	5
11	4
10	4
5	2

Service1	
Workload (ops)	Replicas
15	5
11	4
10	4
5	2

(a) Selecting replicas number for existing workload. (b) Selecting replicas number for non-existing workload.

Fig. 9. Map-table updates for different workload.

this situation, we will select the nearest workload to 8 ops, i.e., 10 ops. Then 4 is chosen for BO Initialization. Finally, *Microscaler* will insert 8 ops and its optimal service scale to the map-table.

3.5.2 Bayesian Optimization Approach

BO is an online learning method for optimizing expensive functions in a black-box way. Since it is non-parametric, it does not have any pre-defined assumptions for the cost model [10]. Mathematically, BO aims to find a global minimum (or maximum) of an unknown objective function

$$f: x^* = \arg\min_{x \in \mathbb{X}} f(x), \tag{9}$$

where $\mathbb{X}$ is the decision space of interest and is often a compact subset of $\mathbb{R}^d$. In this paper, $d = 1$ and $\mathbb{X}$ represents a continuous space spanned by the service instance number. Compared to state-of-the-art methods, BO can dynamically adapt its searching scheme based on the current understanding and confidence interval of the performance model to find the optimal or sub-optimal service replicas number. Furthermore, BO typically needs a small number of samples to find an optimal or sub-optimal solution because BO focuses its search on areas that have the largest expected improvements [10]. While Deep Neural Network [30] can also be used for black-box optimization, it requires lots of training samples which are very difficult to be adopted in our system.

Given scaling-needed services and the arriving workload, our goal is to find the optimal service scale to minimizes the cost of each visit under the condition that satisfying the SLA requirement. In a microservice system, the latency of *frontend* depends on the number of service replicas vector $\vec{x}$ comprised by many other services, denoted by $Latency(\vec{x})$. Since the number of services replicas that are not necessary to scale will not be updated in BO, the cost model only considers the services in the scaling list. Let $Price(\vec{x})$ be the total price in one time unit (e.g., 1 second) for all service instances in the scaling list. Here, the price could be obtained by multiplying the number of service instances and the price for each service instance. We formulate the cost model as follows:

$$\begin{aligned} &\min_{\vec{x}} Cost(\vec{x}) = Price(\vec{x}) * Latency(\vec{x}), \\ &subject\ \ to\ \ T_{min} \leqslant Latency(\vec{x}) \leqslant T_{max} \end{aligned} \tag{10}$$

where T_{max} is the SLA upper bound and T_{min} is the SLA nether bound. And $Cost(\vec{x})$ is the total cost of one request, furthermore, the explicit expression of $Cost(\vec{x})$ is unknown beforehand but can be observed through experiments. Therefore, we choose BO, a black-box optimization method, to resolve this problem.

There are two major choices that must be made when performing BO in Algorithm 2. First, we must select a surrogate model that will express the objective function being optimized. *Microscaler* chooses the Gaussian Process (GP) due to its flexibility and tractability [31].

GP is used as the prior distribution of the target function. It is a kind of prior distribution over functions of the form $f : \mathbb{X} \to \mathbb{R}$, and is defined by a mean function $\mu : \mathbb{X} \to \mathbb{R}$ and a positive definite covariance function (also called *kernel*) $\kappa : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$. Both the mean and covariance function are used to express some prior knowledge about the function being optimized.

In GP, for a series of points $x_{1:t}$ in $\mathbb{X}$, the function value $f(x_{1:t})$ accords to a multivariate normal distribution, that is,

$$f(x_{1:t}) \sim \mathcal{N}(\mu(x_{1:t}), \mathcal{K}(x_{1:t}, x_{1:t})), \tag{11}$$

in which $\mathcal{K}(x_{1:t}, x_{1:t})_{i,j} = \kappa(x_i, x_j)$ represents the covariance matrix.

Suppose we have n observations on f, and want to estimate the value of a new point x. Then we can let $t = n + 1$, $x_{n+1} = x$, and the prior over these $n + 1$ points, $[f(x_{1:t}), f(x)]^T$, is given by equation 11. And the posterior distribution of $f(x)$ can be computed using Bayes' theorem as,

$$f(x)|f(x_{1:n}) \sim \mathcal{N}(\mu_n(x), \sigma_n^2(x)), \tag{12}$$

in which

$$\mu_n(x) = \mathcal{K}(x, x_{1:n})\mathcal{K}(x_{1:n}, x_{1:n})^{-1}(f(x_{1:n}) - \mu(x_{1:n})) + \mu(x) \tag{13}$$

, and

$$\sigma_n^2(x) = \mathcal{K}(x, x) - \mathcal{K}(x, x_{1:n})\mathcal{K}(x_{1:n}, x_{1:n})^{-1}\mathcal{K}(x_{1:n}, x). \tag{14}$$

In short, using GP, for a series of observations $\{x_i, y_i\}_{i=1}^n$, we can calculate the posterior distribution of f at any point $x \in \mathbb{X}$, in the form of $f(x) \sim \mathcal{N}(\mu_x, \sigma_x^2)$.

Next, *Microscaler* must select an *acquisition function*, which is the utility function for determining whether a point $x \in \mathbb{X}$ is suitable as the next observation point. It has the form $a : \mathbb{X} \to \mathbb{R}^+$ and its value represents the degree to which a point x is suitable for improving the cost model. The maximum point of acquisition function, $x_{next} = \arg\max a(x)$, will be used as the next point of f for evaluation.

There are three main strategies to implement an acquisition function: (i) Probability of Improvement (PI) – choosing the point maximizing the probability of improving the current best [32]; (ii) Expected Improvement (EI) – picking the point maximizing the expected improvement over the current best [31]; and (iii) Gaussian Process Upper Confidence Bound (GP-UCB) exploiting lower confidence bounds (upper, when considering maximization) to construct acquisition functions that minimize regret over the course of their optimization [33]. In this paper, we concentrate on Expected Improvement (EI), as it has been shown to be better-behaved than PI [29]. Moreover, it does not require much parameter tuning compared with GP-UCB [31].

One previous work [34] has proposed an easy-to-compute form for EI. Let X_t be the sample of all replicas number whose cost values have been observed by round t, and $m = \min_{\vec{x}} C(\vec{x}) | \vec{x} \in X_t$ as the minimum function value observed so far. For each input $\vec{x}$ which is not observed yet, we can evaluate its expected improvement if it is picked as the next point to observe with the following equation:

$$EI(\vec{x}) = \begin{cases} (m - \mu(\vec{x}))\Phi(Z) + \sigma(\vec{x})\phi(Z), & \text{if } \sigma(\vec{x}) > 0 \\ 0, & \text{if } \sigma(\vec{x}) = 0 \end{cases} \tag{15}$$

where $\sigma(\vec{x}) = \sqrt{k(\vec{x}, \vec{x})}$, $Z = \frac{m - \mu(\vec{x})}{\sigma(\vec{x})}$, and Φ and ϕ are standard normal cumulative distribution function and the standard normal probability density function respectively.

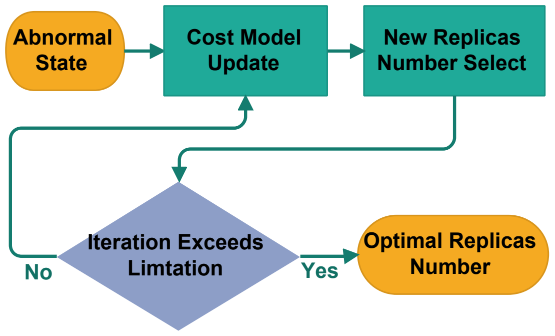


Fig. 10. The workflow of Bayesian Optimization Approach.

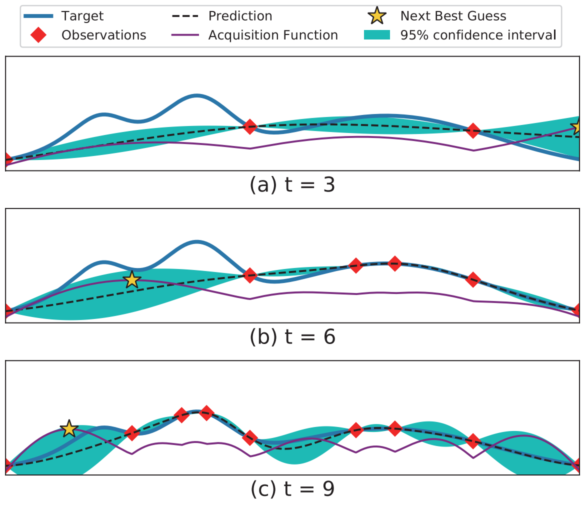


Fig. 11. An example of BO working process.

The acquisition function shown in Eqn(15) is designed to minimize *Cost*$(\vec{x})$ without constraints. However, $T_{min} \leqslant$ *Latency*$(\vec{x}) \leqslant T_{max}$ needs to be considered. It means that when selecting the next number of replicas to evaluate, *Microscaler* should search towards one that is likely to satisfy the SLA constraint. As proposed in [35], we modify the EI acquisition function as:

$$\begin{aligned} EI(\vec{x})' &= Pr\left[T_{\min} \leq Latency(\vec{x}) \leq T_{\max}\right] \times EI(\vec{x}) \\ &= \int_{T_{\min}}^{T_{\max}} p\left(Latency(\vec{x})|\vec{x}\right) dLatency(\vec{x}) \times EI(\vec{x}), \end{aligned} \tag{16}$$

where Pr denotes the possibility of the latency of the selected samples is greater than $T_{\min}$ and less than $T_{\max}$. Conveniently, due to the marginal Gaussianity of *Latency*$(\vec{x})$, $Pr\left[T_{\min} \leq Latency(\vec{x}) \leq T_{\max}\right]$ is a simple univariate Gaussian cumulative distribution function [34].

Fig. 10 shows the workflow of the adaptive online learning approach. The process of optimal service scale searching starts from an abnormal state. BO then dynamically picks the next service replicas number to test based on the cost model and feeds the result to the cost model. The procedure will stop and generate the obtained optimal number when the maximum iteration is exceeded.

Fig. 11 shows the search process of BO. By modeling the target function as a stochastic process (e.g., a Gaussian Process [36]), BO can compute the confidence interval of the target according to one or more observations. A confidence interval is an area that the curve of the target function is most likely (e.g., with 95% probability) passing through [10]. For example, in Fig. 11(a), the blue solid line is the target function and the black dashed line shows the prediction value. With some observations, BO computes the confidence interval that is marked with a cyan shadowed area. The confidence interval is updated (i.e., the posterior distribution in Bayes Theorem) after new observations are taken and the prediction improves as the area of the confidence interval decreases. BO can decide the next point to sample using a pre-defined acquisition function that also gets updated with the confidence interval. As shown in Fig. 11, the yellow star is chosen because the acquisition function indicates that it has the most potential gain. With more iterations, BO can obtain better results.

In addition, we admit that a better prior and acquisition function might be found given some dominion knowledge in specific applications. The confidence interval and the iteration number also may affect the result of the cost model. *Microscaler* can easily tune the above parameter according to specific applications. We discuss the influence of iteration in Section 4.1 and perform an acquisition function comparison between EI and GP-UCB in Section 4.2.

3.5.3 *The Step-by-Step Approach*

BO model cannot always find the optimal service scale with only a few iterations. But it can narrow down the search space efficiently. Based on the result obtained by BO, *Microscaler* leverages a step-by-step heuristic approach to achieve the optimal service scale exactly. In this approach *Microscaler* only increases or decreases one service instance each time since we have been near to the optimal result.

3.6 Auto-scale Action

This paper has built a Docker image for each microservice in the application and stored them in a private Docker Hub. With all necessary dependent libraries in one image, a microservice can be executed on any platform running a container engine quickly. *Microscaler* updates the microservice scale by adding containers or removing containers horizontally. Therefore, *Microscaler* can call the cloud Replica Controller API which are provided by most of cloud infrastructures such as Kubernetes and OpenShift, to keep the service replicas number consistent with the output of the auto-scale decision.

4 Experimental Evaluation

Experiment Settings. *Microscaler* is evaluated in a test distributed system. The test system contains 10 virtual machines (VMs) that host a microservice benchmark. Each VM has a 2-core 2.40GHz CPU, 6GB of memory and runs with Ubuntu16.04 OS. We guarantee all the VMs are in the same local area network so that we can reduce the network jitters.

All services are managed by an Istio-enabled Kubernetes platform. Kubernetes is an open-source system for automating deployment and management of containerized applications, which is one of the best platforms for deploying and running microservices. In addition, we do not consider the stability mechanisms implemented by the Kubernetes autoscaler.

Istio is an open-source service mesh that offers a complete solution to satisfy the diverse requirements (e.g., metrics collecting) of microservice applications. Envoy is deployed as a sidecar to the relevant service in Kubernetes. Since Envoy provides load balancing for microservices, we only need to consider the performance of the whole service rather than each fine-grained service instance.

Benchmark. Although the concept of microservice has been proposed and practiced for several years, there are very few benchmarks. According to our knowledge, Hipster, Sockshop [9], Trainticket [10], and the new proposed Socialnetwork [11] can be leveraged as the microservice benchmark. Hipster-Shop is a microservice demo that simulates the sale of hipster goods of an e-commerce website. Google uses this application to demonstrate Kubernetes, Istio, gRPC and similar cloud-native technologies nowadays. In addition, Hipster-Shop contains a load generator, which defines user behavior, to simulate the visits to the website with concurrent users. We can change the performance of Hipster-Shop by adjusting the number of microservices instances and change the request volume by adjusting the concurrent users in the load generator. In this paper, we choose Hipster-Shop because it is composed of 10 disparate microservices which makes it a perfect choice for our scenario to use and test *Microscaler* and address microservices challenges. While other benchmarks are either over simple (e.g., Sockshop) or lack of load generator (e.g., Trainticket). It needs to be emphasized that our approach can also work in other microservice benchmarks and real-world systems without differences. Moreover, we have pre-downloaded all docker images of Hipster-shop in each VM so that we can reduce the service start time and the pressure on network bandwidth.

Limited by the hardware resources of our testbed, we have a maximum of 15 instances for each service of Hipster-Shop. And we focus only on the violation of upper bound T_{max} because the process of scale in is very similar to the process of scale out. We set the upper bound of the front-end's service request latency T_{max} as $2s$.

4.1 Effectiveness Evaluation

Microscaler will trigger service dependency graph construction and scaling-needed service location after detecting an SLA violation. Fig. 12 illustrates a service dependency graph obtained by *Microscaler* in the last 5 minutes when running a load generator. The ellipse in the service dependency graph represents a microservice that has been visited in the last 5 minutes and the directed arrow indicates that the performance of arrow-tail microservice is dependent on the arrow-head microservice. Therefore, *Microscaler* can build a precise service dependency graph in real-time without any domain knowledge. Neither, we do not need to instrument each service. Compared with other statistics-based approaches

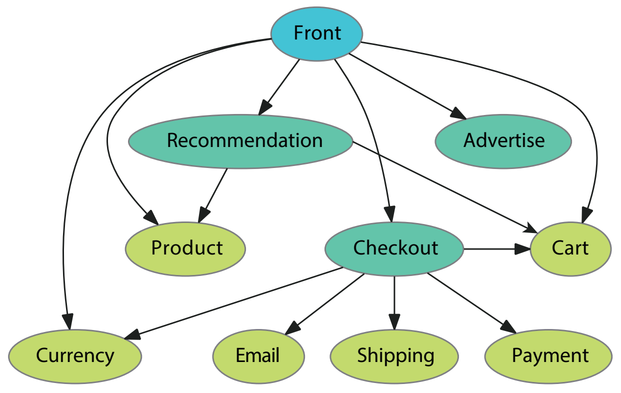


Fig. 12. Service dependency graph of Hipster Shop.

like [23], this approach can obtain more precise service dependencies.

Fig. 13 shows the precision of scaling-needed service determination when there is a service that is under-provision in Hipster-Shop. For simplifying the picture, we use SP to denote *service power*. For the approach combining Pearson product moment correlation or Spearman rank-order correlation coefficients with *service power*, it is shortened as Pearson+SP or Spearman+SP respectively. In addition, we compare the above approach with the traditional CPU and memory utilization threshold approach. Here, we set the CPU and memory utilization's higher threshold as 70%, and its lower threshold as 30% respectively. It means that in the CPU threshold approach if one service's CPU utilization deviates the range [70%, 30%], this approach considers this service as a scaling-needed service.

From Fig. 13, we observe that the precision of Pearson+SP and Spearman+SP is higher than 80% for different services. However, other methods are barely satisfactory in scaling-needed service determination. In addition, we observe that using *Microscaler* can improve scaling-needed determination precision significantly. We also find that Pearson and Spearman perform better than the CPU or memory threshold-based approach which only reflects the resource utilization. Thus, *Microscaler* selects Pearson+SP or Spearman+SP as the method to locate scaling-needed services. We run each scaling-needed determination method for 50 times to reduce the bias introduced by system noises and uncertainties in this experiment.

After finding the scaling-needed services, *Microscaler* will scale them serially. Fig. 14 shows the BO searching process in different services and different iterations. First, we observe that BO can find an optimal or sub-optimal replica number just with a few iterations. In addition, the number of iterations affects the result of the replica number. Each iteration needs to wait for some time to start containers and collect the latency metric, it means that the searching process can cost considerable time. So, selecting an appropriate iteration number is important not only in outputting a good result but also in reducing searching time. From Fig. 14, we can observe that the result of 3 iterations has a greater differential with 4 iterations or 5 iterations. But the result of 4 iterations has a small difference with 5 iterations. It means that 4 could be a suitable iteration number for *Microscaler* because BO can output an optimal or sub-optimal result in 4

9. Sockshop [http://www.github.com/microservices-demo/microservices-demo]
10. Trainticket [http://www.github.com/FudanSELab/train-ticket]
11. Socialnetwork [http://www.github.com/YashchenkoN/social-network]

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2985352, IEEE Transactions on Cloud Computing

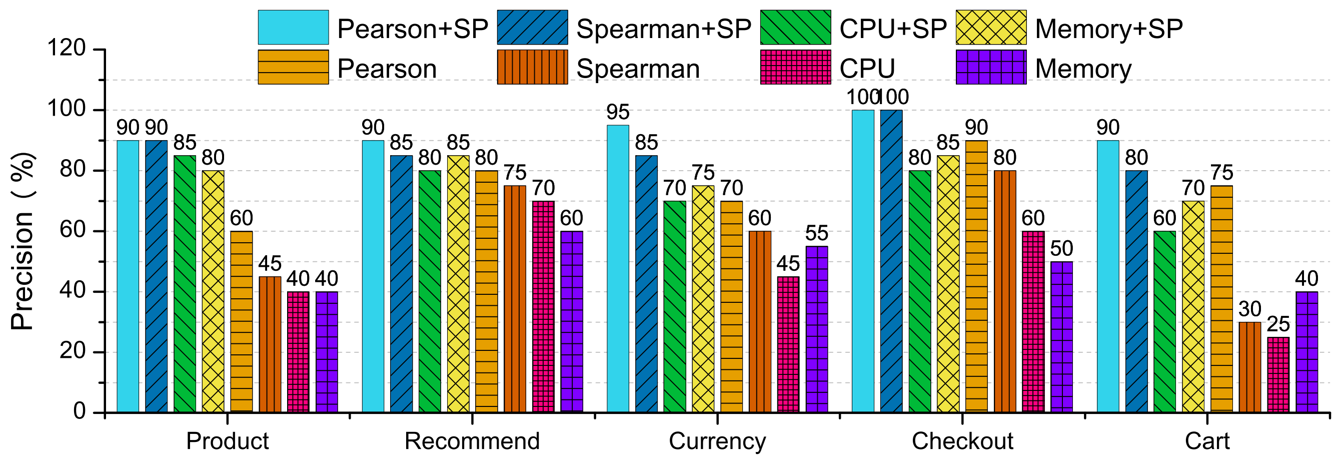


Fig. 13. The precision of scaling-needed service determination procedure.

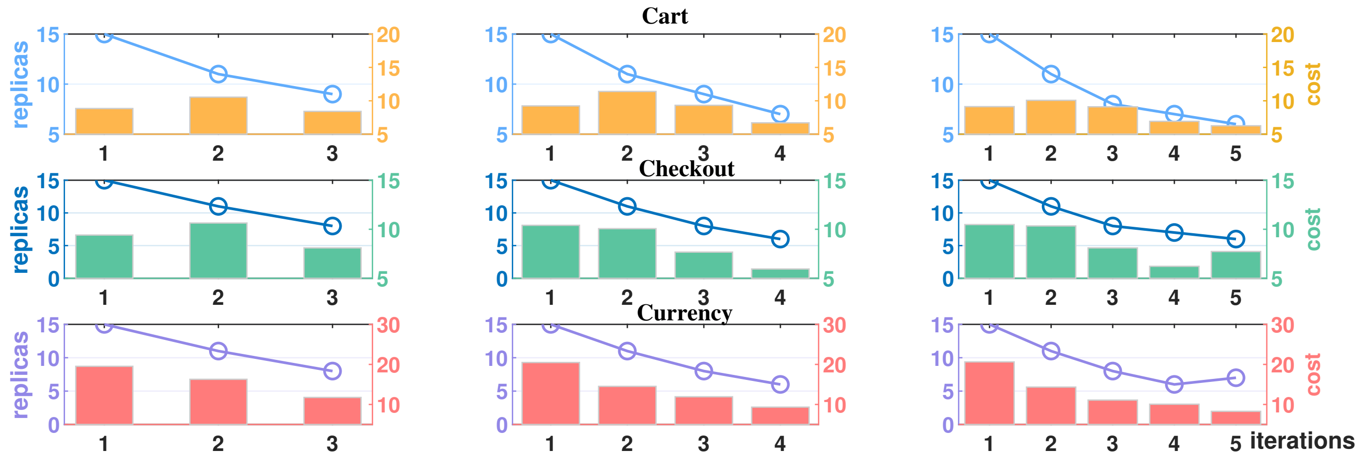


Fig. 14. The autoscaling results for different services obtained by BO.

iterations. Certainly, this parameter could be tuned in other microservice systems by multiple tests.

4.2 Comparisons

To compare the search process between *Microscaler* and other mainstream autoscalers, we adjust the Hipster-shop load generator's concurrent users from 20 to 500 to mimic the workload surge. After that, *Microscaler* detects the latency violation promptly and then finds that the scaling-needed service is the *Product* service. The rule-based autoscalers are common in industry [6] like Amazon Web Service Autoscaling and Kubernetes Horizontal Pod Autoscaler. In this paper, we utilize the Amazon Emulated autoscaler that is a copied version of the AWS autoscaling policy to represent rule-based autoscalers. Here, we set the CPU utilization's higher threshold as 70%. We deliberately use a modifying step of one and two replicas to show the difference between each autoscaler. It means that if one service's CPU utilization exceeds 70% the Amazon (1 replica) Autoscaler adjusts one instance each time and the Amazon (2 replicas) Autoscaler adjusts two instances each time for this service. And the fuzzy autoscaler emulates the model proposed by [9]. For the white-box auto-scaler [37], it will calculate the model between workload and service scale explicitly in advance. Therefore, the white-box autoscaler can reach the optimal service scale by calculating models directly when the workload changes.

Fig. 15 shows the process to reach the desired service scale in some autoscalers which adopt different scaling policies. The left part of Fig. 15 shows how these auto-scalers modify their actions to reach the target replicas number of *Product* service. There is no doubt that the Amazon emulated auto-scaler which updates one replica each time will accurately find the optimal minimal number that meets SLA. It is also shown that it needs more steps to find the optimal replicas than the other three methods. Hence, in Fig. 15, the optimal replicas is 12, the result of BO auto-scaler is 13. This means that BO auto-scaler may cause the unnecessary cost to serve the same workload because they may find the sub-optimal replicas. *Microscaler* searches continuously until finding the optimal service scale. Therefore, it also reaches 12 service instances. With regard to the time of finding the optimal service scale, the White-box auto-scaler can reach its ideal scale in only 1 step, and *Microscaler* need 5 steps. Amazon auto-scaler (1 replicas case), Amazon auto-scaler (2 replicas case) and fuzzy auto-scaler need 10 steps, 5 steps, 6 steps respectively. In short, *Microscaler* can reach the optimal service scale with very few steps.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.2985352, IEEE Transactions on Cloud Computing

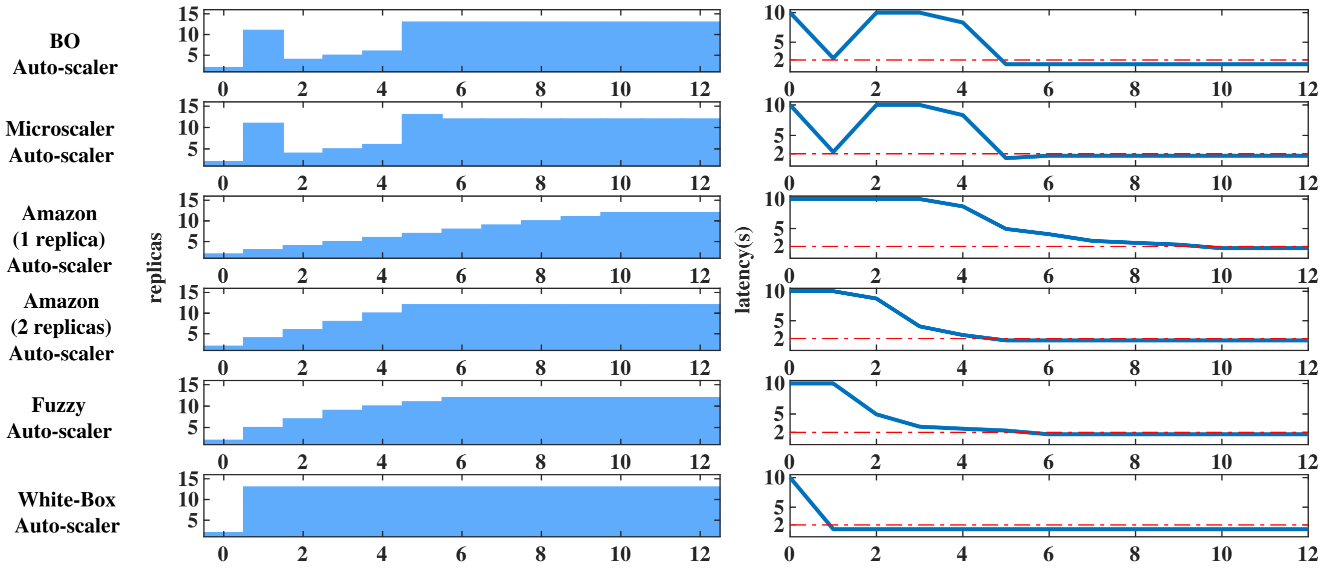


Fig. 15. Changes of numbers of replicas and the request latency obtained by different autoscalers.

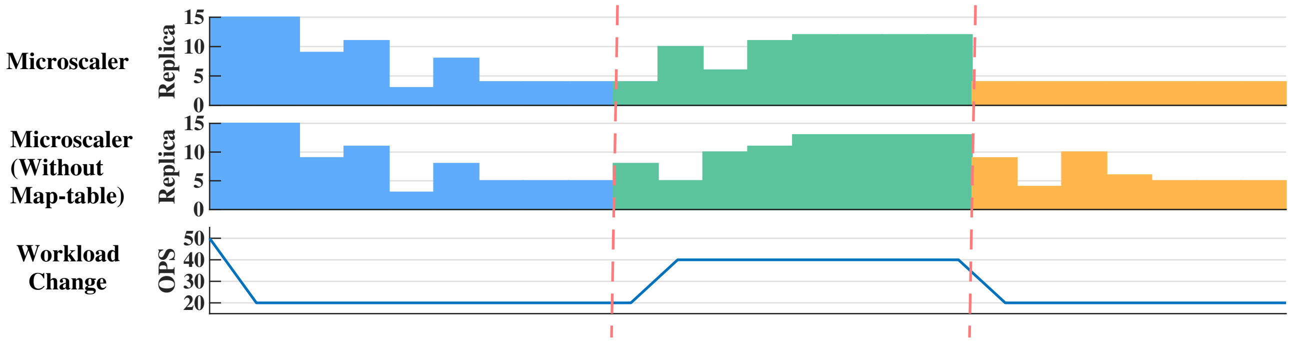


Fig. 16. Performance of Microscaler with and without map-table towards workload change.

To observe the effect of map-table, we divide the workload into three phases. In *Phase1* and the *Phase3*, the workload to application are both 20 ops. While in the *Phase2*, the workload is 40 ops. The graphs of Fig. 16 shows differences in the searching process between *Microscaler* with map-table and without map-table against workload changes. The bottom subfigure in Fig. 16 describes the changes in workload. In *Phase3*, *Microscaler* acquires the scale number from the map-table directly rather than searches the optimal scale from scratch.

To compare the performance of autoscalers in searching the optimal service scale. We set the initial number of users via Hipster-shop load generator to 20 and then we randomly select an integer in $[50, 500]$ as the current users to mimic the workload surge. We run each kind of auto-scaler for 50 times in this experiment to mitigate the bias. From Fig. 17, we can observe that *Microscaler* and Amazon auto-scaler (1 replica case) can reach the optimal service scale every time. BO auto-scaler and fuzzy auto-scaler can provide an acceptable service performance, but the results are not always optimal. Furthermore, BO auto-scaler which uses the EI acquisition function performs better than using the GP-UCB acquisition function. As for the white-box auto-scaler, it can reach the optimal service scale if the current workload had been measured. However, for the workload which is not measured, it may provide insufficient replicas (e.g., optimal-1). In addition, this phenomenon is more obvious at cold-start. So the white-box auto-scaler needs many tests to get a consistent result. And when the service updates, the white-box auto-scaler needs to update models accordingly.

TABLE 1
The Overhead of Microscaler

System Module	Cost
Data Collection	$5\% \pm 1\%$ Single CPU Utilization
SLA Violation Detection	about 0.2 second
Scaling-needed Determination	about 1 seconds
Auto-scale Decision(Unknown Workload)	about 2 minutes
Auto-scale Decision(Known Workload)	about 0.1 seconds

4.3 Discussion

Overhead. TABLE 1 shows the overhead of *Microscaler*. The data collection module takes about 5% CPU utilization including collecting service invocation and service latency with the help of service mesh. In auto-scale decision module, *Microscaler* consumes about 2 minutes facing the work-

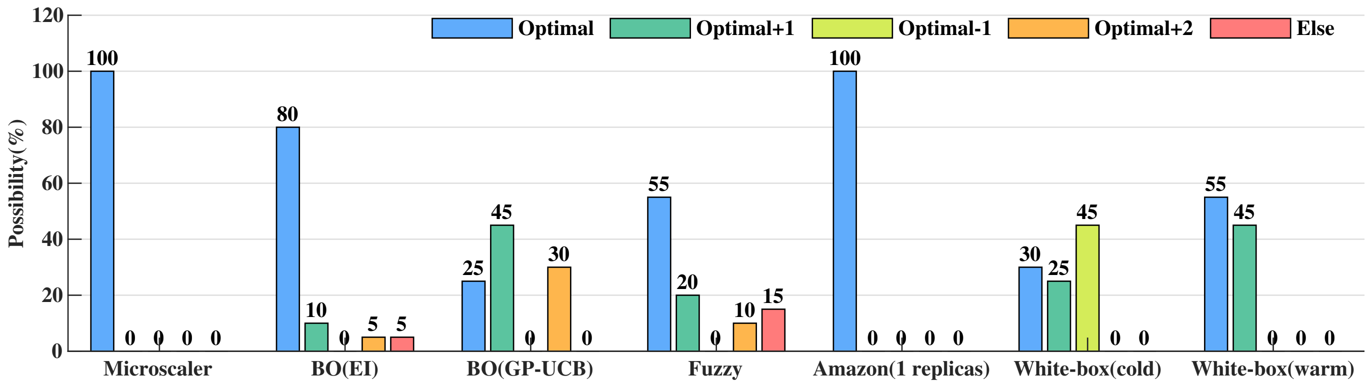


Fig. 17. The probability of reaching the optimal service scale by different autoscalers.

load which has not been stored in the map-table (i.e., unknown workload) because creating containers needs some time after changing service's replica number. Otherwise, the auto-scale decision module can output the optimal result rapidly based on the map-table. From the aforementioned conclusion, state-of-the-art methods need more time than *Microscaler*. Overall, *Microscaler* is a light-weight and flexible autoscaler which can be deployed in a large-scale microservice system readily.

Limitation. Firstly, nowadays almost all variants of service mesh are only suitable for the container-based cloud. They are lack of support for VM-based cloud. Therefore, *Microscaler* is not appropriate for VM autoscaling. Secondly, *service power* performs well when facing a crescent workload, but it may perform worse when the workload experienced a growth spurt. In addition, since BO is a method for optimizing black-box functions, it may reach the sub-optimal but not the optimal service scale which will cause some unnecessary cost. The last but not the least, *Microscaler* reactively scales out the application only when an SLA violation occurs. Hence, *Microscaler* may take actions later than the proactive scaling approaches.

5 Related Work

The autoscaling problem can be defined as how to autonomously and dynamically scale out or sale in to deal with fluctuant workloads without human intervention so that the resource cost is minimized and application's SLAs are satisfied. The accurate resource estimation helps the autoscaler to converge to the optimal resource provisioning in short order. Recently, extensive approaches have been proposed to conduct auto-scaling based on resource estimation models.

5.1 Rule-based Approach

The core of rule-based approaches is a set of rules which consist of the triggering conditions and the corresponding actions, such as "If the memory utilization exceeds 80%, adding one instance". With this approach, a mapping must exist denoting the representative action that must be executed once a specific threshold has been breached. As the simplest method of auto-scaling, it commonly serves as a baseline for comparison and is applied as the basic scaling framework for the researches that focus on other aspects of auto-scaling, such as the work done by Dawound et al. [38], which aims to compare vertical scaling with horizontal scaling, and the work done by Han et al. [39], which considers all resources level (CPUs, memory, I/O, etc) methods. Although a ruled-based autoscaler is simple to implement, it needs a comprehensive and detailed understanding of the application and expert knowledge to determine thresholds and the corresponding actions. Frankly speaking, simple rule-based approaches involve no accurate resource estimation but rely heavily on empirical estimation.

5.2 Fuzzy-based Approach

Fuzzy-based approaches generally pre-define a set of "If-Else" rules based on empirical resource estimation to make provisioning decisions. The major advantage of fuzzy inference compared to rule-based approaches is that it allows users to use linguistic terms like "high, medium, low" instead of accurate numbers to define the conditions and actions [6]. Fuzzy-based approaches first fuzzy the inputs by defined membership functions. Then the fuzzified inputs are used to trigger the action parts in the ruleset concurrently. Finally the results of rules are combined and defuzzied as the output for scaling operation [40] [41]. It is the same as the rule-based reasoning that manually designing the ruleset is troublesome and hard to handle the dynamical production environment. Persico et al. [42] proposed a feedback control scheme that leverages fuzzy logic to self-adjust its parameters in order to cope with unpredictable and highly time-varying public-cloud operating conditions. Compared with the rule-based approaches and fuzzy-based approaches, *Microscaler* does not need to set the inflexible rules and can converge to the optimal scale flexibly without expert knowledge.

5.3 Application Profiling-based Approach

We define application profiling as a process which utilizes synthetic or recorded real workload to test the specific application's saturation point of resources. Offline application profiling can accurately acquire the complete knowledge of how many resources are just enough under different levels of workload intensity. Gandhi et al. [43], Fernandez et al. [44], Qu et al. [45] profiled each type of resources offline to measure their capacity. To overcome the drawback of offline application profiling that the profiling requires to be reconstructed after the application has been updated. Agile [46] is online profiling to derive a resource estimation model for each application tier. Jiang et al. [47] focused on the quick online profiling for multi-tier applications by calculating the correlation of resource requirements that different tiers pose on the same type of VM and the profile of a particular tier on that type of VM. There is no doubt that the processes of testing the saturation point of resources need to consume a lot of resources, while *Microscaler* can reach the optimal point with only a few tests.

5.4 Machine Learning Based Approach

The increasing complexity of cloud applications has rendered the building autoscalers a difficult task for human experts. Therefore, recent studies have exploited the advances of machine learning to create more reliable autoscalers. Machine learning approaches are applied to dynamically build the autonomous controllers which are capable of adding and removing instances on the basis of a variable workload model. In this way, operators can utilize autoscalers without customized settings and preparations. Online machine-learning algorithms are often implemented as feedback controllers to realize self-adaptive evolution and thus are more robust to changes during production. Machine learning approaches suffer from a major drawback that they take time for them to converge to a stable model and thus cause the auto-scaler to perform poorly during the active learning period.

For regression-based autoscalers, they model functions based on observed data(e.g., system utilization, application performance and the workload), and then use them to make predictions. Zhang et al. [48], Collazo-Mojica et al. [49] and Yang et al. [50] developed a cost-aware autoscaling model for workload prediction in which the linear regression model is used. Roy et al. [51] proposed a predictive-aware model for the workload prediction by a second-order autoregressive moving-average (ARMA) model. Islam et al. [52] utilized a combination of neural networks and linear regression to predict future resource demands. For the above regression-based autoscalers, they rely on conducting a statistical hypothesis test of the residual distribution in two-time frames with probably different sizes. If the test result is statistically significant, the model needs to be retrained. If workload patterns change frequently, the cost and time of retraining model cannot be ignored.

For autoscalers based on reinforcement learning, they focus on learning how to react adaptively to generate tables specifying the best provision or deprovision action under each state. Barrett et al. [53] presented a parallel Q-learning method that consists of multiple RL agents that used to reduce the time taken to determine optimal autoscaling action. Bahrpeyma et al. [54] developed an adaptive control approach that is based on continuous reinforcement learning and provides dynamic resource provisioning. Liu et al. [55] proposed a reinforcement learning-based aggressive vitalized resource management system to overcome the rapidly increasing workloads.

The machine learning approaches usually need lots of training samples that extract from historical data. However, microservice applications have a rapid update schedule. New features may be continuously integrated and deployed into each of microservice overtime, which causes it is hard to obtain sufficient and reliable historical data. Unlike machine learning approaches, *Microscaler* does not need the historical data, but can adapt to workload changes in realtime.

6 Conclusion and Future Work

In this paper, we design and implement a novel system named *Microscaler* to help application providers pinpoint the scaling-needed services and scale them to meet the SLA requirement with an optimal cost for microservice systems in the service-mesh-enabled environments. *Microscaler* determines under-provisioning or over-provisioning service instances along the service dependency graph, and a novel criterion i.e., *service power* is proposed to confirm the scaling-needed services. By combining an online learning approach and a step-by-step heuristic approach, *Microscaler* can precisely achieve the optimal service scale satisfying the SLA requirements. The experimental evaluations in a microservice benchmark show that *Microscaler* achieves an average 93% precision in scaling-needed service determination and converges to the optimal service scale faster than several state-of-the-art methods. Moreover, *Microscaler* is lightweight and flexible enough to work in a large-scale microservice system. In the future, we will try to combine the predictive methods with our system to further improve the accuracy and convergence speed of autoscaling. We also intend to reduce the start time of containers which will help to accelerate the process of autoscaling.

7 Acknowledgments

The work described in this paper was supported by the National Key R&D Program of China (2019YFB1804002), the National Science Foundation of China (61802448,61722214,U1811462) and the National Science Foundation of Guangdong (2019A1515012229). The corresponding author is Pengfei Chen.

References

[1] G. Yu, P. Chen, and Z. Zheng, "Microscaler: Automatic scaling for microservices with an online learning approach," in *2019 IEEE International Conference on Web Services*, 2019, pp. 68–75.

[2] A. Zhou, S. Wang, Z. Zheng, C. Hsu, M. R. Lyu, and F. Yang, "On cloud service reliability enhancement with optimal resource usage," *IEEE Trans. Cloud Computing*, vol. 4, no. 4, pp. 452–466, 2016.

[3] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Proc.16th International Conference Service-Oriented Computing*, 2018, pp. 3–20.

[4] S. Newman, *Building Microservices - Designing Fine-grained Systems*. O'Reilly, 2015. [Online]. Available: http://www.worldcat.org/oclc/904463848

[5] T. Chen, R. Bahsoon, and X. Yao, "A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 61:1–61:40, 2018.

[6] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 73:1–73:33, 2018.

[7] J. Zhou, Z. Chen, J. Wang, Z. Zheng, and M. R. Lyu, "Trace bench: An open data set for trace-oriented monitoring," in *IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom 2014*, 2014, pp. 519–526.

[8] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proc.26th International Conference on World Wide Web*, 2017, pp. 469–478.

[9] B. Liu, R. Buyya, and A. N. Toosi, "A fuzzy-based auto-scaler for web applications in cloud computing environments," in *Proc. 16th International Conference Service-Oriented Computing*, 2018, pp. 797–811.

[10] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *Proc. 14th USENIX Symposium on Networked Systems Design and Implementation*, 2017, pp. 469–482.

[11] D. Luong, H. Thieu, A. Outtagarts, and Y. Ghamri-Doudane, "Predictive autoscaling orchestration for cloud-native telecom microservices," in *IEEE 5G World Forum*, 2018, pp. 153–158.

[12] A. U. Gias, G. Casale, and M. Woodside, "ATOM: model-driven autoscaling for microservices," in *39th IEEE International Conference on Distributed Computing Systems,*, 2019, pp. 1994–2004.

[13] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Proc. 10th Computing Colombian Conference*, 2015, pp. 583–590.

[14] I. Prachitmutita, W. Aittinonmongkol, N. Pojjanasuksakul, M. Supattatham, and P. Padungweang, "Auto-scaling microservices on iaas under SLA with cost-effective framework," in *Proc. 10th International Conference on Advanced Computational Intelligence*, 2018, pp. 583–588.

[15] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin, and L. Safina, "Microservices: How to make your application scale," in *Proc. 11th International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, 2017, pp. 95–104.

[16] "Linkerd: Twitter-style operability for microservices," https://linkerd.io/2016/02/18/linkerd-twitter-style-operability-for-microservices/, [Online; accessed 2019].

[17] "Service mesh data plane vs. control plane," https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc, [Online; accessed 2019].

[18] "What is istio," https://istio.io/docs/concepts/what-is-istio/, [Online; accessed 2019].

[19] J. Walter, D. Okanovic, and S. Kounev, "Mapping of service level objectives to performance queries," in *Proc. 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 197–202.

[20] J. Xu, W. Yuan, P. Chen, W. Ping, J. Xu, W. Yuan, P. Chen, and W. Ping, "Lightweight and adaptive service api performance monitoring in highly dynamic cloud environment," in *IEEE International Conference on Services Computing*, 2017, pp. 35–43.

[21] S. Roy, A. C. König, I. Dvorkin, and M. Kumar, "Perfaugur: Robust diagnostics for performance anomalies in cloud services," in *IEEE International Conference on Data Engineering (ICDE)*, 2015, pp. 1167–1178.

[22] P. Bahl, R. Chandr, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *ACM SIGCOMM Computer Communication Review*, vol. 37, 2007, pp. 13–24.

[23] P. Chen, Y. Qi, P. Zheng, and D. Hou, "Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *Proc. IEEE Conference on Computer Communications*, 2014, pp. 1887–1895.

[24] J. Weng, J. H. Wang, J. Yang, and Y. Yang, "Root cause analysis of anomalies of multitier services in public clouds," *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, pp. 1646–1659, 2018.

[25] B. Sang, J. Zhan, Z. Zhang, L. Wang, D. Xu, Y. Huang, and D. Meng, "Precise, scalable and online request tracing for multitier services of black boxes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 6, pp. 1159–1167, 2012.

[26] I. Olkin, J. W. Pratt *et al.*, "Unbiased estimation of certain correlation coefficients," *The Annals of Mathematical Statistics*, vol. 29, no. 1, pp. 201–211, 1958.

[27] N. S. Chok, "Pearson's versus spearman's and kendall's correlation coefficients for continuous data," Ph.D. dissertation, University of Pittsburgh, 2010.

[28] D. Shen, Z. Lu *et al.*, "Computation of correlation coefficient and its confidence interval in sas," *SUGI: Paper*, pp. 170–31, 2006.

[29] E. Brochu, V. M. Cora, and N. de Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *CoRR*, vol. abs/1012.2599, 2010.

[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, and e. a. Marc G. Bellemare, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[31] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proc. Advances in Neural Information Processing Systems*, 2012, pp. 2951–2959.

[32] H. J. Kushner, "A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise," *Journal of Basic Engineering*, vol. 86, no. 1, pp. 97–106, 1964.

[33] N. Srinivas, A. Krause, S. Kakade, and M. W. Seeger, "Gaussian process optimization in the bandit setting: No regret and experimental design," in *Proc. 27th International Conference on Machine Learning*, 2010, pp. 1015–1022.

[34] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *J. Global Optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[35] J. R. Gardner, M. J. Kusner, Z. E. Xu, K. Q. Weinberger, and J. P. Cunningham, "Bayesian optimization with inequality constraints," in *Proceedings of the 31th International Conference on Machine Learning*, 2014, pp. 937–945.

[36] D. J. MacKay, "Introduction to gaussian processes," *NATO ASI Series F Computer and Systems Sciences*, vol. 168, pp. 133–166, 1998.

[37] U. Sharma, P. J. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *Proc. International Conference on Distributed Computing Systems*, 2011, pp. 559–570.

[38] W. Dawoud, I. Takouna, and C. Meinel, "Elastic virtual machine for fine-grained cloud resource provisioning," in *Global Trends in Computing and Communication Systems*, 2012, pp. 11–25.

[39] R. Han, L. Guo, M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012, pp. 644–651.

[40] P. Lama and X. Zhou, "Efficient server provisioning with end-to-end delay guarantee on multi-tier clusters," in *17th International Workshop on Quality of Service*, 2009, pp. 1–9.

[41] S. Frey, C. Lüthje, C. Reich, and N. L. Clarke, "Cloud qos scaling by fuzzy logic," in *Proc. IEEE International Conference on Cloud Engineering*, 2014, pp. 343–348.

[42] V. Persico, D. Grimaldi, A. Pescapè, A. Salvi, and S. Santini, "A fuzzy approach based on heterogeneous metrics for scaling out public clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8, pp. 2117–2130, 2017.

[43] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Trans. Comput. Syst.*, vol. 30, no. 4, pp. 14:1–14:26, 2012.

[44] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *2014 IEEE International Conference on Cloud Engineering*, 2014, pp. 195–204.

[45] C. Qu, R. N. Calheiros, and R. Buyya, "A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances," *Journal Network and Computer Applications*, vol. 65, pp. 167–180, 2016.

[46] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "AGILE: elastic distributed resource scaling for infrastructure-as-a-service," in *10th International Conference on Autonomic Computing*, 2013, pp. 69–82.

[47] J. Dejun, G. Pierre, and C.-H. Chi, "Resource provisioning of web applications in heterogeneous clouds," in *Proc. 2nd USENIX Conference on Web Application Development*, 2011, pp. 5–5.

[48] Y. Zhang, G. Huang, X. Liu, and H. Mei, "Integrating resource consumption and allocation for infrastructure resources on-demand," in *IEEE International Conference on Cloud Computing*, 2010, pp. 75–82.

[49] X. J. Collazo-Mojica, S. M. Sadjadi, J. Ejarque, and R. M. Badia, "Cloud application resource mapping and scaling based on monitoring of qos constraints," in *Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering*, 2012, pp. 88–93.

[50] J. Yang, C. Liu, Y. Shang, and B. Cheng, "A cost-aware auto-scaling approach using the workload prediction in service clouds," *Information Systems Frontiers*, vol. 16, no. 1, pp. 7–18, 2014.

[51] N. Roy, A. Dubey, and A. S. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *IEEE International Conference on Cloud Computing*, 2011, pp. 500–507.

[52] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Comp. Syst.*, vol. 28, no. 1, pp. 155–162, 2012.

[53] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[54] F. Bahrpeyma, H. Haghighi, and A. Zakerolhosseini, "An adaptive RL based approach for dynamic resource provisioning in cloud virtualized data centers," *Computing*, vol. 97, no. 12, pp. 1209–1234, 2015.

[55] J. Liu, Y. Zhang, Y. Zhou, D. Zhang, and H. Liu, "Aggressive resource provisioning for ensuring qos in virtualized environments," *IEEE Trans. Cloud Computing*, vol. 3, no. 2, pp. 119–131, 2015.

Guangba Yu received his BE degree from Guangzhou University of Chinese Medicine, China, in 2018. He is now an postgraduate at School of Data and Computer Science with Sun Yat-sen University, China. His current research areas include distributed system, cloud computing, and AI driven operations.

Pengfei Chen is currently an associated professor in School of Data and Computer Science of Sun Yat-sen University. Meanwhile, he is a Ph.D. advisor. Dr. Chen graduated from the department of computer science of Xi'an Jiaotong University with a Ph.D. degree in 2016. He was selected as a member of "Rising Star" program of Microsoft Research Asia during Jul, 2012 – Nov, 2012. After graduation, Dr. Chen worked as a Research Scientist in IBM Research China during Jun, 2016-Jan, 2018. He visited IBM T.J. Watson from Feb, 2017 to Apr, 2017. Now, he is interested in distributed systems, AIOps, cloud computing, Microservice and BlockChain. Especially, he has strong skills in cloud computing. Until now, Dr. Chen has published more than 30 papers in some international conferences including IEEE INFOCOM, WWW, IEEE ICSOC, IEEE ICWS, IEEE BigData and journals including IEEE TDSC, IEEE TR, IEEE TSC, IEEE TETC, IEEE TCC. He serves as of program committee member of multiple conferences and reviewers of some internal journals such as IEEE Transactions on Cybernetics, Information Science, and Neurocomputing.

Zibin Zheng received his Ph.D. degree from Chinese University of Hong Kong, in 2011. He is currently a Professor at School of Data and Computer Science with Sun Yat-sen University, China. He serves as Chairman of the Software Engineering Department. He published over 120 international journal and conference papers, including 3 ESI highlycited papers. According to Google Scholar, his papers have more than 7000 citations, with an H-index of 42. His research interests include blockchain, services computing, software engineering, and financial big data. He was a recipient of several awards, including the Top 50 Influential Papers in Blockchain of 2018, the ACM SIGSOFT Distinguished Paper Award at ICSE2010, the Best Student Paper Award at ICWS2010. He served as BlockSys'19 and CollaborateCom'16 General Co-Chair, SC2'19, ICIOT'18 and IoV'14 PC Co-Chair.