Practical Efficient Microservice Autoscaling with QoS Assurance

摘要

云应用程序越来越多地从单一服务转向基于敏捷微服务的部署。然而，由于松散耦合和交互组件的数量庞大，微服务的有效资源管理构成了一个重大障碍。各种微服务之间的相互依赖性使得现有的云资源自动扩展技术无效。与此同时，基于机器学习（ML）的方法试图捕捉微服务中的复杂关系，需要大量的训练数据，并导致故意违反 SLO。此外，这些 ML 繁重的方法在适应动态变化的微服务操作环境方面很慢。在本文中，我们提出了 PEMA（Practical Efficient Microservice Autoscaling），这是一个轻量级的微服务资源管理器，通过机会性资源减少来找到有效的资源分配。PEMA 的轻量级设计实现了新颖的工作负载感知和自适应资源管理。使用原型微服务实现，我们表明，PEMA 可以找到有效的资源分配，并相比基于商业规则的资源分配，节省高达 33% 的资源。

关键词：自动缩放；微服务；资源管理；云计算；服务质量

1 引言

微服务架构在面向用户的云应用程序中的渗透率越来越高，在这些应用程序中，松散耦合的小型服务组件（即，微服务）一起工作以服务用户请求 [4]。如图 1 所示，微服务架构与传统的具有几个大型应用程序层的单体部署有很大不同，例如面向用户前端、后端业务逻辑和数据库 [7]。与单体应用程序不同，小型微服务可以由小型专门的 DevOps 团队轻松管理和更新。此外，微服务通常是无状态的，并使用轻量级 API 进行通信 [2,3]。因此，它们提供了敏捷的资源管理和扩展，更好的容错能力，以及不同微服务之间的平台无关兼容性，这些都是单片应用程序无法匹配的 [10]。

微服务有其自身的挑战，在本文中，我们将重点关注其资源管理。原则上，微服务资源管理与单片应用程序相同-实现所需的性能（例如，端到端响应延迟）与最小资源分配 [1]。然而，基于微服务的应用程序的资源管理更具挑战性，因为这些应用程序由于负责应用程序性能的微服务数量庞大而具有更大的配置空间。例如，如果我们考虑一个具有 m 个微服务的应用程序，其中每个微服务可以配置 n 个不同的 CPU 分配，那么将有 n^m 个可能的资源配置。此外，微服务具有复杂的通信拓扑和相互依赖性，这使得识别和减轻服务质量（QoS）违规变得更加困难 [4]。单个用户请求可能会遍历多个微服务，如果关键路径中的任何微服务成为瓶颈，端到端响应时间将显著增加。我们在原型微服务上的激励性实验表明，相同数量的 CPU 分配可能导致应用程序延迟增加 250% 以上，具体取决于资源在不同微服务之间的分布方式。

与此同时，为具有几个服务层的单片应用开发的现有资源管理技术无法轻松捕获复杂的微服务交互，以做出有效的资源分配选择。然而，解决微服务的这些资源管理挑战至关重要，因为越来越多的生产云服务已经采用微服务架构 [9]。

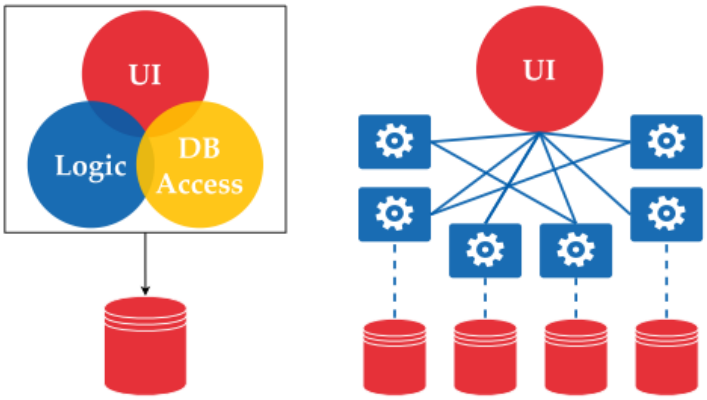

图 1. 微服务架构图

2 相关工作

由于越来越多的兴趣，最近的一些工作试图解决微服务中的资源管理挑战 [9]。他们专注于利用机器学习（ML）技术来捕捉微服务资源和性能之间的复杂关系。例如，FIRM [8] 使用支持向量机（SVM）和强化学习的组合来定位 SLO 违规的根本原因，并应用资源自动缩放来避免这些违规。另一方面，Sage [5] 使用监督训练来使用因果贝叶斯网络识别不同微服务之间的依赖关系，并使用图形编码器来跟踪违反 QoS 的微服务以调整其资源。

然而，这种基于 ML 的工作从根本上受到其广泛培训要求的限制，无论是在捕获微服务动态的培训时间还是数据解析方面（例如，请求级跟踪以构建依赖关系图）。更重要的是，为了从数据中学习，一些基于 ML 的技术故意导致或允许 SLO 违规，这在生产系统中是不受欢迎的 [6]。此外，微服务架构和相互依赖性的任何变化都需要重新训练系统。这种 ML 再培训可能会成为真实的微服务应用程序的障碍，这些应用程序需要频繁进行软件/代码更新。ML 再培训也可以由底层云硬件的变化触发，这些变化是由于服务器迁移和升级引起的。另一方面，微服务的资源需求每天都随着工作负载而变化。然而，专注于 SLO 违规的现有方法并不直接将动态工作负载纳入其学习中。

3 本文方法

3.1 本文方法概述

为了避免上述方法的障碍，我们提出了 PEMA（实用高效微服务自动缩放），这是一种轻量级的微服务资源管理器，不需要大量的培训。PEMA 利用基于迭代反馈的调整来找到满足 SLO 的有效资源分配。PEMA 并没有找到最佳的资源配置，而是首先将大量资源分配给所有微服务以满足 SLO，然后尝试利用资源减少的机会。为微服务分配丰富的资源可以很容易地完成，因为云原生应用程序享有很大程度的资源可扩展性。初始（和低效）资源分配可以使

用现有的基于规则的资源管理器来实现。使用这种机会主义的资源减少方法，PEMA 避免了造成故意的 SLO 违规，因为它总是为微服务分配足够的资源，即使在性能不佳时（即，减少资源浪费的机会）。为了实现 PEMA 的方法，我们引入了“单调资源减少”的概念，即我们要么减少微服务的资源，要么保持不变。相比之下，可以通过针对一些微服务的资源减少和针对一些其他微服务的资源增加来进行非单调资源减少，其中总体总资源减少（即，总减少量大于总增加量）。我们观察到，单调的资源减少导致响应时间的单调增加。因此，我们可以使用响应时间作为反馈，以识别资源减少机会，从而进行渐进单调的资源变化，以达到有效的分配。此外，基于对原型微服务实现的实验，我们发现，我们可以只使用两个微服务级别的性能指标- CPU 利用率和 CPU 节流时间来避免瓶颈服务中的资源减少。

3.2 减少资源的机会

在 PEMA 中，类似于梯度下降的方法，我们首先为所有微服务分配足够的资源，然后根据资源变化如何影响端到端响应时间，逐渐减少资源。我们根据前一时间段观察到的响应时间，定期更新资源分配。由于我们依赖于响应时间统计数据，我们设置了足够长的更新间隔，以获得稳定的响应时间统计数据。例如，在 SockShop 中，我们使用的更新间隔为两分钟。

在时间步 t 进行资源减少时，我们首先决定减少资源的微服务数量 n^t，使用以下公式：

$$n^t = N \cdot \min\left(\frac{R - r^{t-1}}{\alpha R}, 1\right) \tag{1}$$

其中，$r^{t-1} = \mathcal{F}(x^{t-1})$ 是前一时间步的响应时间。$\alpha \leq 1$ 是一个用户定义的非负参数，决定了我们想要多积极地减少资源。较小的 α 将更积极地减少资源，反之亦然。

接下来，使用与公式 (1) 类似的方法，我们决定在 n^t 微服务中减少资源的百分比，使用以下公式：

$$\Delta^t = \beta \cdot \min\left(\frac{R - r^{t-1}}{\alpha R}, 1\right) \cdot 100\% \tag{2}$$

其中，$\beta \leq 1$ 是另一个用户定义的参数，决定了在一次时间步中任何微服务的最大资源减少量。较高的 β 值会使 PEMA 在更新间隔之间积极地改变资源，反之亦然。

使用公式 (1) 和 (2)，PEMA 动态调整资源减少量，当我们的响应时间 r^t 接近 SLO（服务水平目标）限制 R 时。我们还可以动态设置 α 和 β 的值，当 $R - r^{t-1}$ 较高时进行更积极的减少，并随着 r^t 接近 R 减少每次间隔的减少量。此外，为了避免响应时间的瞬态扰动触发资源变化，我们可以通过降低 R 的百分比（例如，到 95%）来保持响应时间缓冲区。

3.3 避免瓶颈服务

为了决定候选微服务 n^t，我们首先选取那些 CPU 节流时间低于各自阈值的微服务集合。我们将这些微服务的索引集合表示为 $I^t = \{i : h_i^{t-1} \leq H_i^{th}\}$。然后，我们将 I^t 中每个微服务的利用率标准化到它们各自的利用率阈值 U_i^{th}，并更新 I^t 中每个微服务的概率，如下所示：

$$p_i^t = 1 - \frac{u_i^{*t-1} - \min_{i \in I^t}(u_i^{*t-1})}{1 - \min_{i \in I^t}(u_i^{*t-1})} \tag{3}$$

这里，$\min_{i\in I^t}(u_i^{*t-1})$ 表示 I^t 中所有微服务的最小标准化利用率。公式 (3) 表明，如果一个微服务的利用率等于其阈值，即 $u_i^{*t-1}=1$，那么它的概率将为“零”（$p_i^t=0$），而利用率最低的微服务，即 $u_i^{*t-1}=\min_{i\in I^t}(u_i^{*t-1})$，将有“一”的概率（$p_i^t=1$）。我们为第 i 个微服务创建一个新的候选集合 I^{*t}，并为其分配包含概率 p_i^t。如果 I^{*t} 的大小等于或小于 n^t，我们取整个 I^{*t} 集合，并减少 I^{*t} 中每个微服务的资源 Δ^t。然而，如果 I^{*t} 的大小大于 n^t，我们则从 I^{*t} 中均匀随机选择 n^t 个微服务。

3.4 迭代优化

PEMA 通过迭代方式应用资源减少，并将所有的资源分配 x^t 和响应时间 r^t 保存在“资源分配历史数据库（RHDb）”中。RHDb 的目的是允许 PEMA 在所有微服务的 SLO（服务水平目标）违规情况下，回滚到之前的满足 SLO 的资源分配。即使在延迟接近 SLO 时资源减少会减慢，PEMA 也不能保证其机会性资源减少永远不会导致 SLO 违规。此外，微服务实现的变化或其硬件配置的变化也可能改变最优资源分配并导致 SLO 违规。在这种情况下，回滚到先前的配置允许 PEMA 重新开始寻找新的最优解，而不是将资源分配重置为最大值并从头开始。虽然 RHDb 本身由于其轻量级的单表实现并不增加显著的开销，但回滚操作可能会导致 PEMA 寻找有效资源分配的额外迭代。尽管如此，使用 RHDb 进行回滚的机制对于 PEMA 的适应性和 QoS（服务质量）保证至关重要。

3.5 逃离次优配置

PEMA 通过单调资源减少和概率性选择微服务来减少资源，这可能导致 PEMA 在早期做出不利的资源减少决策（例如，使特定的微服务达到瓶颈，并将响应时间推向服务水平目标 SLO），并停留在低效的资源分配状态，即使其他微服务有多余的资源。这可能会迫使 PEMA 过早地放慢速度，甚至停止进一步的资源减少。为了摆脱这种低效的资源分配，我们实现了随机探索，其中 PEMA 以概率 p_e^t 回滚到 RHDb（资源分配历史数据库）中的一个随机先前资源分配。我们根据响应延迟设置 p_e^t 如下：

$$p_e^t = A\cdot\min\left(\frac{R-r^{t-1}}{\alpha R},1\right)+B \tag{4}$$

这里，A 和 B 是探索参数，分别决定最大和最小探索概率，并满足 $0\le B\le A\le 1$ 和 $A+B\le 1$。随着 PEMA 的响应时间 r^{t-1} 接近 SLO R，探索概率会降低。随机探索还允许 PEMA “回溯”其采取的资源减少路径，并识别之前错过的减少机会。自然地，探索的程度会影响我们达到有效资源分配的速度。尽管如此，我们预计这种探索不会增加显著的开销，因为 PEMA 可以在几十次迭代中找到有效的资源分配。

4 复现细节

4.1 资源减少

如图 2 所示，为了决定候选微服务，首先根据采集的系统 CPU 资源利用率指标和系统 CPU 节流时间指标，来对每个微服务进行更新，随后根据公式 (1) 来决定候选微服务，如果候选集合的大小等于或小于微服务总数量，我们取整个集合。然而，如果候选集合的大小大

于微服务总数量，我们则从候选集和中均匀随机选择微服务。随后如图 3，4 所示，参考公式 (2)，利用 istio 读取过去 120s 窗口内的响应延迟，与预设的 SLO 进行相减，随后利用预先设好的 α 和 β 计算 Δ^t，最后将每个微服务容器对应的乘以计算后出的比例得出新的 CPU。

```
if latency < threshold:
    # update each container's utilization limit with current utilizations
    for container in system_metrics.keys():  # update limit of containers.
        # max_value = max(system_metrics[container]['cpu_utilization'])
        if system_metrics[container]['cpu_utilization'] > CONTAINER_UTIL_LIMITS[container]:
            CONTAINER_UTIL_LIMITS[container] = system_metrics[container]['cpu_utilization']

    # update threshold of each container with current value
    for x in system_metrics.keys():  # update throttle of containers
        if system_metrics[x]['throttles'] >= CONTAINER_THROTTLE_LIMITS[x]:
            CONTAINER_THROTTLE_LIMITS[x] = system_metrics[x]['throttles']

    # calculate N_S for candidate container numbers
    # select_container_numbers = int((number_of_containers / alpha) * (delta_response / SLO))
    select_container_numbers = int((NUMBER_OF_CONTAINERS / alpha) * (delta_response / threshold))

    # check if # candidate containers (n_s) greater than total container,
    if select_container_numbers > NUMBER_OF_CONTAINERS:
        select_container_numbers = NUMBER_OF_CONTAINERS

    # now choose candidate containers based on n_s, systems metrics (utils, throttles) and current configs
    candidate_containers = choose_containers(select_container_numbers, system_metrics, system_configs)
```

图 2. 挑选候补微服务代码

```
1 usage   rajibhossen
def calculate_delta_si(beta, delta_response, alpha, threshold):
    value = (beta / alpha) * min((delta_response / threshold), alpha)
    return value
```

图 3. 减少比例计算代码

```
new_configs = {}
delta_si = calculate_delta_si(beta, delta_response, alpha, threshold)
if random.uniform( a: 0,  b: 1) < random_exploration:
    for x in candidate_configs:
        new_configs[x] = max(DEFAULT_CONFIGS[x], round((current_configs[x] * (1 + delta_si)), 1))

    for x in current_configs:
        if x not in new_configs:
            new_configs[x] = round(current_configs[x], 1)
else:
    for x in candidate_configs:
        new_configs[x] = round((current_configs[x] * (1 - delta_si)), 1)

    for x in current_configs:
        if x not in new_configs:
            new_configs[x] = round(current_configs[x], 1)
return new_configs, delta_si
```


图 4. 资源减少计算代码

4.2 利用 Prometheus 进行指标采集

如图 5 所示，引入 Prometheus 工具来采集系统指标，利用 istio 來生成每個容器所需作用的指标，例如响应时间，并发请求数等。1. 读取过去 120s 窗口内的 front-end 请求次数；2. 读取过去 120s 窗口内的 front-end 相应延迟。因为用户主要使用前端页面，所以 front-end 的指标是主要参考对象。

```python
def get_response_latency(container_name, duration=60, percentile=0.95):
    duration = str(duration)
    metric_url = f"istio_request_duration_milliseconds_bucket{{destination_workload=\"{container_name}\",reporter=\"destination\"}}"
    query_url = f"{BASE_URL}/api/v1/query?query=histogram_quantile({percentile},sum(rate({metric_url}[{duration}s])) by (le,destination_workload))"
    results = requests.get(url=query_url)
    query_data = results.json()
    if query_data['data']['result']:
        percentile_response = query_data['data']['result'][0]['value'][1]
        return percentile_response
    else:
        return None
```

图 5. 指标采集代码

4.3 引入随机探索

利用公式 (4) 在探索参数的影响下，计算出随机探索概率阈值，如图 6 所示，若是随机的概率没有超过阈值，就会进行随机探索，对当前容器配置进行资源增加，回到之前没有探索的一个配置下重新进行资源减少，最后以该配置作为下一次迭代的新配置。

```python
new_configs = {}
delta_si = calculate_delta_si(beta, delta_response, alpha, threshold)
if random.uniform( a: 0,  b: 1) < random_exploration:
    for x in candidate_configs:
        new_configs[x] = max(DEFAULT_CONFIGS[x], round((current_configs[x] * (1 + delta_si)), 1))

    for x in current_configs:
        if x not in new_configs:
            new_configs[x] = round(current_configs[x], 1)
else:
    for x in candidate_configs:
        new_configs[x] = round((current_configs[x] * (1 - delta_si)), 1)

    for x in current_configs:
        if x not in new_configs:
            new_configs[x] = round(current_configs[x], 1)
return new_configs, delta_si
```

随机探索

图 6. 随机探索代码

4.4 设计 VPA

为了对比 PEMA 的效率，其主要利用对单个容器的资源进行减少来在不影响 SLO 的情况下找到最高利用率的配置。因此参考 Kurbenetes 已有的 VPA 方法，其思想为若是利用率或是自定义指标超过了所设阈值，就会对容器进行资源上的减少，如图 7 所示，对 front-end 进行了 CPU 资源控制，所设最大 CPU 为 5 个单位，若是监控指标未超过阈值，则会进行 CPU 减少，若是超过阈值会对 CPU 进行增加，其最大不会超过所设 CPU 阈值。

```yaml
apiVersion: vpa.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: front-end-vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       "Deployment"
    name:       "front-end"
  updatePolicy:
    updateMode: "Auto"
  resourcePolicy:
    containerPolicies:
      - containerName: 'front-end'
        controlledResources:
          - cpu
        maxAllowed:
          cpu: "5"
        minAllowed:
          cpu: "100m"
```

图 7. VPA 方法

4.5 实验环境搭建

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. 原型微服务：如图 6 所示的 Sock shop 微服务。

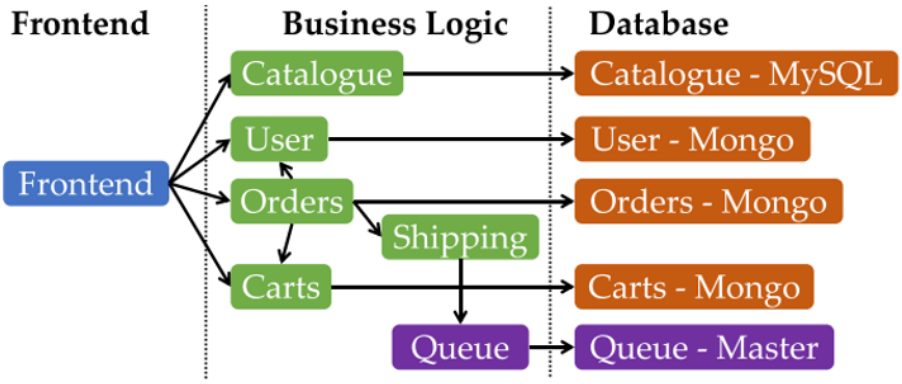

图 8. Sock shop 架构图

5 实验结果分析

5.1 在不同概率下进行配置探索

如图 7 所示，在两种概率下对 CPU 配置进行探索。从图中可以看出，高概率探索时，会回滚到更高的资源分配配置进行探索，随后找到更高效的资源配置。在高概率探索的情况下，经历了几次 SLO 违规，之后又回滚并最终调整到符合 SLO 的配置，而低探索则更稳定地找

到符合 SLO 的配置。

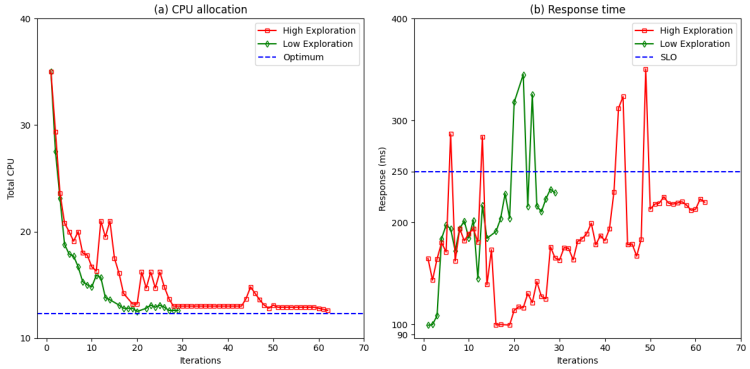

图 9. 高低概率探索配置图

5.2 PEMA 具体配置

如图 8 所示，在 700 并发请求数下 Sock shop 的近最优 CPU 分配。从图中可以看出，carts 微服务分配了最多的 CPU 资源，达到了 5 个单位，并没有进行资源减少，是由于它需要处理大量的用户请求和复杂计算，因此是 Sock shop 的瓶颈微服务。而 catalogue 和 payment 微服务分配的 CPU 资源较少，分别为 0.5 和 0.2 个单位，这意味着这些服务的计算需求较低，能够在较少的资源下高效运行。这种资源分配策略有助于优化整个微服务系统的性能，确保关键服务的响应速度和稳定性，同时对资源需求较低的服务则分配较少的资源，实现资源的合理利用。

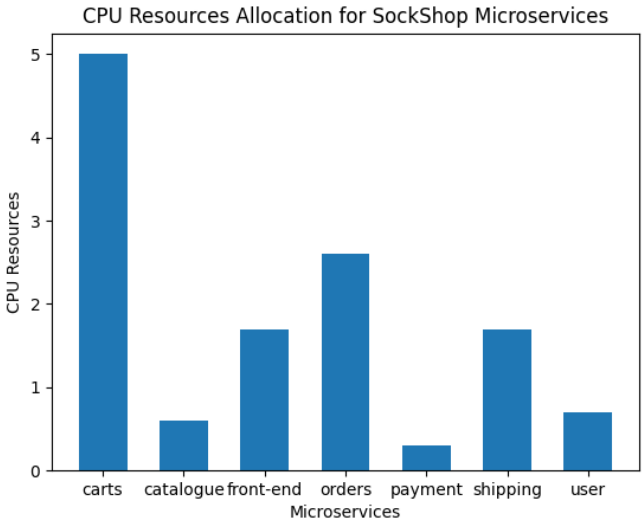

图 10. 700RPS 下 CPU 配置图

5.3 性能分析

我们将 PEMA 的资源配置效率与两个基准策略–最优（OPTM）和基于规则（RULE）进行了比较。在 OPTM 中，我们使用穷举试错搜索来确定最佳的可能资源分配。如果任何微服务中的少量资源减少（在本例中为 0.1 CPU）导致 SLO 违规，则我们将资源分配确定为最佳。

RULE 是 Kubernetes 的基于规则的资源缩放。我们选择 RULE 作为商业可用的资源分配算法来衡量 PEMA 的效率改进。我们使用 OPTM 的资源分配对每个工作负载级别的每个资源分配进行了归一化。如图 9，分别展示出了三种不同算法的 SockShop 的资源分配。我们看到，PEMA 的资源配置效率非常接近于 OPTM。我们还观察到，PEMA 的效率随着工作负载的增加而漂移。另一方面，PEMA 始终优于 RULE，在高工作负载下为 SockShop 节省了多达 33% 的资源分配。

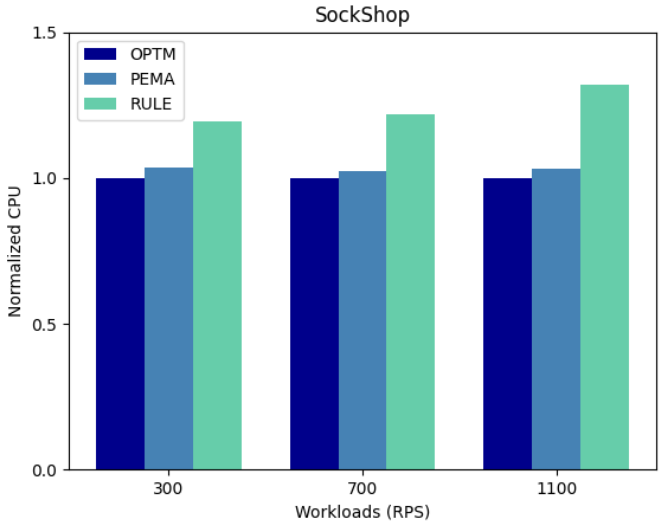

图 11. 性能比较图

6 总结与展望

本文中利用了 PEMA，这是一种基于迭代反馈的微服务自动扩展方法。PEMA 采用轻量级设计，仅需应用端到端性能、微服务级别的 CPU 利用率和 CPU 节流时间，即可实现高效的微服务资源分配。通过原型微服务实现，我们验证了 PEMA 的三个特点：1. 无需大量训练数据 2. 避免 SLO 违规 3. 适应性强，同时也表明 PEMA 能够实现接近最优资源分配的性能，并且与商业上使用的基于规则的资源分配相比，最多可节省 33% 的资源。

PEMA 的实现存在几个局限性，首先，当 PEMA 导致意外的服务级别目标（SLO）违规时，它会在下一个时间步长回滚资源配置。因此，在整个资源更新间隔期间（例如 2 分钟），应用程序会遭受性能不佳的影响。PEMA 可以通过实现更高分辨率的性能监控（例如在 10 秒内）来改进，尽早捕获 SLO 违规，并回滚配置以缓解违规情况。此外，PEMA 会将配置回滚到最近一次没有 SLO 违规的配置，而不会考虑 SLO 违规的程度。例如，如果响应时间显著高于 SLO 的 QoS 违规，则表明 PEMA 应该回滚到更早的配置以分配更多资源。另一方面，虽然 PEMA 会在其分配历史数据库 RHDb 中记录所有微服务的资源分配情况和响应时间，以用于回滚和探索目的，但它在决策过程中并未利用这些信息。最后，PEMA 在本研究中仅考虑 CPU 资源分配，而内存和 I/O 资源分配也可能对微服务的性能很重要，具体取决于应用的性质。此外，PEMA 也没有明确地解决垂直（即，增加一个节点中的资源）和水平（即，增加节点的数量）资源缩放，所以 PEMA 后续可以尝试与 VPA 或 HPA 结合起来，能够明确地完成节点的资源缩放。

参考文献

[1] Azure autoscale. https://azure.microsoft.com/en-us/features/autoscale/. Last Accessed: 01/05/2022.

[2] The definition of microservice. https://martinfowler.com/microservices/, 2022. Accessed: 01/26/2022.

[3] Introduction. https://www.nginx.com/blog/introduction-to-microservices, 2022. Accessed: 2022-01-20.

[4] Y. G. et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS '19*, page 3–18, 2019.

[5] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 135–151, 2021.

[6] Yu Gan, Meghna Pancholi, Dailun Cheng, Siyuan Hu, Yuan He, and Christina Delimitrou. Seer: Leveraging big data to navigate the increasing complexity of cloud debugging, 2018.

[7] Xiaofeng Hou, Chao Li, Jiacheng Liu, Lu Zhang, Shaolei Ren, Jingwen Leng, Quan Chen, and Minyi Guo. Alphar: Learning-powered resource management for irregular, dynamic microservice graph. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 797–806, 2021.

[8] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.

[9] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 167–181, 2021.

[10] 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. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC ' 18. ACM, October 2018.