2019 IEEE International Conference on Web Services (ICWS)

Microscaler: Automatic Scaling for Microservices with an Online Learning Approach

Guangba Yu[1], Pengfei Chen*[1] and Zibin Zheng[1 2]

[1]*School of Data and Computer Science, Sun Yat-sen University*
[2]*National Engineering Research Center of Digital Life, Sun Yat-sen University*
Guangzhou 510006, China
Email: yubg5@mail2.sysu.edu.cn, {chenpf7, zhzibin}@mail.sysu.edu.cn

***Abstract*—Recently, the microservice becomes a popular architecture to construct cloud native systems due to its agility. In cloud native systems, autoscaling is a core enabling technique to adapt to workload changes by scaling out/in. However, it becomes a challenging problem in a microservice system, since such a system usually comprises a large number of different micro services with complex interactions. When bursty and unpredictable workloads arrive, it is difficult to pinpoint the scaling-needed services which need to scale and evaluate how much resource 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 micro-service systems. *Microscaler* collects the quality of service metrics (QoS) with the help of the service mesh enabled infrastructure. Then, it determines the under-provisioning or over-provisioning services with a novel criterion named *service power*. By combining an online learning approach and a step-by-step heuristic approach, *Microscaler* could achieve the optimal service scale satisfying the SLA requirements. The experimental evaluations in a micro-service benchmark show that *Microscaler* converges to the optimal service scale faster than several state-of-the-art methods.**

***Index Terms*—Auto-scaling, Micro service, Service Mesh, Bayesian Optimization, Cloud computing**

I. Introduction

Driven by the promising features of cloud computing such as pay-as-you-go, elasticity, and on-demand, many cloud native systems are emerging. In order to take advantage of cloud resources more efficiently, those enterprises build or reconstruct their application architectures from monolithic to microservice. With the micro-service architecture, an application is decoupled into many loosely distributed fine-grained services [1]. Each service has simple and independent functions following the SRP (Single Responsibility Principle) [2]. The micro-service architecture allows applications scale out/in for partial services in a fine granularity (e.g., container), which reduces the scaling cost compared with the conventional virtual machine scaling.

As the workload of one application in the Cloud is unpredictable, it is crucial to automatically scale out/in with resource as few as possible on condition that meeting the SLA requirements, in the face of workload changes . The under-provisioning should be avoided during the scaling phase. On the other side, the over-provisioning leads to resource wasting and extra cost [3], the application provider needs a cost-optimal scaling without SLA violations.

Cloud computing provides a flexible resource allocation mechanism. But it is up to the application owner to leverage the flexible infrastructure [4]. However, an automatic scaling approach in micro-service environments is notoriously difficult due to the following challenges:

- **Service metric collection**. Most mainstream cloud platforms today do not provide sufficient means to monitor application in a fine granularity [5]. Therefore, it is difficult to obtain service-level performance metrics in the wild. Although the application-level instrumentation can achieve that, application developers must be familiar how to expose such performance metrics.
- **Scaling-needed service determination**. A large number of services co-exit in a micro-service system and the interactions among them are complex. Hence, a performance anomaly may result in multiple anomalies in many services simultaneously. Therefore, it it non-trivial to determine the scaling-needed services when a performance anomaly occurs.
- **Performance and cost optimization**. An auto-scaler 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 [6]. But there is not an explicit regulation model between resource of each services and the request volume. The auto-scaler needs to determine how many resources are required to scale effectively and efficiently.

Extensive methods have been proposed to solve auto-scaling problems for virtual machines in cloud environment [7], [6]. However, very few methods can auto-scale quantitatively in the dynamic micro-service environment. Moreover, most of 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 three procedures, namely service metric collection, scaling-needed service determination and auto-scale decision. *Microscaler* collects the service metrics exposed by the service mesh driven infrastructure continuously. In this paper, we first find all abnormal services.

978-1-7281-2717-0/19/$31.00 ©2019 IEEE
DOI 10.1109/ICWS.2019.00023

Then we trigger the scaling process to meet SLA requirements. *Microscaler* leverages the *service power* to determine the scaling-needed services and decides the service size by combining a Bayesian Optimization based model and a step-by-step heuristic model. In addition, *Microscaler* works in a real-time mode to adapt to the dynamic workload. Moreover, we validate *Microscaler's* effectiveness and efficiency in a micro-service benchmark, namely Hipster-shop [8], managed by Istio [9] (i.e., a popular service mesh) enabled Kubernetes [10]. The experimental evaluations show that *Microscaler* 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 problems of autoscaling in micro-service systems.
- We propose a novel criterion named *service power* to determine the scaling-needed services in micro-service systems, which reduces unnecessary scaling processes.
- We combine an online learning approach and a step-by-step heuristic approach to build a model for auto-scaling. The model can obtain the optimal service scale with only a few iterations.
- We design and implement a prototype, namely *Microscaler*, to evaluate the proposed autoscaling approach in a micro-service benchmarking system.

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 results. Finally, we discuss the related work in Section 5 and conclude this paper in Section 6.

II. Background and Motivation

A. Service Mesh

Recently, the microservice architecture becomes a surge and a necessary element for cloud native systems. In such a system, a single application might consist of hundreds of micro services. Each micro service might have thousands of instances. And each instance might be in a constantly-changing state as it is dynamically scheduled by an cloud orchestrator like Kubernetes. To make the development, deployment, and orchestration of micro-service systems easy, the concept of service mesh is proposed recently. Service mesh is a dedicated infrastructure layer for handling service-to-service communication. It is responsible for the reliable distribution of requests through the complex topology of micro services. In practice, a service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code without the awareness of applications.

Istio [9] is a completely open source service mesh that sits transparently on existing distributed applications and offers a complete solution to satisfy the diverse requirements (e.g., metrics collecting, request tracing) of micro-service applications [9]. Istio provides a high-performance proxy to mediate all inbound and outbound traffic for all services. The service metrics provided by Istio will help us determine when to scale and how to scale.

B. Motivation

Although some cloud platforms like Amazon EC2, Kubernetes have been equipped with an autoscaling mechanism, they are mainly triggered by the exceeding of resource limits (e.g., higher than 90% CPU utilization). However, when a service has a higher resource utilization, it does not mean that the service needs to scale. It may be caused by a software bug such as an infinite loop. Moreover, the existing tools do not determine which services need to scale before scaling. Therefore, they do not work well in micro-service systems. With the ever growing scale and complexity of modern micro-service applications, it is difficult to locate the micro service which is under-provisioning or over-provisioning. Making effective scale decision to utilize resources exactly is challenging as well. Following this motivation, we propose *Microscaler* to conduct a cost-optimal autoscaling.

III. System Design

A. System Overview

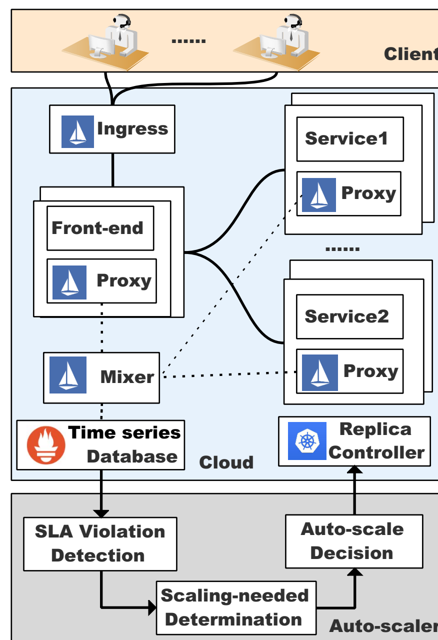

Fig. 1. System architecture of *Microscaler*

Fig. 1. shows the system architecture of *Microscaler*. Micro-service applications are deployed in a service mesh enabled cloud platform. A service mesh infrastructure mainly comprises a data plane and a control plane. At the data plane, a network proxy is deployed as a sidecar to each relevant service instance (i.e., container). The issued request will go through the ingress before visiting the front-end of application. At the control plane, a telemetry component (e.g., Mixer [9]) collects all network communication from proxy and stores them in a time series database (e.g., Prometheus). *Microscaler*

continually monitors the performance metrics of the front-end service within a sliding window. The scaling-needed service determination process will be triggered when an SLA violation is detected. After that, *Microscaler* will scale out/in services according to the result generated by the auto-scale decision module. Finally, *Microscaler* issues a command to Cloud Controller to perform scaling.

B. Service Metrics Collection

In the service metrics collection module, *Microscaler* mainly collects Quality of Service (QoS) metrics (i.e., service latency) of each service with the help of service mesh. Microservice applications run in a service mesh-enabled cloud environment, with proxy sidecars injected along side each service. These sidecars mediate and control all network communication between micro services along with a control plane component like Mixer in Istio. Moreover, we can integrate a customized metric collection template [9] to get metrics at different levels such as container level, pod level, and service level. A time series database is leveraged to store the service metrics exposed by the service mesh.

QoS Metrics. SLA provides information on the scope, the quality and the responsibilities of each micro service and its provider [11]. They involve QoS metrics 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 QoS metrics from service mesh for monitoring services and determining scaling-needed services.

C. SLA violation Detection

Micro services are monitored periodically (e.g., per 10 seconds) to keep consistent with the SLA in the face of fluctuated request volumes. The SLA of a service is specified by its providers when the service is deployed. A performance anomaly is an event or a collection of events that cause SLA violations. In this paper, *Microscaler* selects the service request latency to denote the application performance rather than the resource utilization (e.g., CPU) or the request volume as this metric directly expresses end users' experience.

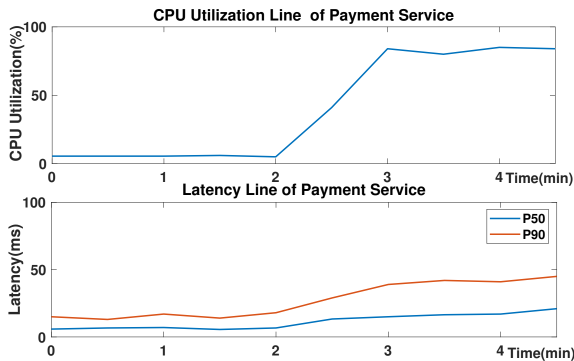

Fig. 2. An example of high CPU utilization but low service latency

In microservice applications, the correlation between resource utilization and service request latency might be not so strong. As Fig. 2. shows, 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. As for the request workload, if taking the request workload as a trigger, it is unsure to conduct autoscaling as the requests will be dispatched to different down-streaming services. Considering that a small request volume congests on the same micro service, even though the workload at that time do not trigger scaling, the request latency may exceed the SLA. On the contrary, if a huge workload volume could be distributed to different service instances in an appropriate manner, 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 to trigger unnecessary scaling [6].

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 service request latency. If the request latency measurements are higher than T_{max} or lower than T_{min}, the SLA has been violated. And to avoid the detector from detecting the similar anomaly multiple times, we set the detection window as 5 minutes for each SLA violation. A side effect of this method is that the detector is unable to detect another violation until the window is filled again [5].

Specially, this paper assumes that the instantaneous and minuscule requests which are abnormal may not be caused by the application itself. Those extreme cases should not trigger scaling. In order to address this problem, *Microscaler* uses P90, i.e., 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 meanwhile.

D. Scaling-needed Service Determination

Service Power. In this paper, we propose a novel criterion to determine whether it is necessary to scale. Define P_{50} as the average latency for the slowest 50 percent of requests to one micro-service over the last 30 seconds and $P90$ is the average latency for the slowest 10 percent of requests. *Microscaler* leverages the ratio between P_{50} and P_{90} to represent the service power denoted by $\mathbb{P}$, namely $\mathbb{P} = \frac{P_{50}}{P_{90}}$. 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., $P_{90} > P_{50} * 2$ in this paper), it means that the micro service could only handle part of requests but about 10 percent of request to the micro service 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 micro-service needs to scale. If P_{50} is close to P_{90} (e.g., $P_{50} * 1.2 > P_{90}$ in this paper), it means that the micro service could handle most of requests. Therefore, the service power is strong and the micro service does not need to scale at all. Fig.3 and Fig. 4 show the

latency changes along with different percentages at normal and abnormal state respectively. From these two figures, we observe that $\mathbb{P}$ is close to 1 at normal state while $\mathbb{P}$ is higher than 2 at abnormal state.

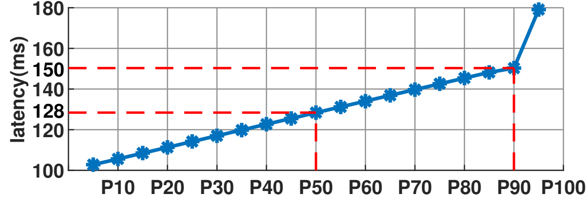

Fig. 3. An example of service power at normal state

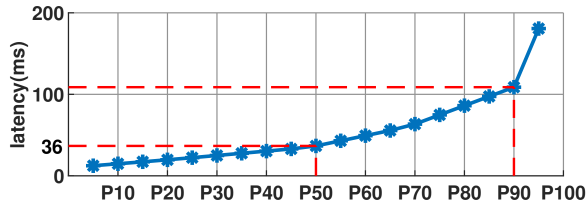

Fig. 4. An example of service power at abnormal state

We assume that all abnormal services that are experiencing SLA violations in a micro-service application could been found. After getting the list of abnormal services, *Microscaler* uses the service power to determine scaling-needed services in order to decrease unnecessary scaling.

Algorithm 1 The Service Determination Algorithm

Input: A list of abnormal services, A

Output: A list of scale out services, S_o, A list of scale in services, S_i,

1: **for all** $service \in A$ **do**
2: **if** $service.P_{50} * 2 < service.P_{90}$ **then**
3: $S_o.append(service)$
4: **else if** $service.P_{50} * 1.2 > service.P_{90}$ **then**
5: $S_i.append(service)$
6: **else**
7: $continue$
8: **end if**
9: **end for**
10: **return** S_o, S_i

E. Auto-scale Decision

After determining the services which need to scale, it is equally important to determine how many instances should be scaled out / in precisely and quickly. *Microscaler* combines Bayesian Optimization (BO) [12] 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 iterations and then uses the step-by-step approach to reach the optimal result.

Algorithm 2 The Auto-scale Decision Algorithm

1: $N = BO(Service, Latency)$
2: **while** $Latency < SLA_{max}$ **do**
3: $N = N - 1$
4: $Update\ Latency$
5: **end while**
6: **return** $N + 1$

1) Bayesian Optimization Approach: BO is a 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 performance model [7]. Mathematically, BO aims to find a global minimizer (or maximizer) of an unknown objective function f: $x^* = \arg\min_{x \in \mathbb{X}} f(x)$, where X is the decision space of interest and is often a compact subset of $\mathbb{R}^d$. In this paper, $d = 1$ and X represents the space of service 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 a optimal or sub-optimal solution because BO focus its search on areas that have the largest expected improvements [7]. While Deep Neural Network [13] can also be used for black-box optimization, it requires lots of training samples which are very difficult to work in real-world systems.

For given scaling-needed services and workload, our goal is to find the optimal service scale to minimizes the cost of each visit under the condition that satisfying the SLA. The latency of front-end depends on the number of service instances vector $\vec{x}$. Since the number of services which are not necessary to scale will not be updated in BO, the performance model only considers the service in the plann list. Let $Price(\vec{x})$ be the price for services in the list. We formulate the performance model as follows:

$$\begin{aligned} &\underset{\vec{x}}{minimize}\, Cost(\vec{x}) = Price(\vec{x}) * Latency(\vec{x}) \\ &subject\ to\ \ SLA_{min} \leqslant Latency(\vec{x}) \leqslant SLA_{max} \end{aligned} \tag{1}$$

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, $Cost(\vec{x})$ is unknown beforehand but can be observed through experiments.

The target function shown in Eqn(1) is designed to minimize $Cost(\vec{x})$ without further constraints. However, $T_{min} \leqslant T(\vec{x}) \leqslant T_{max}$ must need to consider. It means that when selecting the next number of instance to evaluate, *Microscaler* should have a bias towards one that is likely to satisfy the SLA constraint. To achieve this goal, we modify the cost model as:

$$\underset{\vec{x}}{minimize}\, Cost(\vec{x}) = Price(\vec{x}) * T(\vec{x}) + \omega * T(\vec{x}) \tag{2}$$

where if $T_{min} \leqslant T(\vec{x}) \leqslant T_{max}, \omega$ is set as -10, else ω is set as $+10$. ω could be tuned according to different applications.

There are two major choices that must be made when performing BO. First, one must select a prior over functions that will express assumptions about the function being optimized. *Microscaler* chooses the Gaussian process prior,

due to its flexibility and tractability [14]. Then *Microscaler* must select an *acquisition function* next. This paper chooses Gaussian Process Upper Confidence Bound (GP-UCB) since it is an intuitive upper-confidence based algorithm and bounds its cumulative regret in terms of maximal information gain. Therefore, the strategy to update points follows as

$$x_t = \arg\min_{x \in D} \mu_{t-1}(x) - \kappa \sigma_{t-1}(x) \quad (3)$$

where D denotes the search space, $\mu(x)$ is a mean function, $\sigma(x)$ is the standard deviation function and κ is a tunable parameter to balance between exploitation and exploration.

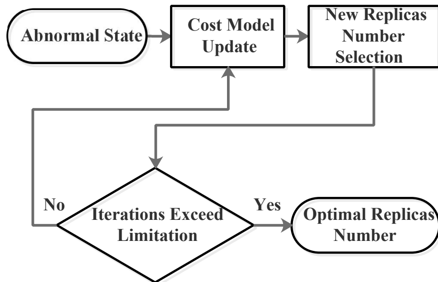

Fig. 5. The workflow of Bayesian Optimization

Fig. 5. shows the process of service scale searching. Searching starts an SLA violation. Then a list of scaling-need services are determined. BO then dynamically picks the next service replicas number to run based on the cost model and feed the result to the cost model. The model will stop and output the optimal number which has been found when it exceeds the maximum iterations.

By modeling the target function as a stochastic process (e.g., a Gaussian Process [15]), BO can compute the confidence interval of target according to one or more observations. A confidence interval is an area that the curve of target function is most likely (e.g., with 95% probability) passing through [7]. For example, in Fig. 6.(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. 6., the yellow star is chosen because the acquisition function indicates that it has the most potential gain. There is no doubt that the more iterations, the better result in BO. We will discuss how to select appropriate maximum iterations in part IV.

2) The Step-by-Step Approach: BO model can not 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 modifies one service instance each time until the latency of front-end is lower than the SLA.

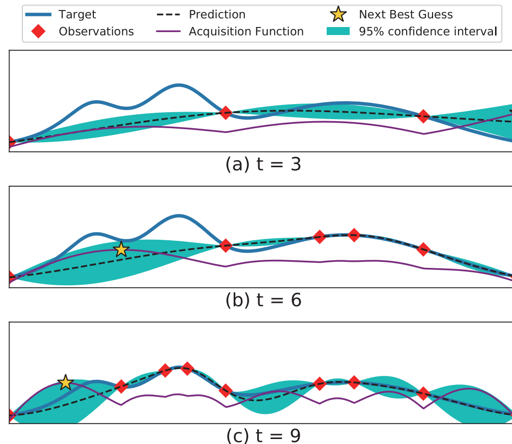

Fig. 6. An example of BO working process

F. Scale Action

This paper has built a Docker [16] image for each micro service in application and stored them in a private Docker Hub. By including all necessary dependencies in the image, a micro service can be executed on any platform running a container engines quickly. *Microscaler* modifies the micro service's processing capacity by adding containers or removing containers horizontally. Therefore *Microscaler* can call the cloud Replicate Controller API to keep the service replicas number in cloud with the output of auto-scale decision.

IV. Experimental Evaluation

Experiment Settings. *Microscaler* is evaluated in a self-constructed distributed system. The test system contains 10 virtual machines (VMs) that host a micro-service benchmark. Each VM has a 2-core 2.40GHz CPU, 6GB of memory and runs with Ubuntu OS. All the services are managed by an Istio-enabled Kubernetes [10] platform. Kubernetes is an open-source system for automating deployment and management of containerized application, which is one of best platform for deploying and running micro services. In our experiment, Envoy is deployed as a sidecar to the relevant service in Kubernetes. Since Envoy provides load balancing for micro-services, we only need to consider the performance of the whole service rather than each fine-grained service instance.

Benchmark. Hipster-Shop [8] is a micro-service 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. Hipster-Shop is composed of 10 micro services written in different languages that talk to each other over gRPC. In addition, Hipster-Shop contains a load generator, which defines user behavior, to simulate the visits to the website with simultaneous users. We can change the performance of Hipster-Shop by adjusting the number of micro-services instance and change the request volume by adjusting the concurrent users.

Limited by the hardware resources of our testbed, we have a maximum of 15 microservice instances for each service in 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 the process of scale out. We set the upper bound of the front-end's service request latency T_{max} as $2s$.

A. Effectiveness Evaluation

After finding the scaling-needed services, *Microscaler* will scale them serially. Fig. 7. shows the BO search process in different services and different iterations. First, we can conclude that BO can find an optimal or sub-optimal replica number just after a few iterations. In addition, the number of iteration affects the result of replica number. Each iteration needs to wait some time for starting containers and obtaining latency, it means that the searching process will cost a lot of time. So selecting an appropriate iteration number is important not only in generating a good result but also in reducing search time. From Fig. 7., we can observe that the result of 3 iterations outperforms the ones with 4 iterations or 5 iterations. But the result of 4 iterations has small difference with 5 iterations. It means that the suitable iteration number is 4 for *Microscaler* because BO can obtain an optimal or sub-optimal result in 4 iterations with the maximum limit of 15 microservice instances.

B. Comparisons

Fig. 8. shows the process to reach the desired service scale in some auto-scalers which adopt different scaling policies. The Amazon Emulated auto-scaler is a copied version of the AWS auto-scaling policy. We deliberately use a modifying step of one and two replicas to show the difference of each auto-scalers. And the fuzzy auto-scaler emulates the model proposed by [6]. In order to meet the SLA requirements as quickly as possible, we scale the replicas number to the maximum 15 first, and then wind-down the replicas until finding the minimal number that satisfies the SLA. For the white-box auto-scaler, it will calculate the model between workload and service scale explicitly in advance. Therefore, the white-box auto-scaler can reach the optimal service scale by calculating models directly when the workload changes.

The left graphs of Fig. 8. show the auto-scalers specific updating actions to reach the target replicas number. There is no doubt that the Amazon emulated auto-scaler which modifying one replicas each time will accurately find the optimal minimal number that meets the SLA. It is also obvious that it needs more steps to find the optimal replicas number than other three methods. Hence, in Fig. 8., the optimal replicas number is 5, the result of BO auto-scaler is 6. This means that BO auto-scaler may cause the unnecessary cost to serve the same workload because they may find the sub-optimal replica number. *Microscaler* searches continuously after BO until finding the optimal service scale. Therefore, it also reaches 5 service instances. With regard to the time of finding optimal service scale, White-box auto-scaler can reach its ideal scale in only 1 step, and *Microscaler* needs 5 steps. Amazon auto-scaler(1 replicas), Amazon auto-scaler(2 replicas) and fuzzy auto-scaler needs 12 steps, 6 steps, 6 steps respectively. In short, *Microscaler* can reach optimal service scale with a very few iterations.

From Fig. 9, we can observe that *Microscaler* and Amazon auto-scaler(1 replicas) can reach 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. As for the white-box auto-scaler, it can reach optimal service scale if 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.

C. Discussion

TABLE I
THE OVERHEAD OF *Microscaler*

System Module	Cost
Data Collection	$5\% \pm 1\%$ Single CPU Utilization
SLA Violation Detection	about 0.2 second
Scaling-needed Comfirmation	about 1 seconds
Autoscaling Decision	about 2 minutes

Overhead. TABLE I shows the overhead of *Microscaler*.The data collection module takes about 5% utilization when collecting service invocation and service latency with the help of Service Mesh. In auto-scale decision module, *Microscaler* consumes about 2 minutes because creating containers needs some time after changing service's replica number. From the aforementioned conclusion, the state-of-the-art methods need more time than *Microscaler*. Overall, *Microscaler* is a light-weight and flexible auto-scaler which can be deployed in a large-scale micro-service system readily.

Limitation. Firstly, nowadays, almost all the products of Service Mesh are only suitable for the container-based Cloud and they lack of support for VM-based Cloud, so *Microscaler* is not appropriate for VM autoscaling. 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.

V. RELATED WORK

Resource estimation is the core of auto-scaling as it determines the efficiency of resource provisioning. Recently, Various approaches have been proposed to conduct auto-scaling based on resource estimation.

Rule-Based Approach. Rule-based approaches define a set of rules consisting of triggering conditions and corresponding actions, such as "If CPU utilization exceeds 80%, add one instances". Theoretically, simple rule-based approaches involve no accurate resource estimation but empirical estimation. As the simplest version of auto-scaling, it commonly serves as a baseline for comparison and is used as the basic scaling framework for works that focus on other aspects of auto-scaling, such as the work done by Dawound et al. [17], which aims to compare vertical scaling and horizontal scaling.

Fuzzy-Based Approach. The core of Fuzzy-based approaches is a set of predefined "If-Else" rules. The major

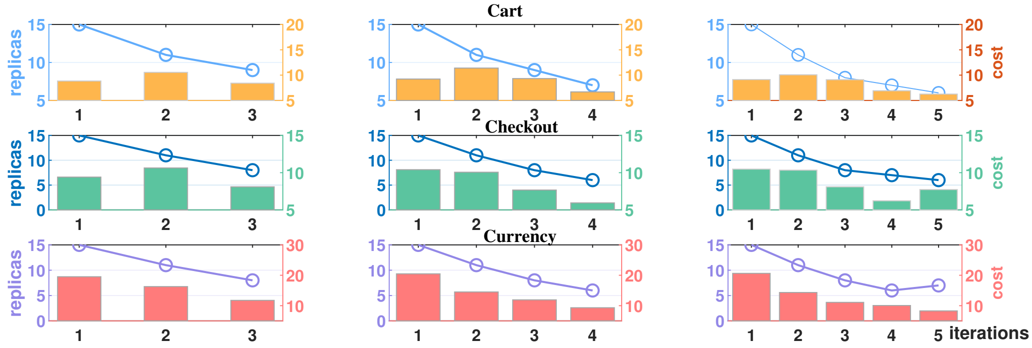

Fig. 7. The autoscaling results for different services obtained by BO

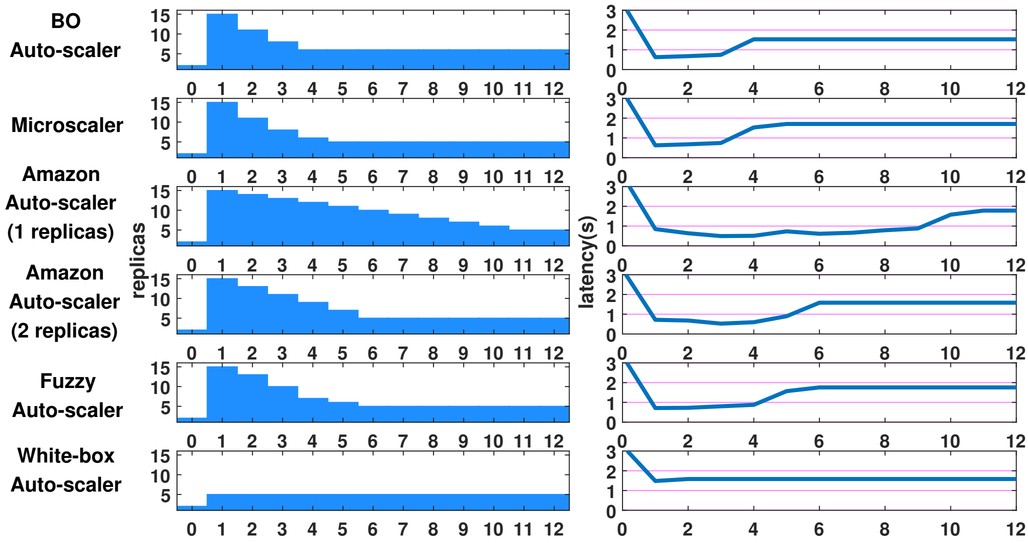

Fig. 8. Changes of replicas and request latency obtained by different autoscalers

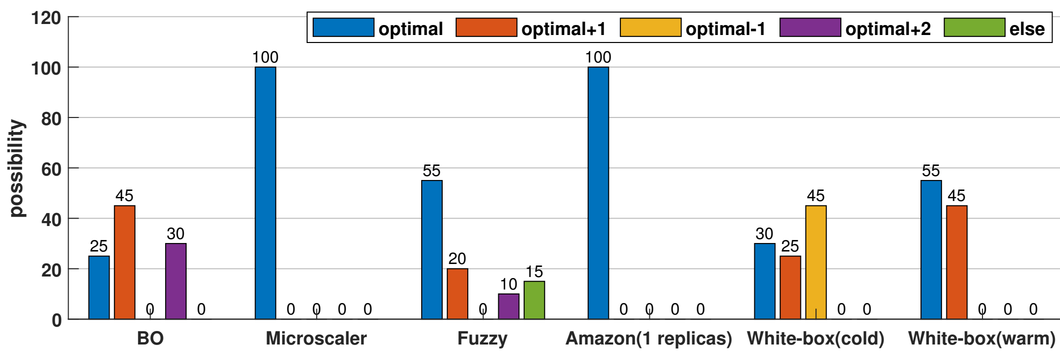

Fig. 9. The probability of reaching optimal service scale by different autoscalers

advantage of fuzzy inference compared to simple rule-based reasoning is that it allows users to use linguistic terms like "high, medium, low" instead of accurate numbers to define the conditions and actions [3]. Fuzzy-based approaches work as follows: the inputs are first fuzzified by defined functions; then the fuzzified inputs are used to trigger the action parts in all the rules in parallel; the results of rules are then combined and finally defuzzied as the output for scaling operation. Representative approaches of this kind include the one proposed by Frey et al. [18].

Application Profiling-Based Approach. The application profiling based approach tests the saturation point of resources for one specific application using a synthetic or recorded real workloads. An application can precisely obtain the knowledge of how many resources are just enough to handle the given workload intensity concurrently. Offline profiling can produce the complete spectrum of resource consumption under different levels of workload [19]. Jiang et al. [20] proposed a quick online profiling for multi-layer applications by studying the correlation of resource requirements that different tiers pose on the same type of VM. But when the application updates, the profiling needs to update accordingly.

Machine Learning Approach. Machine learning approaches are applied to dynamically build the resource model under different workloads. In this way, service providers can use the auto-scalers without customized settings and preparations. Online machine-learning algorithms(e.g. Reinforcement learning [21] [22], Regression [23] [24]) are more robust to changes during production as the learning algorithm can adaptively adjust the model on the fly regarding any notable events. Though offline learning can also be used to fulfill the task, it inevitably involves human interventions and thus loses the benefit of using machine learning [3].

VI. Conclusion and Future Work

This paper designs and implements *Microscaler*, a system to help application providers pinpoint the scaling-needed services and scale them automatically in service-mesh-enabled micro-service environments. A novel criterion i.e., *service power* is proposed to determine the scaling-needed services. Moreover, a black-box optimization approach is presented to optimize the scaling cost. The experimental evaluations in a micro-service benchmark environment show that *Microscaler* converges to the optimal service scale faster than several state-of-the-art methods. we will try to adjust multiple scaling-need services simultaneously with the help of BO in microservice environment.

Acknowledgments

The work described in this paper was supported by the National Key Research and Development Program (2016YFB1000101), the National Natural Science Foundation of China (61802448,U1811462), the Guangdong Province Universities and Colleges Pearl River Scholar Funded Scheme (2016) and the Pearl River S&T Nova Program of Guangzhou (201710010046). The corresponding author is Pengfei Chen.

References

[1] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings*, 2018, pp. 3–20.

[2] S. Newman, *Building microservices - designing fine-grained systems, 1st Edition*. O'Reilly, 2015. [Online]. Available: http://www.worldcat.org/oclc/904463848

[3] 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.

[4] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *11th International Conference on Autonomic Computing, ICAC '14, Philadelphia, PA, USA, June 18-20, 2014.*, 2014, pp. 57–64.

[5] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*, 2017, pp. 469–478.

[6] B. Liu, R. Buyya, and A. N. Toosi, "A fuzzy-based auto-scaler for web applications in cloud computing environments," in *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings*, 2018, pp. 797–811.

[7] 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 *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, 2017, pp. 469–482.

[8] "Hipster," https://github.com/GoogleCloudPlatform/microservices-demo, 2019, [Online; accessed 1-February-2019].

[9] "Istio," https://istio.io, 2019, [Online; accessed 1-February-2019].

[10] "Kubernetes," https://kubernetes.io, 2019, [Online; accessed 1-February-2019].

[11] J. Walter, D. Okanovic, and S. Kounev, "Mapping of service level objectives to performance queries," in *Companion Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, 2017, pp. 197–202.

[12] 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," *arXiv preprint arXiv:1012.2599*, 2010.

[13] 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.

[14] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.*, 2012, pp. 2960–2968.

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

[16] "Docker," https://www.docker.com, 2019, [Online; accessed 1-February-2019].

[17] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. Vahdat, "Model-based resource provisioning in a web service utility," in *4th USENIX Symposium on Internet Technologies and Systems, USITS'03, Seattle, Washington, USA, March 26-28, 2003*, 2003.

[18] S. Frey, C. Lüthje, C. Reich, and N. L. Clarke, "Cloud qos scaling by fuzzy logic," in *2014 IEEE International Conference on Cloud Engineering, Boston, MA, USA, March 11-14, 2014*, 2014, pp. 343–348.

[19] U. Sharma, P. J. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*, 2011, pp. 559–570.

[20] D. Jiang, G. Pierre, and C. Chi, "Resource provisioning of web applications in heterogeneous clouds," in *2nd USENIX Conference on Web Application Development, WebApps'11, Portland, Oregon, USA, June 15-16, 2011*, 2011.

[21] 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.

[22] W. Iqbal, M. N. Dailey, and D. Carrera, "Unsupervised learning of dynamic resource provisioning policies for cloud-hosted multitier web applications," *IEEE Systems Journal*, vol. 10, no. 4, pp. 1435–1446, 2016.

[23] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao, "Energy-aware server provisioning and load dispatching for connection-intensive internet services," in *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, 2008, pp. 337–350.

[24] D. Grimaldi, V. Persico, A. Pescapè, A. Salvi, and S. Santini, "A feedback-control approach for resource management in public clouds," in *2015 IEEE Global Communications Conference, GLOBECOM 2015, San Diego, CA, USA, December 6-10, 2015*, 2015, pp. 1–7.