Computer Science and Application 计算机科学与应用, 2022, 12(6), 1685-1699
Published Online June 2022 in Hans. http://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2022.126169

Hans汉斯

云计算中基于微服务监控系统的自动性能分析方法

池昱儒[1]，何志恒[1]，余兴胜[2]，夏文俊[2]

[1]武汉大学计算机学院， 湖北 武汉
[2]中铁第四勘察设计院集团有限公司，湖北 武汉

收稿日期：2022年5月26日；录用日期：2022年6月23日；发布日期：2022年6月30日

摘 要

随着云计算技术在各个领域的不断深入与发展，众多微服务之间复杂的调用关系以及相互影响也给微服务的数据监控，运行维护带来了新的挑战。目前针对多服务应用的自动伸缩方法一次只能伸缩一个瓶颈微服务，会导致瓶颈转移问题；或者依赖于服务调用图谱的静态性，不能解决动态负载下多服务应用的自动伸缩问题。本文提出了一种云计算环境下基于微服务监控系统的多服务应用自动性能分析方法，均衡地将入口服务的SLA分解为每个服务的响应时间SLO，精准获得每个服务在不超过响应时间SLO的条件下能够服务的最大流量负载和对应资源利用率，为多微服务应用集群伸缩提供了精确性能指标。

关键词

云计算，微服务应用，微服务监控系统，自动性能分析

An Automatic Performance Profiling Method Based on Microservice Monitoring System in Cloud Computing

Shiru Chi[1], Zhiheng He[1], Xingsheng Yu[2], Wenjun Xia[2]

[1]School of Computer Science, Wuhan University, Wuhan Hubei
[2]China Railway Siyuan Survey and Design Group Co., Ltd., Wuhan Hubei

Received: May 26th, 2022; accepted: Jun. 23rd, 2022; published: Jun. 30th, 2022

Abstract

With the continuous deepening and development of cloud computing technology in various fields,

文章引用: 池昱儒, 何志恒, 余兴胜, 夏文俊. 云计算中基于微服务监控系统的自动性能分析方法[J]. 计算机科学与应用, 2022, 12(6): 1685-1699. DOI: 10.12677/csa.2022.126169

the complex calling relationship and interaction between many microservices also bring new challenges to the data monitoring, operation and maintenance of microservices. The current automatic scaling method for multi-service applications can only scale one bottleneck microservice at a time, which will lead to the problem of bottleneck transfer; or relying on the static nature of the service call graph, it cannot solve the automatic scaling problem of multi-service applications under dynamic load. This paper proposes an automatic performance profiling method for multi-service applications in cloud computing environment based on a micro service monitoring system, which decomposes the SLA of the ingress service into the response time SLO of each service in a balanced manner, and accurately obtains the maximum traffic load that can be served and the corresponding resource utilization of each service under the condition that the response time SLO does not exceed SLA.

Keywords

Cloud Computing, Microservice Applications, Microservice Monitoring System, Automatic Performance Profiling

1. 引言

用户在部署采用微服务架构的多服务应用[1]之后，会指定一个服务作为入口服务，并为该入口服务的响应时间指定上界作为 SLA。为了满足该 SLA，在流量高峰期通常需要扩大服务该接口的所有服务的服务实例数量以提高服务处理流量负载的能力，但是水平自动伸缩器仅能考虑到单个服务的 RPS 和资源利用率指标(即 SLO)来进行对应服务的自动伸缩，而不能考虑到多服务应用中不同服务之间的相互关联性。其面临的问题是，难以确定微服务的性能瓶颈以及目前针对多服务应用的自动伸缩方法[2]一次只能伸缩一个瓶颈微服务，会导致瓶颈转移问题，一些基于测试和性能分析的伸缩算法[3]能够更为准确地把握到负载、SLO 和服务 SLA 之间的关系，从而更好地提高资源利用率。然而这些伸缩算法通常需要离线分析和侵入式改造，无法在线分析服务的性能。本文的研究重点就是在构建微服务监控系统的基础上，将用户为入口服务指定的 SLA 分解成为多服务应用中不同服务的 SLO，使得自动伸缩器可以根据 SLO 进行自动伸缩。

2. 微服务监控系统方案设计

为了实时获得多服务应用的相关指标，本文采用 Prometheus 来监控多服务应用和 Kubernetes 集群状态，搭建了如图 1 所示的监控系统。

Prometheus 从不同的 Exporter 拉取指标数据，支持不同场景下对应目标的指标暴露。Node Exporter 暴露节点的 CPU 和内存利用量相关的指标；Kubelet 负责暴露 Pod 的 CPU 和内存资源利用量相关的指标；Kube-State-Metrics 负责暴露 Kubernetes 定义的如 Pod、Deployment、StatefulSet 等工作负载计算资源相关的指标。在 Istio 服务网格中，服务实例的边车容器便是服务实例流量相关指标的 Exporter，通过收集、整理这些数据可以得到不同时间段服务之间的调用关系图谱、服务的流量负载、服务的响应时间等指标。

Kiali 从 Prometheus 拉取服务网格中的服务实例的流量指标，梳理出服务调用图谱；Prometheus Adapter 通过 API 聚合的方式，从 Prometheus 拉取数据来实现 Kubernetes 定义的 Metrics API，让集群内

的组件如 HPA 和自适应自动性能分析器可以通过访问 Metrics API 来获取相关指标；Grafana 通过配置数据源的形式对 Prometheus 收集到的指标数据进行可视化与度量分析，并展示给运维管理人员。

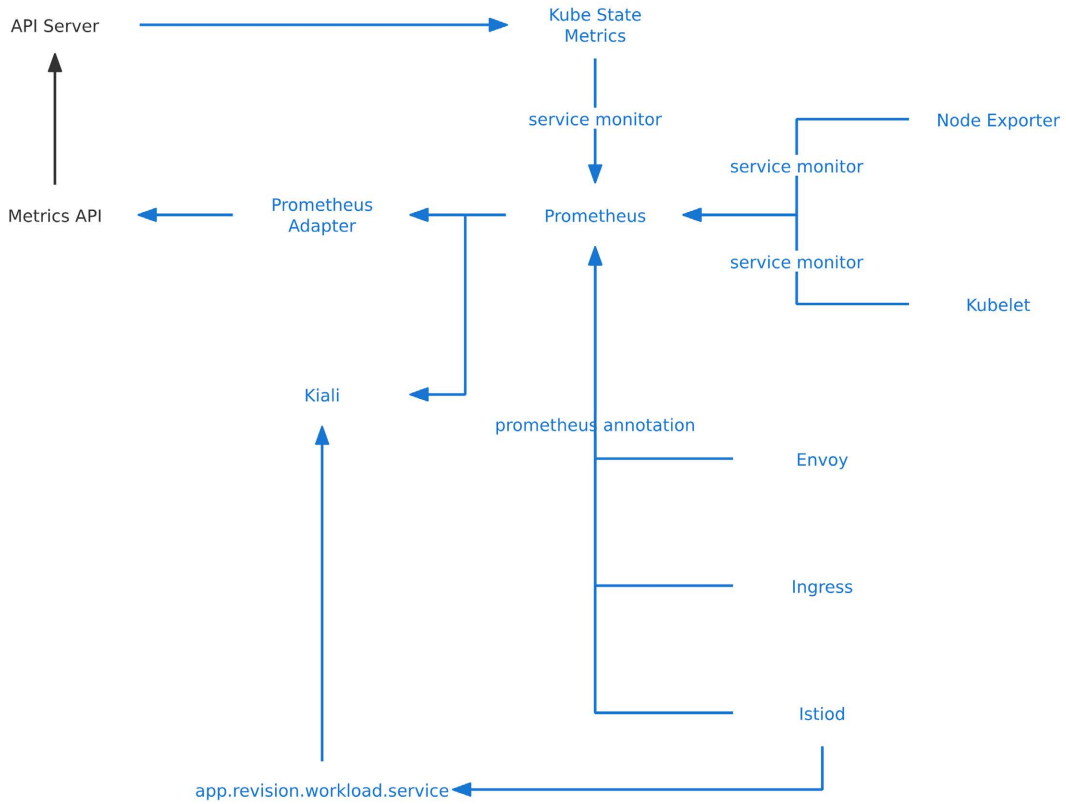


Figure 1. Diagram of monitoring solution for multi-service application
图 1. 针对多服务应用的监控方案

3. 基于微服务监控系统方案的自动性能分析方法

3.1. 服务指标分析

本文采用延迟数据的 P90 分位数来衡量服务所提供接口的性能，P90 分位数即分割出最慢的百分之十延迟数据的临界值。根据研究[4]，延迟的 P90 分位数更能稳定地反映微服务在云环境下的性能情况。

入口服务的响应时间 R_1 实际上由该调用涉及到的所有微服务的响应时间和入口服务自身的处理时间所决定，可表示为式 1：

$$R_1 = f\left(R_2, R_3, \cdots, R_n\right) + S_1 \tag{1}$$

在式 1 中，R_1 表示入口服务的响应时间，R_2 到 R_n 表示入口服务调用的多个后台服务的响应时间，S_1 表示入口服务自身处理请求所需要的时间，f 是一个涉及到线性变换和最大值变换的函数，线性变换的权重由每一次入口服务被调用会导致多少次其他服务被调用来决定，是否有最大值变换由入口服务多次调用其他服务时采用并行还是串行来决定。如果入口服务的调用不涉及到任何其他服务，则其响应时间 R_1 仅跟 S_1 有关。对非入口服务的一次调用的响应时间 $R_2, R_3, \cdots, R_n$ 也可以按照式 1 去推导，最终响应时间 R_1 可以表示为式 2：

$$R_1 = g\left(S_1, S_2, \cdots, S_n\right) \tag{2}$$

在式 2 中，S_i 表示服务i处理请求所需要的时间，g 是一个具有嵌套线性变换和最大值变换的函数。服务 i 对于请求的处理时间 S_i [5]可以表示为 3：

$$S_i = W_i + P_i \tag{3}$$

在式 3 中，W_i 表示等待时间，P_i 表示实际处理请求所需时间，等待时间与服务实例的每秒请求数有关。

3.2. 多服务应用自动性能分析

(一) 流量管理方案

通过调整流向服务的流量负载来探索服务的性能指标是一种快捷、轻量的方式，本文采用 Istio 服务网格的流量治理功能来实现多服务应用自动性能分析中的切流。

Istio 服务网格定义了虚拟服务和目标规则两种计算资源来支持流量管理。目标规则定义了流量流向一个服务的子网时，子网可以通过选择服务下的某服务实例，实现更为精细的路由目标管理。虚拟服务定义了流量流向一个服务的路由规则，即访问该服务的流量应该如何流向对应的服务子网。另外，虚拟服务提供镜像流量及延迟注入等功能。

(二) 入口服务 SLA 分解方法

由式 2 和式 3 可知，当对于入口服务的请求量足够小，而部署的多服务应用处理流量负载能力足够大的时候，每个微服务的请求等待时间为零，请求处理时间等于实际处理时间，本文将多服务应用的这一状态称为多服务应用的最优状态。

为了测试出多服务应用在最优状态下各个微服务的响应时间，本文基于多服务应用模拟和切流技术设计了算法 1，算法流程如图 2 所示。

算法 1　测试最优状态下服务响应时间

Input: 多服务应用中后端微服务数量 n，历史时间窗口 historyWindowSize，比率阈值 ratioThreshold，最小流量百分比 minPercentage，调整步长 step

Output: 每个服务的最优响应时间

```
isOptimal = false;
while isOptimal==false do
    // 调低流向模拟多服务应用的流量百分比
    mockPercentage = max(mockPercentage - step, minPercentage);
    // 等待修改生效
    sleep(waitTime);
    isOptimal = true;
    i = 1;
    while i <= n do
        currentLatency = getCurrentLatency(i);
        // 获得服务 i 在时间窗口 historyWindowSize 内的平均的响应时间
        averageLatency = getAverageLatency(i, historyWindowSize);
        if |currentLatency - averageLatency|/averageLatency > ratioThreshold then
            isOptimal=false;
            break;
        end
    end
end
return 当前每个服务的响应时间
```

Figure 2. Diagram of the service response time algorithm in optimal state
图 2. 最优状态下服务响应时间算法

首先在集群中部署与实际生产中功能相同的模拟多服务应用，其每个服务的服务实例只需部署一个；然后将用户流向生产多服务应用的流量复制作为镜像流量导向模拟多服务应用，通过调整该镜像流量的百分比进而调整模拟多服务应用所需要服务流量的每秒请求量；当导向模拟多服务应用的流量足够小的时候，模拟多服务应用的响应时间即为最优状态下的响应时间，表示为式 4：

$$R^{optimal}=\left(R_1^{optimal},R_2^{optimal},\cdots,R_n^{optimal}\right) \tag{4}$$

在式 4 中，$R_1^{optimal}$ 到 $R_n^{optimal}$ 表示多服务应用处于最优状态时中各个微服务的响应时间。当本文的生产多服务应用以最优状态下的响应时间为目的进行自动伸缩时，本文的生产多服务应用实际上是一个平衡系统[6]，但是其会由于追求更快的响应时间而导致资源浪费。为了最大程度地避免资源浪费，本文尝试使不在最优状态下的各个微服务更为均衡地承担违反 SLA 的风险，将 SLA 分解到各个微服务中，通过将最优状态下各个微服务的响应时间按照入口服务 SLA 和最优响应时间的比率进行同比扩缩后作为各个微服务的响应时间 SLO，可表示为式 5：

$$\begin{cases}\lambda=R_{entering}^{SLA}\Big/R_{entering}^{optimal}\\ R^{SLO}=R^{optimal}\times\lambda\end{cases} \tag{5}$$

(三) 单个服务实例最大负载能力测试方法

在通过金丝雀测试拿到每个微服务的响应时间 SLO 后，需要将其转换为流量负载 SLO 和资源利用率 SLO。根据式 1 可知，一个服务的响应时间取决于其所依赖服务的响应时间和它自身的处理时间，服务自身的处理时间随着需要处理请求数量的增加而增加直至超过服务自身的 SLO。基于此，本文设计如下方法：为每个生产微服务部署一个金丝雀服务实例，将流向生产微服务的流量复制并导向金丝雀服务实例，通过调整镜像流量百分比的大小测试在不超过服务 SLO 的情况下单个服务实例所能承受的最大 RPS 和承受最大 RPS 时的资源利用率。基于此思想设计了算法 2，算法流程如图 3 所示。

算法 2　单个服务最大负载能力测试方法

Input: 将要进行切流的微服务 i，服务 i 响应时间目标 R_i^{SLO}，流量阈值 threshhold，调整步长 step

Output: 服务实例所能承受最大 RPS 以及对应的资源利用率

```
currentLatency = getCurrentLatency(i)
currentRPS,currentUtilization = 0,0
while currentLatency < R_i^SLO do
    currentRPS, currentUtilization = getCurrentRPSAndUtilization(i)
    // 判断每秒请求数是否超过阈值
    if canaryRps < threshhold then
        // 指数增加流量百分比
        canaryPercentage = min(canaryPercentage*2, 100);
    else
        // 线性增加流量百分比
        canaryPercentage = min(canaryPercentage + step, 100);
    end
    // 等待修改生效
    sleep(waitTime);
end
return currentRPS、currentUtilization
```

Figure 3. Diagram of the algorithm for the maximum load capacity of a single service
图 3. 单个服务最大负载能力算法

3.3. 多服务应用部署架构

基于上述入口服务 SLA 分解方法和单个服务实例最大负载能力测试方法，本文设计了如图 4 所示的多服务应用部署架构。

多服务应用的部署分为两块，一个是生产多服务应用，用来给用户提供功能服务以及支持单个服务最大负载能力测试；另一块是模拟多服务应用，用来支持形成最优状态下的多服务应用，从而支持将入口服务的 SLA 分解成多个服务的响应时间 SLO 以及为生产多服务应用中的金丝雀服务实例提供稳定的下游响应时间。

整个多服务应用通过一个 Istio 虚拟服务暴露给用户，流向该虚拟服务的流量将被导向生产多服务应用的入口服务；另外，该流量的镜像流量将流向模拟多服务应用的入口服务。每一个服务都通过一个 Istio 虚拟服务暴露给上游服务，如果流量来源是上游服务中的主部署单元，则将流量导向本服务的主部署单元，并且复制镜像流量导向本服务的金丝雀服务实例；如果流量来源是上游服务的金丝雀实例，则将流量导向模拟多服务应用中对应的下游服务。

本文首先使得模拟多服务应用始终处于最优状态下；为了弥补最优状态下的响应时间与分解后的 SLO 响应时间之间的差距，本文采用 Istio 虚拟服务提供的延迟注入功能，即将生产应用中的金丝雀服务实例到虚拟多服务应用的每一个请求都注入对应下游服务的 $R^{SLO}_{downstream} - R^{optimal}_{downstream}$ 时间。这样，金丝雀服务实例的响应时间的影响因素就仅与访问金丝雀服务实例的流量有关。

为了使得模拟多服务应用始终处于最优状态，本文通过两种策略：

对于模拟多服务应用，本文会提供扩容上限和下限，通过记录模拟多服务应用中每个服务的响应时间，当超过本文通过分解得到的 SLO 响应时间时，增加该模拟服务服务实例数量直至超出所设定的上界。

为了使得流向模拟多服务应用的流量在不影响测试的情况下尽可能地少，本文将生产多服务应用中金丝雀服务实例的数量设置为 1。

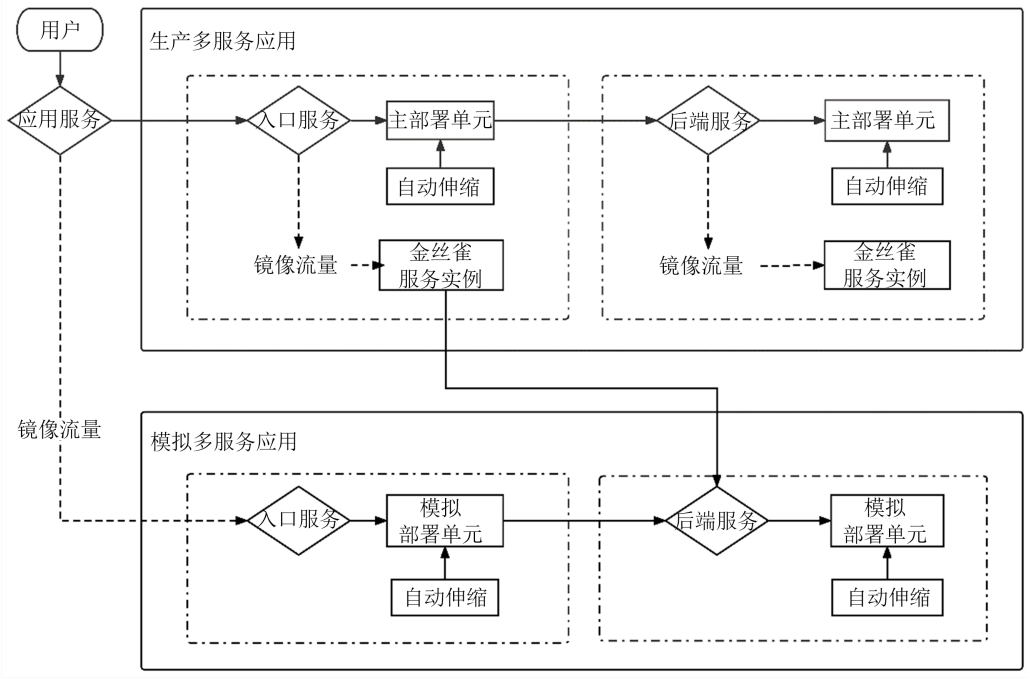


Figure 4. Diagram of the multi-service application deployment architecture
图 4. 多服务应用部署架构

4. 实验与分析

4.1. 实验环境

Kubernetes 版本 1.22.2，一个 Master 节点，两个 Worker 节点，每个节点 40 CPU、200 G 内存，centos7 操作系统。Docker 版本 20.10.9，Istio 版本 1.11.4，Kube-Prometheus 版本 0.10，Prometheus Adapter 版本 3.0.0。

采用 Kiali 作为服务调用拓扑图的生成工具，Grafana 作为数据可视化和度量分析工具，以 Prometheus 作为数据源。

本文所提出的微服务监控系统针对的是多服务应用程序，因此本文选择了 Online Boutique 多服务应用作为测试样例，以验证监控系统及自动性能分析算法的有效性。

4.2. 实验效果评估标准

(一) 是否有瓶颈转移现象。(二) SLA 违反频率和程度。(三) 资源利用率。

为了对自动伸缩器的性能进行对比分析，本文通过实验评估如下两种自动伸缩器的性能：

- Kubernetes 默认水平自动伸缩器：伸缩器采用用户指定的 CPU 资源利用率阈值来计算伸缩的目标服务实例数量。
- 基于自动性能分析的自动伸缩器：伸缩器采用本文提出的多服务应用自动性能分析器得到的 SLO 响应时间，最大流量负载 RPS 和 CPU 资源利用率，自动伸缩器将测试出的目标指标作为阈值进行扩容触发和目标服务实例数的计算。

4.3. 实验过程和结果分析

对于 Online Boutique 应用，本文采用 100 ms 作为前端提供服务的 SLA，测试出的各个服务的响应时间 SLO 和最大负载能力见表 1。

Table 1. Online Boutique service response time SLO and maximum load capacity
表 1. Online Boutique 服务响应时间 SLO 和最大负载能力

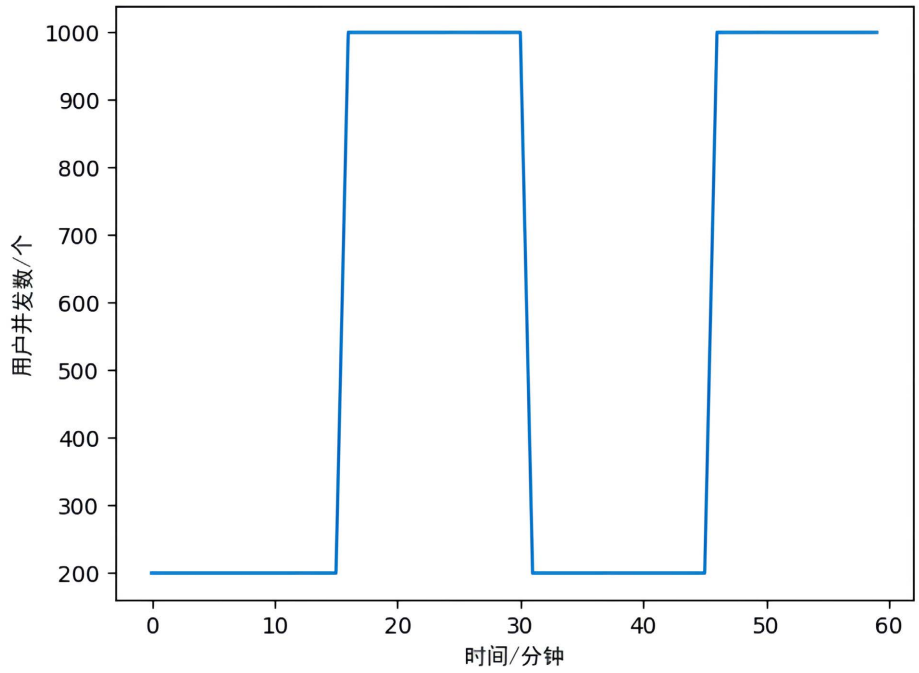
服务	响应时间	最大 RPS	CPU 利用率
EmailServcice	5 ms	17.5*	50%*
CheckoutService	210 ms	2294 m	50%
RecommendationService	20 ms	26,266 m	47%
Frontend	100 ms	11,466 m	62%
PaymentService	5 ms	17.5*	50%*
ProductcatalogService	10 ms	219,933 m	110%
CartService	20 ms	20,866 m	17%
CurrencyService	10 ms	120,399 m	75%
ShippingService	5 ms	17.5*	50%*
AdService	5 ms	109,333	40%

表 1 中数据上标*表示单个服务实例已经可以承受负载，在测试中无需进行扩容，所以最大 RPS 被设置为压测过程中服务所承载过的最大 RPS，最大资源利用率被设置为 50%。

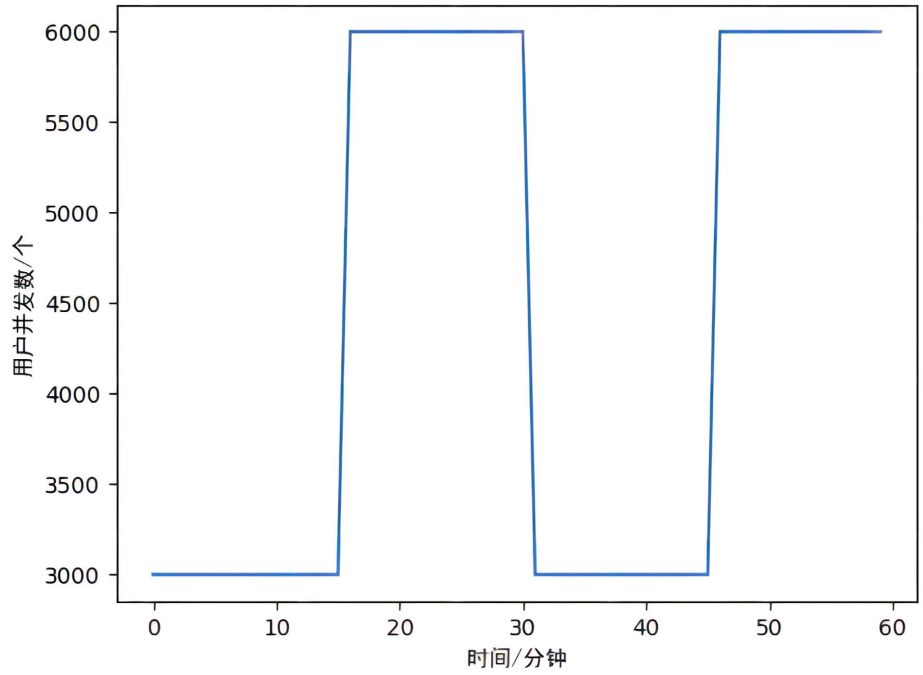
实验采用两种负载方式：突发负载采用一高一低脉冲式的用户波动来产生请求数，高低负载每 15 min

切换一次，用户数量变化曲线如图 5 所示：

本文采用描述了阿里云近两千个微服务 12 个小时的运行信息的微服务数据集[7]的入口服务当中流量负载变化最明显的负载数据作为测试的真实负载，用户数量变化曲线如图 6 所示：

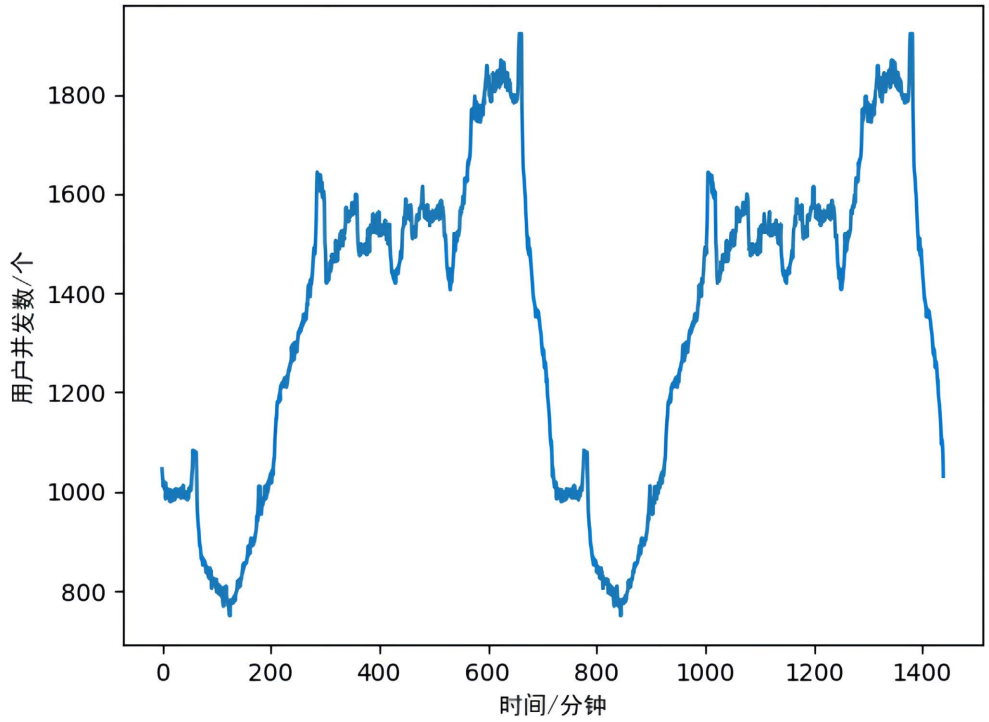


(a) Online Boutique

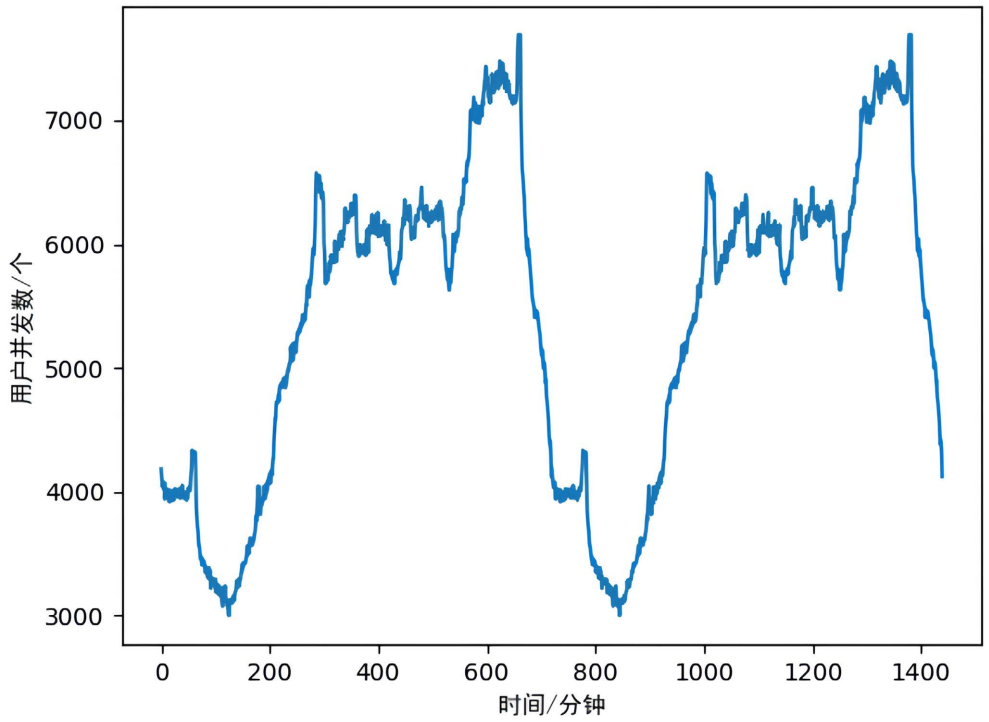


(b) 自动布桥

Figure 5. Diagram of the curve of the number of concurrent users with sudden load
图 5. 突发负载并发用户数量变化曲线



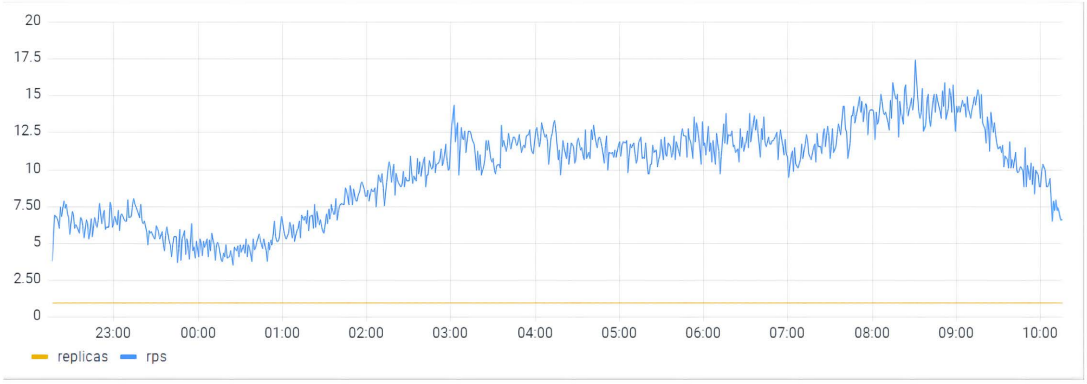
(a) Online Boutique



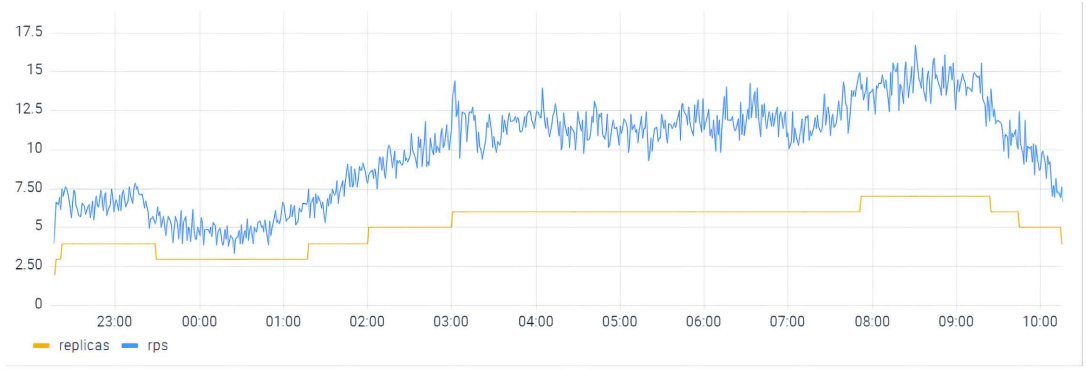
(b) 自动布桥

Figure 6. Diagram of the curve of the number of concurrent users under real load
图 6. 真实负载并发用户数量变化曲线

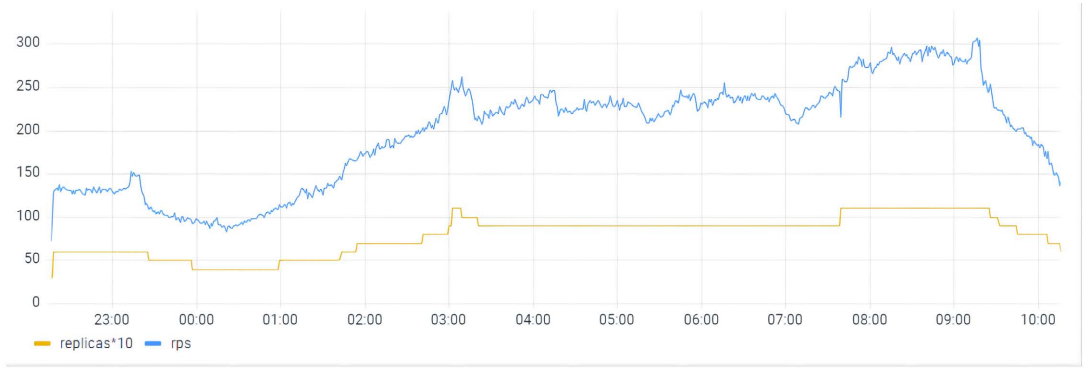
在自动性能分析下，采用真实负载进行测试时，服务实例数量随每秒请求数变化的曲线如图 7：



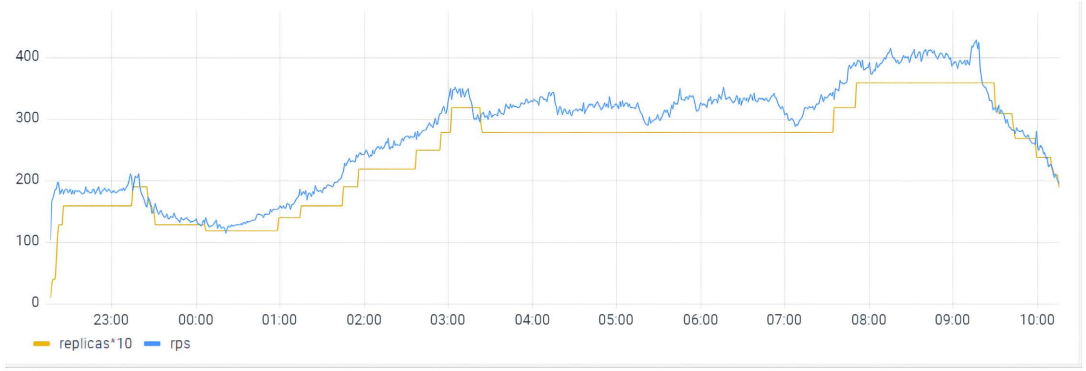
(a) EmailService



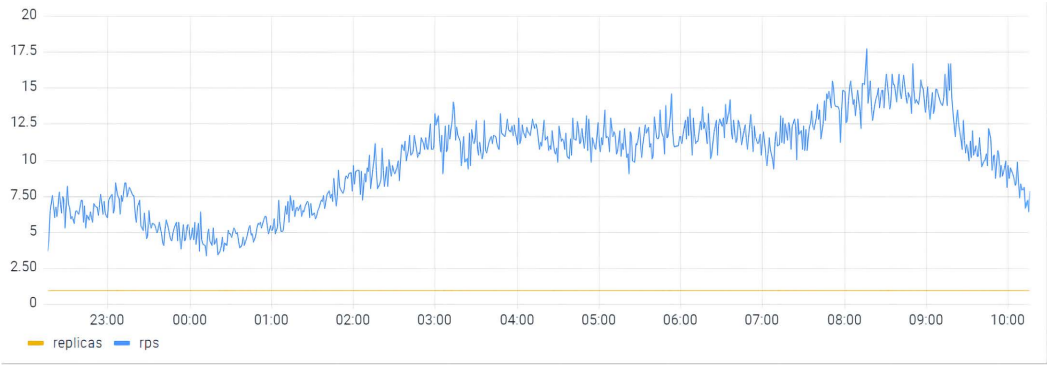
(b) CheckoutService



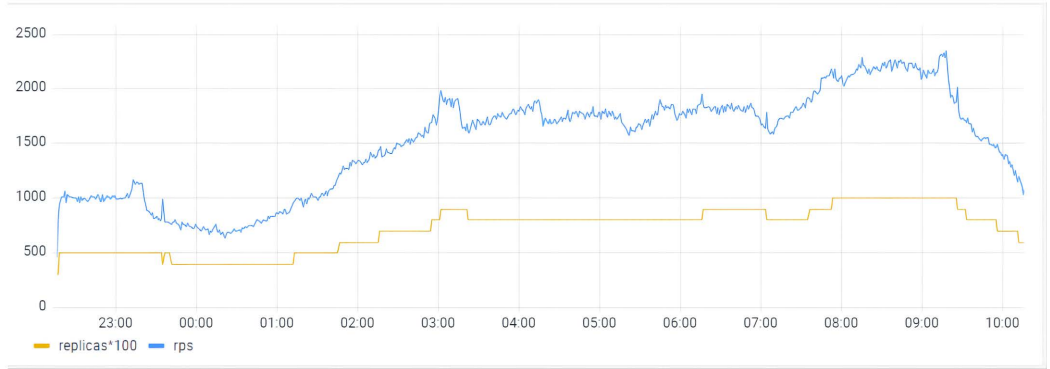
(c) RecommendationService



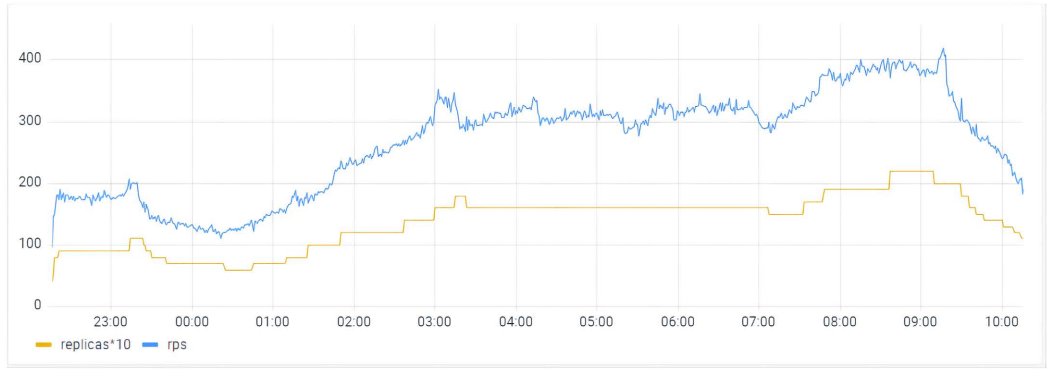
(d) Frontend



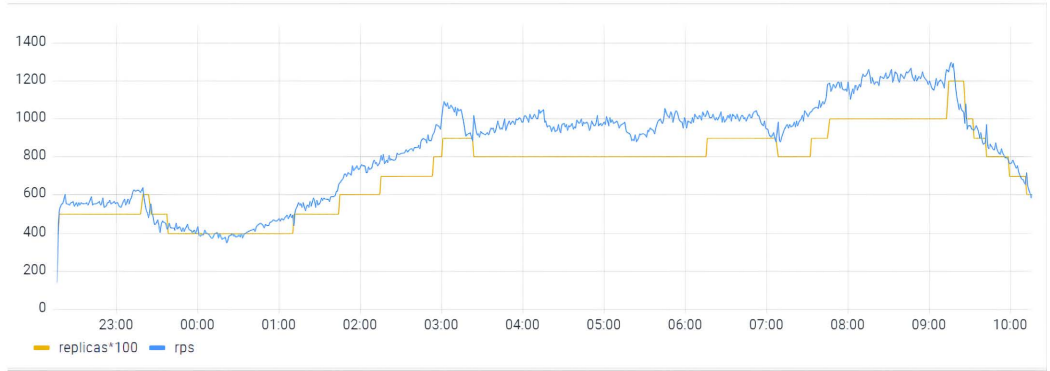
(e) PaymentService



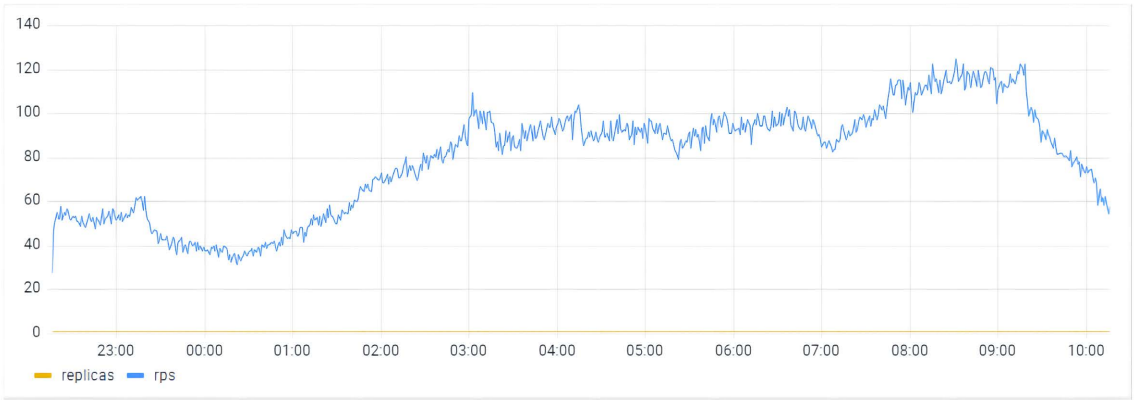
(f) ProductcatalogService



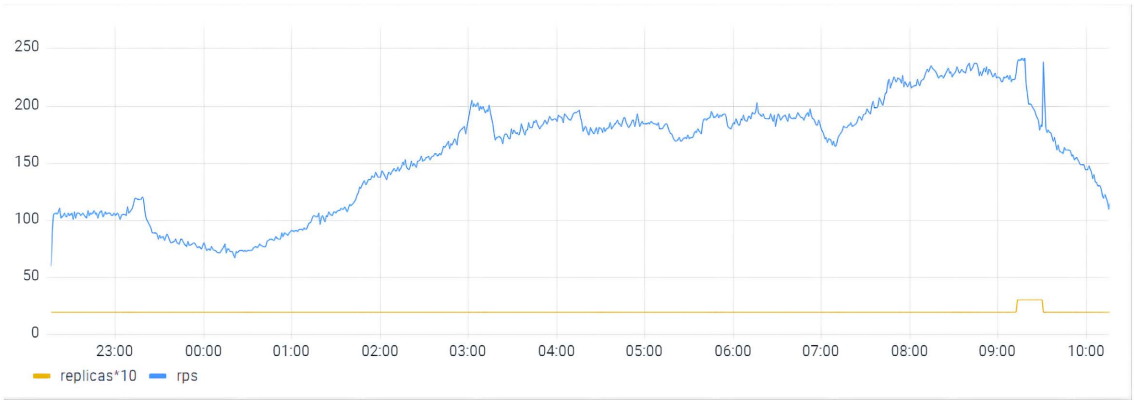
(g) CartService



(h) CurrencyService



(i) ShippingService

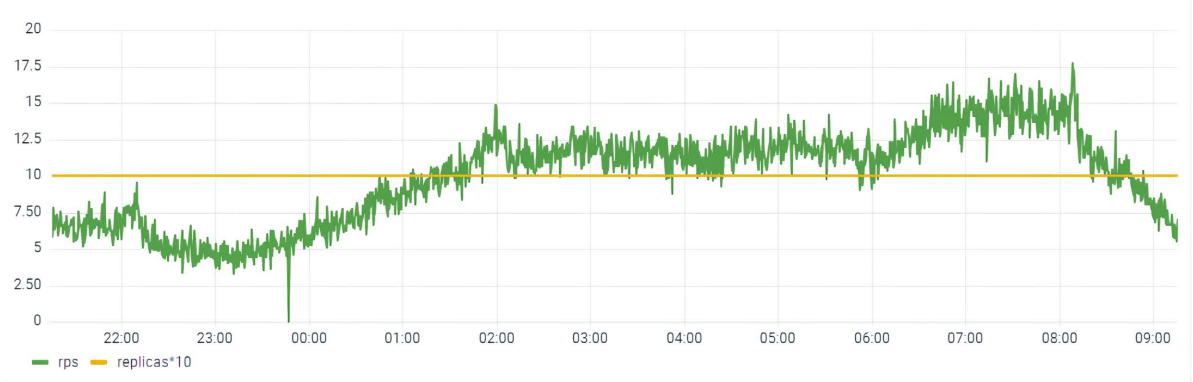


(j) AdService

Figure 7. Diagram of Online Boutique App, real load, automatic performance analysis: autoscaler responsiveness
图 7. Online Boutique 应用、真实负载、自动性能分析：自动伸缩器响应速度

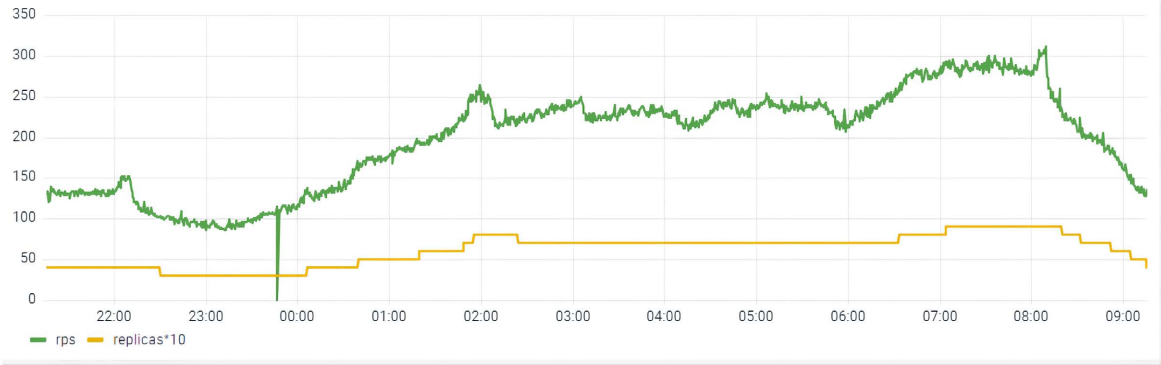
由图 7 可知：除了少数由于流量负载不足而无需伸缩服务实例的服务之外，基于自动性能分析的自动伸缩器对于其他服务都能够及时的响应流量负载的变化，随着流量负载的增加而增加服务实例数量以使得 SLA 不被违反，随着流量负载的减少而减少服务实例数量以节省资源。

在 CPU 利用率阈值为 50%的伸缩策略下，采用真实负载进行测试时，服务实例数量随每秒请求数变化的曲线如图 8：

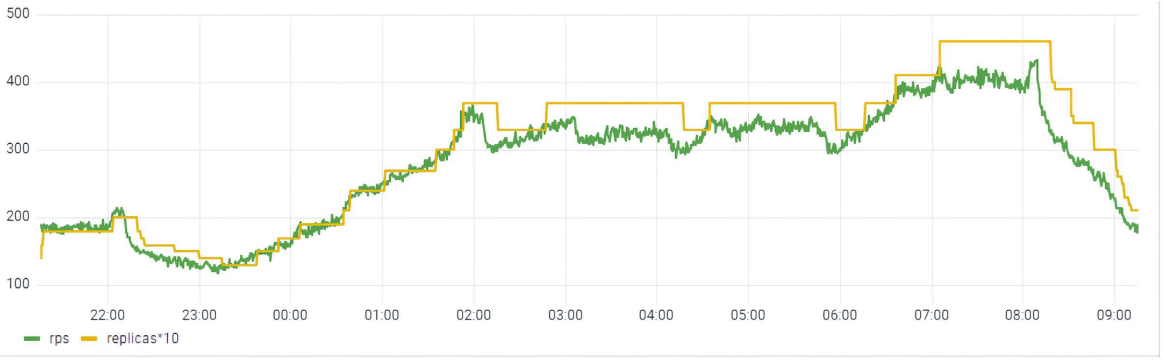


(a) EmailService

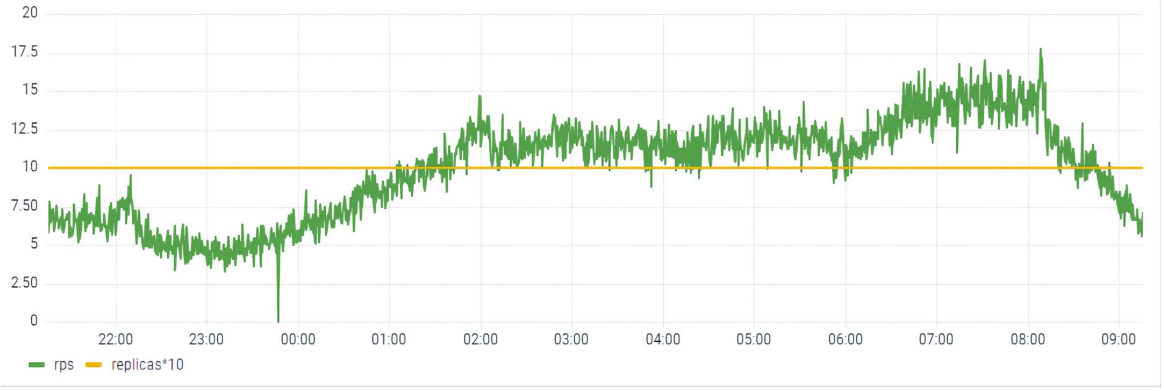
(b) CheckoutService



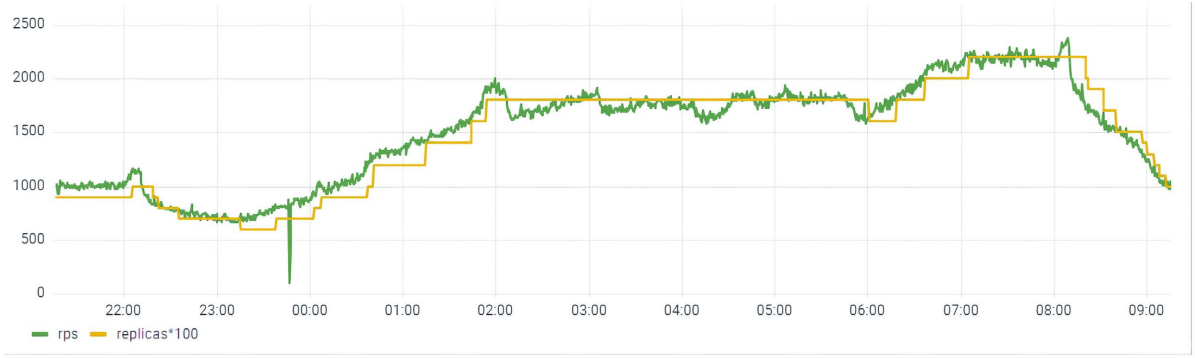
(c) RecommendationService



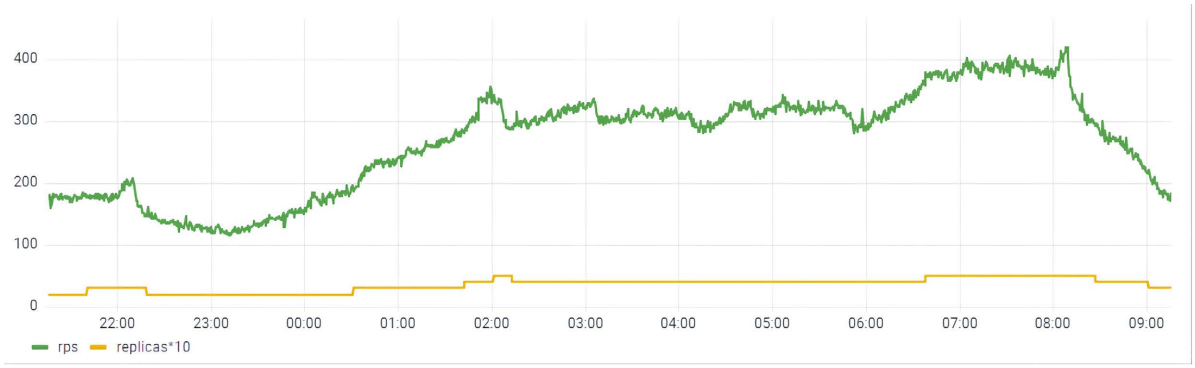
(d) Frontend



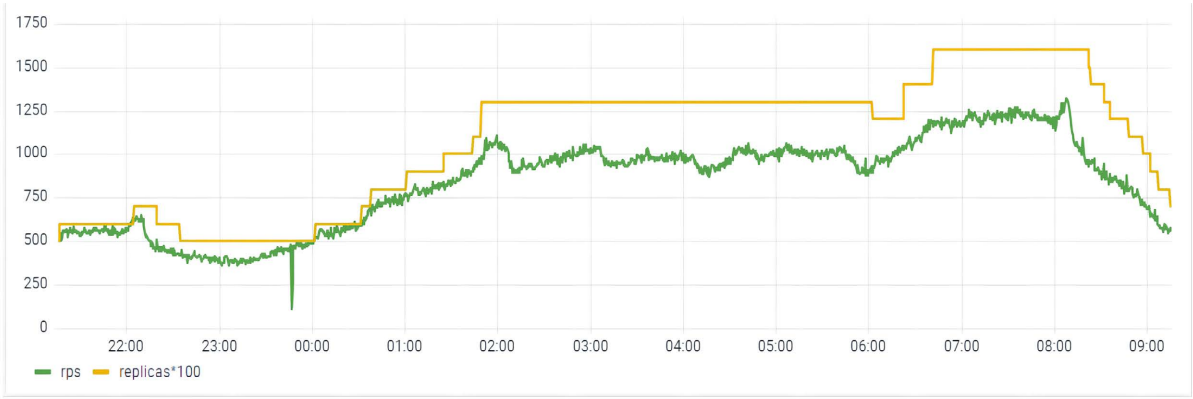
(e) PaymentService



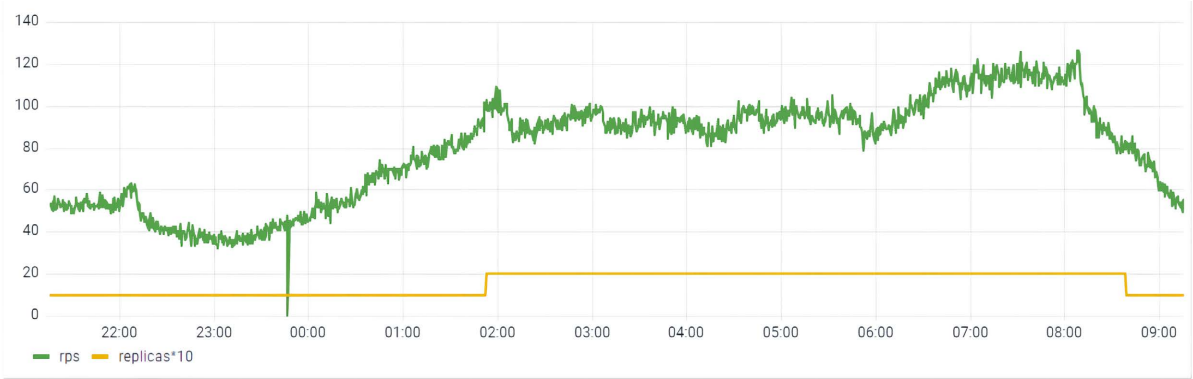
(f) ProductcatalogService



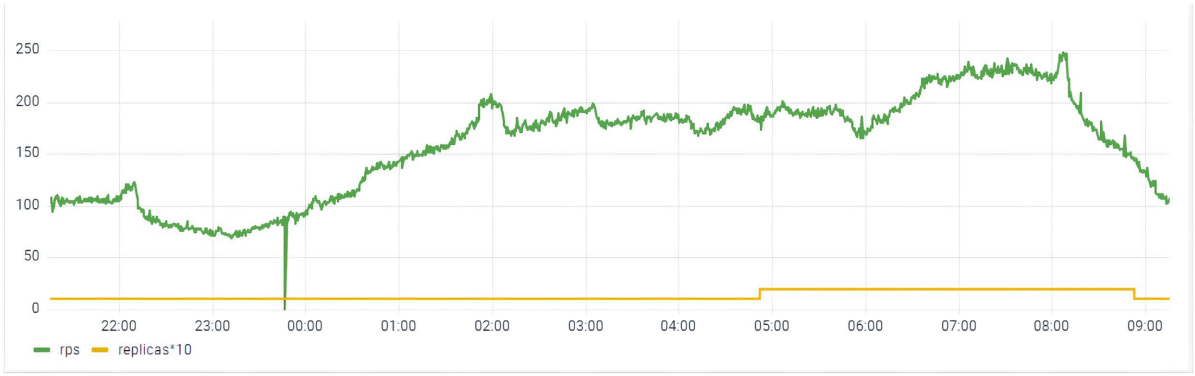
(g) CartService



(h) CurrencyService



(i) ShippingService



(j) AdService

Figure 8. Diagram of Online Boutique App, real load, CPU threshold 50%: autoscaler responsiveness
图 8. Online Boutique 应用、真实负载、CPU 阈值为 50%：自动伸缩器响应速度

由图 8 可知：除了少数由于流量负载不足而无需自动伸缩的服务之外，Kubernetes 默认水平自动伸缩器对于其他服务的伸缩稍显滞后，但也能够随着流量的增加而增加服务实例，随着副本数减少而减少服务实例。

5. 总结

面对用户流量负载的动态性和多服务应用不同服务之间调用关系的复杂性，本文提出了一个针对多服务应用的自动性能分析方法，能够将多服务应用的 SLA 均衡地分解为应用中不同服务的响应时间 SLO，最大程度地避免瓶颈和瓶颈转移现象；利用服务网格提供的镜像流量机制，能够无侵入地、精准地测试出每个服务的最大负载能力，获得单个服务实例最大每秒请求数和资源利用率，尽可能地使得应用在不违反 SLA 的条件下提高资源利用率。

基金项目

中国铁建股份有限公司科技重大专项(2020-A01)。

参考文献

[1] Jiang, D.J., Pierre, G. and Chi, C.H. (2010) Autonomous Resource Provisioning for Multi-Service Web Applications. In: *Proceedings of the* 19*th International Conference on World Wide Web*, Association for Computing Machinery, New York, 471-480. https://doi.org/10.1145/1772690.1772739

[2] Yang, Z., Nguyen, P., Jin, H.M., *et al*. (2019) Miras: Model-Based Reinforcement Learning for Microservice Resource Allocation over Scientific Workflows. 2019 *IEEE* 39*th International Conference on Distributed Computing Systems* (*ICDCS*), Dallas, 7-10 July 2019, 122-132. https://doi.org/10.1109/ICDCS.2019.00021

[3] Fernandez, H., Pierre, G. and Kielmann, T. (2014) Autoscaling Web Applications in Heterogeneous Cloud Infrastructures. 2014 *IEEE International Conference on Cloud Engineering*, Boston, 11-14 March 2014, 195-204. https://doi.org/10.1109/IC2E.2014.25

[4] Roy, S., König, A.C., Dvorkin, I., *et al*. (2015) Perfaugur: Robust Diagnostics for Performance Anomalies in Cloud Services. 2015 *IEEE* 31*st International Conference on Data Engineering*, Seoul, 13-17 April 2015, 1167-1178. https://doi.org/10.1109/ICDE.2015.7113365

[5] Gergin, I., Simmons, B. and Litoiu, M. (2014) A Decentralized Autonomic Architecture for Performance Control in the Cloud. 2014 *IEEE International Conference on Cloud Engineering. IEEE*, Boston, 11-14 March 2014, 574-579. https://doi.org/10.1109/IC2E.2014.75

[6] Chok, N.S. (2010) Pearson's Versus Spearman's and Kendall's Correlation Coefficients for Continuous Data. Doctoral Dissertation, University of Pittsburgh, Pittsburgh.

[7] Luo, S.T., Xu, H.L., Lu, C.Z., *et al*. (2021) Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In: *Proceedings of the ACM Symposium on Cloud Computing*, Association for Computing Machinery, New York, 412-426. https://doi.org/10.1145/3472883.3487003