SHOWAR: Right-Sizing And Efficient Scheduling of Microservices

摘要

微服务架构已被广泛用于设计分布式云应用程序，其中应用程序被解耦为多个小组件（即“微服务”）。部署微服务的难点之一是为每个微服务找到最佳的资源配置量（即大小）和实例数（即副本），能够在保证良好的性能下，避免资源的浪费。本文介绍了 SHOWAR，这是一个通过确定每个微服务的副本数量（水平扩展）和 CPU 和内存数量（垂直扩展）来配置资源的框架。对于垂直扩展，SHOWAR 使用历史资源使用的经验方差来找到最佳规模并减少资源浪费。对于水平缩放，SHOWAR 同样使用了历史资源来计算下一步需要配置的 Replica 数量。此外，一旦找到每个微服务的大小，SHOWAR 就通过为调度器生成亲和规则（即提示）来弥合最佳资源分配和调度之间的差距，以进一步提高性能。我们使用各种微服务应用程序和真实世界的工作负载进行的实验表明，与最先进的自动缩放和调度系统相比，SHOWAR 平均将资源分配提高了 22%，同时将第 99% 的端到端用户请求延迟提高了 20%。

关键词：自动缩放、微服务、调度亲和性、云计算

1 引言

微服务架构是最近一种越来越流行的设计交互式和面向用户服务的范式，其中数百个小而细粒度的组件（即“微服务”）共同致力于在分布式环境中为最终用户请求提供服务。将应用程序分解为小型微服务带来了几个好处。首先，允许不同的开发团队独立处理（可能在技术上）不同的微服务。此外，每个微服务都可以根据其自身的状态和传入的工作负载进行独立扩展和配置，从而提高整个应用程序的性能和可靠性 [13]。最后，微服务架构可以将故障缩小到某一个更细粒度的微服务，在保障整体服务不受影响的状态下，进行小范围的修复。

部署微服务的一个重要挑战是集群资源的分配和扩展。为了能够保证服务的性能，在部署微服务之前，开发人员（应用程序所有者）必须根据 CPU 和内存等计算资源指定每个微服务的大小（即“垂直大小”），以及每个微服务的实例（或“副本”）数量（即“水平规模”）然而，微服务的资源需求将取决于潜在的复杂需求过程，并且可能难以先验预测。一方面，分配比微服务正常运行所需的计算资源更多的计算资源会导致集群资源利用率低，造成资源浪费。另一方面，分配的资源少于微服务所需的资源可能会导致性能下降和服务不可用，造成重大的经济损失。在本文中，我们介绍了 SHOWAR，这是一个为 Kubernetes [8] 管理的微服务的水平和垂直自动缩放而设计的系统。对于垂直自动缩放，SHOWAR 利用历史资源使用情况

的差异来找到每个微服务的最佳资源大小，以保持良好的性能，同时避免低资源利用率。对于水平自动缩放，SHOWAR 使用来自 Linux 内核线程调度程序队列的指标作为其自动缩放信号，以做出更准确、更有意义的自动缩放决策。SHOWAR 的核心使用控制理论的基本思想是根据微服务运行时的信号控制每个微服务的副本数量。此外，还设计了一个比例-积分-微分（PID）控制器 [6] 作为一个有状态的自动缩放器，它使用历史自动缩放动作和当前运行时间测量来做出下一个水平自动缩放决策，并保持微服务的 “稳定”。同时，为了防止资源争用并管理对微服务性能的噪声邻居影响，SHOWAR 使用不同微服务之间历史资源使用的估计相关性来为 Kubernetes 调度器生成规则。本文将 SHOWAR 的与默认的垂直伸缩和水平伸缩进行了比较。使用真实世界的生产工作负载，维基百科的 Access 数据集作为固定的负载。我们的结果表明，SHOWAR 在有效的资源分配和端到端请求延迟的尾部分布方面优于这些基线。最后实验证明，SHOWER 框架能够在保证性能的情况下，大大减少资源的浪费。

2 相关工作

随着云计算的发展，因为云计算的方便性，可以扩展性高，越来越多的业务部署在云上，从而也衍生出了新的业务开发模式——微服务。从单机应用程序向数百个松散耦合的微服务的图形进行重大转变。微服务从根本上改变了当前云系统设计的许多范式，并在优化服务质量（QoS）和利用率时带来了机遇和挑战。

从将整个应用程序功能包含在单个二进制文件中的复杂单机服务，到具有数十或数百个松散耦合的微服务的架构中，微服务的理念模式已经成为大型应用开发的范式。亚马逊、推特、奈飞、苹果等大型云提供商已经采用了微服务应用模式，截至 2016 年底，Netflex 报告出其生态系统中有 200 多个微服务。微服务越来越受欢迎有几个原因。首先，它们促进了可组合的软件设计，简化并加速了开发，每个微服务都负责应用程序功能的一小部分；其次，微服务实现了编程语言和框架的异构，每一层都使用最合适的语言开发，只需要一个通用的 API 来让微服务彼此通信; 最后，微服务简化了 bug 修复模式和性能调试，因为 bug 可以在特定的层中隔离，而不像单体服务那样，解决 bug 通常需要对整个服务进行故障排除。

微服务的提出，大大增加了大规模服务集群的服务可用性和稳定性，而面对大量的微服务，如何管理微服务，对这些微服务的部署，通信等进行编排，目前业界普遍使用的是 docker [5] 或者 Kubernetes [8] 等容器管理和编排工具。微服务的管理主要包括服务扩缩容和调度两面方面，而服务调度主要考虑的是如何在有限的资源下安排微服务在不同的节点上进行部署，使得集群更加稳定，资源利用率更高，服务有更加健壮的生命。微服务的自动伸缩，已经在公共云资源 [2] [3] [4] [11] 的背景下研究了自动缩放，包括不同类型的工作负载 [1] [12] 和微服务。行业从业者广泛使用的微服务自动缩放框架是 Kubernetes [9] 和 Google Autopilot 自动缩放器 [7] 的框架。对于垂直自动缩放，Kubernetes 垂直自动缩放器使用 CPU 和内存的历史资源使用量。然后，它将微服务在下一个时间窗口的资源（CPU 和内存）限制设置为过去时间资源的 90% 分位数 x115%。额外的 15% 用作保证服务可用的资源，以避免为微服务提供的资源不足。谷歌自动驾驶仪的垂直自动缩放器也采用了同样的方法，只是稍作修改，对于内存资源，它使用的不是 P90，而是使用最后一个采样作为下一个窗口时间的限制，但对于 CPU 资源，它使用 P95×1.15（而不是 P90）作为下一个时间窗口。对于水平自动缩放，Kubernetes 和 Google Autopilot 都采用了相同的方法，如图 1 所示，Rm 为计算的副本，其中 T 为用户

$$R_M = \frac{\sum_r \text{P95}_r}{T^*},\ r \in \{M\text{'s current replicas}\}$$

图 1. 副本计算公式

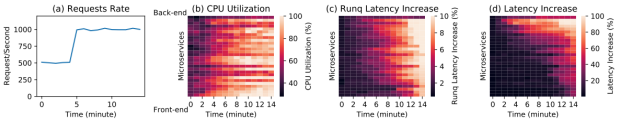


图 2. Enter Caption

设置的目标资源使用率，而 P95 为某个副本的资源使用率，从而计算出每个副本的资源使用率是否超出或者低于目标使用率，来进行水平扩缩容。作为滑动窗口方法的另一种替代方案，谷歌自动驾驶垂直自动缩放仪也为其垂直自动缩放器采用了一套机器学习模型。这些模型试图优化自动缩放操作的成本函数（例如，供应不足和过度供应），并选择成本最低的模型对每个微服务执行垂直自动缩放操作。在 SHOWAR 的水平自动缩放器设计中，我们采用了一种完全不同的方法，我们没有使用 CPU 利用率作为自动缩放指标，而是使用服务的响应延迟时间加上服务成功率来设计一个更准确、更有状态的自动缩放器。使用服务的延迟时间能够更快地完成水平缩放，使得服务的质量能够快速得到保证。

3 本文方法

3.1 本文方法概述

部署微服务的挑战之一是为每个微服务找到最佳的资源数量（即大小）和实例数量（即副本），以保持良好的性能，并防止资源浪费和利用不足本文提出了一个框架，用来最优配置每个微服务的资源数量 (vertical scaling)，以及副本数量（horizontal scaling)。垂直伸缩利用历史数据的经验方差找到一个最优数量。水平伸缩使用服务质量作为依据，使得水平扩缩容能够快速响应扩缩容，使得服务质量得到保障。同时，在确定资源分配模式后，会利用亲和性和反亲和性来确定调度行为，利用不同服务之间的指标的相关性，来确定反亲和性规则，使得具有强相关性的资源能够调度到不同的节点上，避免资源的竞争，导致服务质量下降。

3.2 垂直伸缩

k8s 的默认垂直伸缩策略：通过检测过去一个窗口内的历史数据，通过计算 Π*(1+) ，来计算并设置下一个窗口的资源。但是这种方法存在一个缺点，因为 a 需要手动设置，如果 a 过小，可能会导致频繁的伸缩，如果 a 过大，会导致资源资源浪费。本文提出一个” three-sigma” 法则：$s = u + 3\delta$ ，如果窗口内的值，发生了比较大的变化，例如超过 15%，则重新分配资源。由于考虑了方差的变化，只有在资源变化巨大的时候，才重新进行资源分配，避免资源的过度分配而浪费资源。

3.3 水平伸缩

默认水平伸缩是通过检测资源的实际使用率，来决策部署的 replica 的数量，然而使用 cpu 作为水平调度依据，并不能直接反应出资源的劣化情况。如图 2 所示，在服务请求量增大的时候，子图 2 是 cpu 使用率的变化情况，而子图 3 是服务的请求延迟情况，可以看到，随着请求的增加，cpu 使用率并不是随着线性增加的，所以本文通过使用服务的请求延迟作为水平扩缩容的指标，使用服务延迟，能够真实反映出当前微服务的延迟情况，除此之外，本文还会计算服务的请求成功率，如果请求成功率下降到一定程度，也会自动触发扩容，保证服务的可用性。

3.4 联合使用水平和垂直伸缩

kubernetes 的默认伸缩规则是不允许同时使用水平伸缩和垂直伸缩的。然而本文提出的 SHOWAR 框架能够通过以下规则，来联合使用水平伸缩和垂直伸缩：

规则 1：垂直伸缩决策优于任何水平伸缩决策，因为如果微服务的最小内存资源无法正确设置，会导致微服务频繁 Over Of Memory Error，这种错误会直接导致服务不可用，即使设置再多的副本，也无法解决 OOM 错误，所以任务时候都是水平伸缩优于垂直伸缩。

规则 2: 如果垂直自动缩放器的决策是为 Pod 设置多个内核，则它会通过共享通道向水平自动缩放器发送信号，并且不会继续执行垂直自动缩放操作。根据谷歌云平台的 Kubernetes 最佳实践，由于缺乏并行性，建议对于大多数工作负载，不超过一个核心。我们使用这一建议，并将其纳入 SHOWAR 的垂直自动缩放器设计中。

3.5 调度

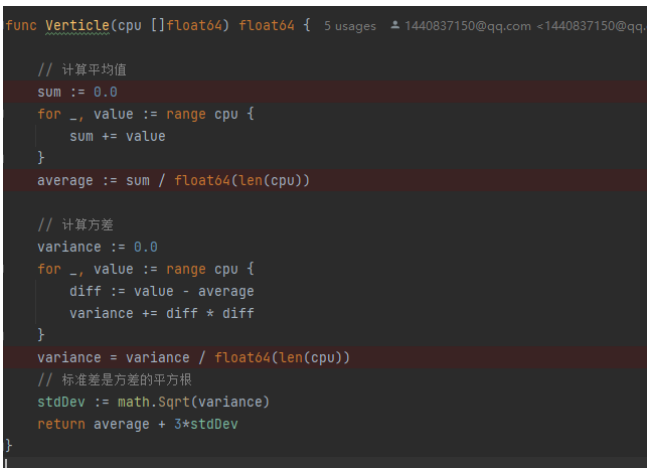
通过计算历史资源（过去一个时间窗口内的 cpu，mem 和带宽的使用率）的皮尔森系数资源（比如 CPU 或内存）使用模式的正相关性越高，它们之间对该资源的资源争用就越高。所以会给对应的微服务生成相应的反亲和性规则，让其部署到不同的节点上，减少资源竞争的情况。

4 复现细节

4.1 垂直伸缩

垂直伸缩控制器，会启动一个无限循环的线程，每分钟读取一次资源，更新 pod 的请求值。首先，通过 prometheus 读取过去 5 分钟内 cpu 和内存资源，其次，通过公式 $u + 3\delta$ 来计算下一个窗口的 request 资源和 Limit 资源设置，其中 u 为资源窗口内的平均值，而 δ 为方差。如果 cpu 计算结果超过 1m，直接分配新的 pod，而不更新 pod 的资源。计算 3δ 的代码如图 3 所示，

最后，从 Annotation 中读取上一个窗口的资源设置，如果资源设置超过上个资源窗口资源设置的 15%，或者低于 85%，才启动更新，避免频繁更新。例如图 4 会通过获取存储在 Annotation 上的上一个时间窗口计算出的值和当前窗口内计算出的值进行比较，最后分别判断是否需要进行内存或者 cpu 的更新。



```go
func Verticle(cpu []float64) float64 {  5 usages  1440837150@qq.com <1440837150@qq.
    // 计算平均值
    sum := 0.0
    for _, value := range cpu {
        sum += value
    }
    average := sum / float64(len(cpu))

    // 计算方差
    variance := 0.0
    for _, value := range cpu {
        diff := value - average
        variance += diff * diff
    }
    variance = variance / float64(len(cpu))
    // 标准差是方差的平方根
    stdDev := math.Sqrt(variance)
    return average + 3*stdDev
}
```

图 3. 垂直伸缩计算代码

```go
    // cpu更新
    if verticleCpuCompute > lastCpuVerticle {
        // 判断是否超过15%
        if verticleCpuCompute > lastCpuVerticle*1.15 {...}
    } else if verticleCpuCompute < lastCpuVerticle {
        if verticleCpuCompute < lastCpuVerticle*0.85 {...}
    }
    // 内存更新
    if verticleMemCompute > lastMemoryVerticle {
        // 判断是否超过15%
        if verticleMemCompute > lastMemoryVerticle*1.15 {...}
    } else if verticleMemCompute < lastMemoryVerticle {
        if verticleMemCompute < lastMemoryVerticle*0.85 {...}
    }
}
```

图 4. 伸缩判断

为了考虑联合使用垂直伸缩和水平伸缩，在垂直伸缩时计算出需要扩容 CPU 的时候，会判断 CPU 是否会超过 1 核，如果超过，则直接进行水平伸缩。同时在缩容的时候为了保证计算出的值不会太低，代码中还做了最低的配额的限制。如图 5 所示

4.2 水平伸缩

水平伸缩控制器，会启动一个无限循环的线程，每分钟读取一次资源，更新 replica 的请求值。1. 通过 istio 工具读取过去 60s 窗口内的请求延时；2. 通过预设值的请求延时目标，通过副本计算公式，计算出需要部署的副本个数，如图 6 所示，是水平伸缩的计算代码；

为了避免资源变化的波动性给副本计算带来的影响，参考默认的水平伸缩的解决资源变化波动性解决方案，在缩容的时候，需要确保距离上次扩容超过 5 分钟才会进行缩容；同时，为了保证服务可用性，本文增加了监控请求的成功率，如果当前服务的请求成功率低于 99%，则会扩容一个副本，如图 6 所示，会通过 istio 获取服务的请求成功率，再和预设的 99% 作为比较，如果低于 99% 则增加一个副本。

4.3 调度亲和性生成器

通过 prometheus 读取过去 5 分钟内 deploy 对应 pod 的 cpu 或者 mem，istio 读取带宽。1. 计算 pod 之间不同资源的皮尔森系数 2. 对于 cpu 和 network，如果皮尔森系数 <=-0.8 ，生成亲和性规则；对于 memory，如果皮尔森系数 >= 0.8，则生成反亲和性规则。如图 7 所

图 5. 限制

图 6. 水平伸缩计算代码

图 7. 请求成功率保证

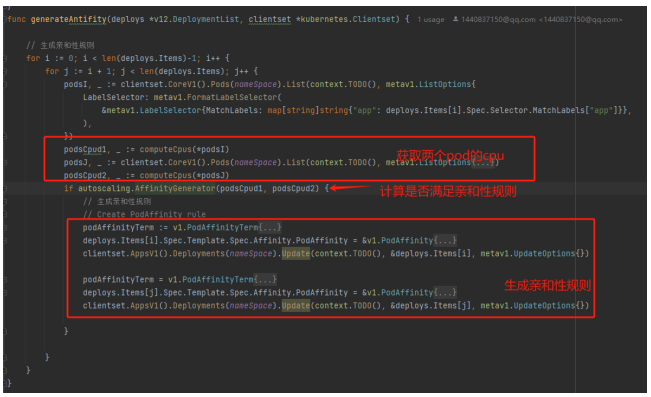


图 8. 调度亲和性

示，通过两个循环来计算两个不同的 pod 之间的过去一个时间窗口内的 cpu 的皮尔森系数，如果皮尔森系数 <=0.8 则生成亲和性规则。

4.4 实验条件

1. k8s 集群：1 个主节点，3 个从节点
2. 资源配置：cpu：64 核心；内存：128GiB
3. 实验数据集：论文里面使用的维基百科的 access 数据集。

5 实验结果分析

5.1 数据集分析

维基百科一个月数据集图像，以及某一天的数据集的请求量展示如图-月数据集以及图-天数据集所示

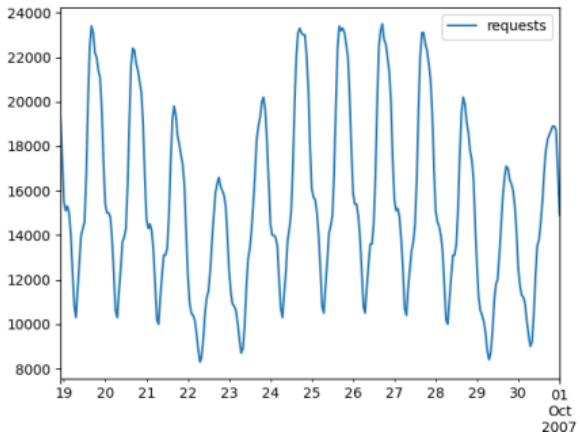


图 9. 月数据集

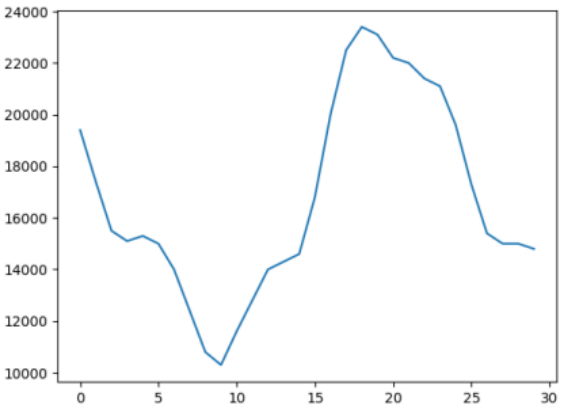


图 10. 天数据集

5.2 满资源运行情况

如图 5 展示了，在没有资源限制的情况下，负载发送的测试的请求发送，发送成功数量，以及返回 200 状态码的请求量。

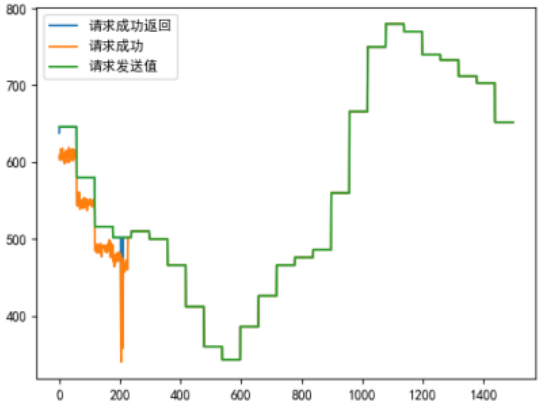


图 11. Enter Caption

5.3 默认垂直伸缩运行情况

如图 6 展示了，在默认垂直伸缩的情况下，负载发送的测试的请求发送，发送成功数量，以及返回 200 状态码的请求量。

5.4 默认水平伸缩运行情况

如图 7 展示了，在默认水平伸缩的情况下，负载发送的测试的请求发送，发送成功数量，以及返回 200 状态码的请求量。

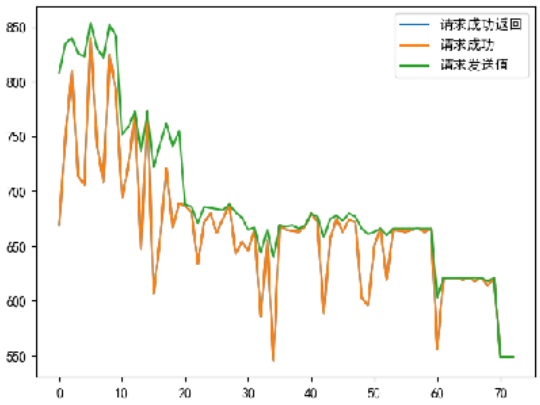


图 12. Enter Caption

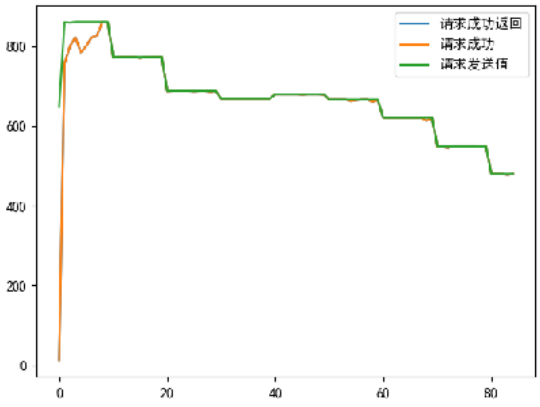


图 13. Enter Caption

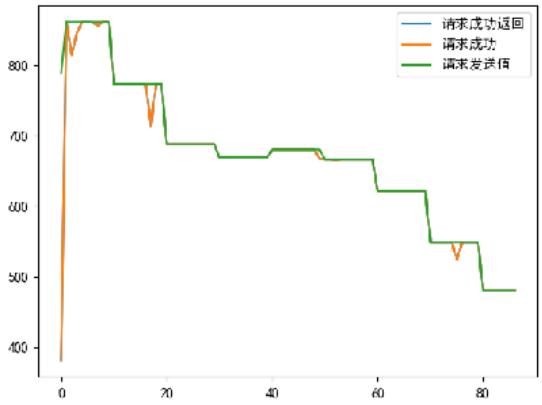


图 14. Enter Caption

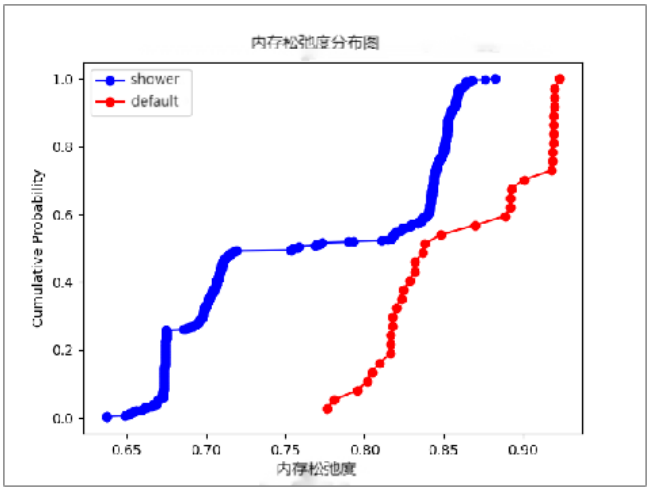


图 15. Enter Caption

5.5 SHOWER 伸缩运行情况

如图 8 展示了，在 SHOWER 的情况下，负载发送的测试的请求发送，发送成功数量，以及返回 200 状态码的请求量。

5.6 性能分析

如图 9 展示了在默认垂直伸缩以及 SHOWER 的对比中，相比默认垂直伸缩，SHOWER 的内存松弛度更小，这意味这在运行过程中，SHOWER 会给微服务内存的 overproviding 更少，避免资源浪费。在图 10 中展示了相比默认水平伸缩,SHOWER 的副本个数更少，默认水平伸缩的副本大多数时候已经超过了 70 个，然而水平伸缩大多数时候分布在 40 个左右，在副本变化较小的时候，能够减少频繁创建副本的开销，使得系统更加稳定。如图 11 所示，SHOWER 与默认的垂直伸缩和水平伸缩做了对比，SHOWER 的延迟相比水平伸缩和垂直伸缩，有了明显的改善，集中分布在 200ms 附近，能够大大提升服务质量。

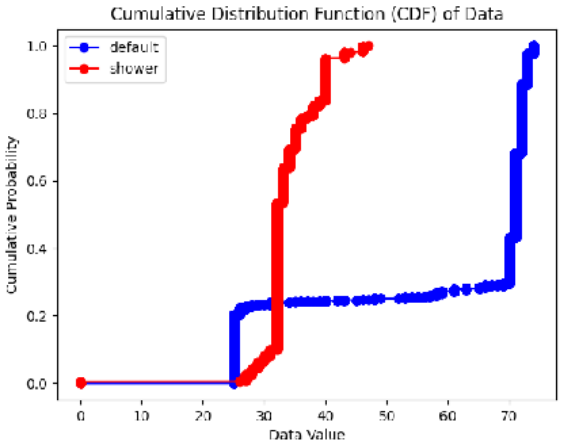


图 16. Enter Caption

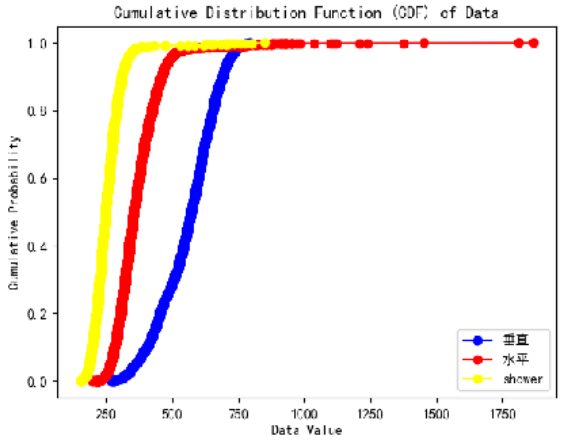


图 17. Enter Caption

6 总结与展望

总之，我们提出了 SHOWAR 框架，该框架由垂直自动缩放器、水平自动缩放器和微服务的调度亲和规则生成器组成。SHOWAR 的垂直自动缩放器包含了历史资源使用量的经验方差，以找到最佳大小并减少资源使用松弛（分配的资源和实际资源使用量之间的差异）。对于水平缩放，SHOWAR 使用控制理论的思想以及内核级性能指标（即 eBPF 运行延迟）来执行准确的水平缩放操作。特别是使用比例-积分-微分控制器（PID 控制器）作为有状态控制器来控制每个微服务的副本数量。SHOWAR 中的垂直和水平自动缩放器协同工作，在保持良好性能的同时提高资源利用率。此外，一旦找到每个微服务的大小，SHOWAR 就会通过为任务调度器生成亲和提示来进一步提高性能，从而弥合最佳资源分配和调度之间的差距。我们使用各种微服务应用程序和真实世界的工作负载进行的经验实验表明，与最先进的自动缩放系统相比，SHOWAR 平均将资源分配提高了 22%（从而节省了成本），同时将第 99% 的端到端用户请求延迟提高了 20%。

然而，SHOWAR 目前的一个主要限制是它对微服务的资源使用是被动的。因此，一个适当的探索途径是为 SHOWAR 提供近期工作负载和资源使用预测，例如 [10]。结合其当前的设计，预测不久的将来的工作负载可以改善 SHOWAR 的资源分配，并防止由于不适当的自动缩放操作而导致的性能下降。

参考文献

[1] Timothy Zhu Ataollah Fatahi Baarzi and Bhuvan Urgaonkar. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. *In Proceedings ofthe ACMSymposium on Cloud Computing*, 2019.

[2] Docker. Autoscaling in amazon web sevices cloud. https://aws.amazon.com/autoscaling/. April, 11, 2023.

[3] Docker. Autoscaling in google cloud platform. https://cloud.google.com/compute/docs/load-balancing-and-autoscaling. April, 11, 2023.

[4] Docker. Autoscaling in microsoft azure cloud. https://azure.microsoft.com/en-us/features/autoscale/. April, 11, 2023.

[5] Docker. Docker. https://docs.docker.com/engine/swarm/. April, 11, 2023.

[6] J.A. Hellerstein et al. Feedback control for computing systems. *IEEE Press / Wiley*, 2004.

[7] J. Swiderski P. Zych P. Broniek J. Kusmierek P. Nowak B. Strack P. Witusowski S. Hand K. Rzadca, P. Findeisen and J. Wilkes. Autopilot: Workload autoscaling at google. *In Proc. ACMEuroSys*, 2020.

[8] Kubernetes. Kubernetes. https://kubernetes.io/. April, 11, 2023.

[9] Kubernetes. Kubernetes autoscalers. https://github.com/kubernetes/autoscaler. April, 11, 2023.

[10] Netflix. Netflix' s predictive auto scaling engine. https://netflixtechblog.com/scryer-netflixs-predictive-auto-scaling-engine-a3f8fc922270. April, 11, 2023.

[11] Jose Miguel-Alonso Tania Lorido-Botran and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal ofgrid computing*, 2014.

[12] Moritz Hoffmann Desislava Dimitrova Matthew Forshaw Vasiliki Kalavri, John Liagouris and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. *In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[13] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. page 149–161, 2018.