LEAD: Latency-Efficient Application Deployment for Microservices Architecture

Mohammad-Ali Mirzaei
Amirkabir University of Technology
momirzaei@aut.ac.ir

Parsa Poorsistani
Amirkabir University of Technology
parsa.ps@aut.ac.ir

Seyyed Ahmad Javadi
Amirkabir University of Technology
sjavadi@aut.ac.ir

***Abstract*—In cloud computing, microservices architecture has become the preferred choice for many applications. Accordingly, several small and decoupled containerized services hosted on different servers communicate through the network. Therefore, communication latency significantly impacts end-to-end latency. Kubernetes, the de facto standard for container orchestration, cannot reduce this overhead due to its lack of awareness of service interactions.**

We present LEAD, a Latency-Efficient Application Deployment framework that integrates with Kubernetes without modifying its core components. LEAD considers the inter-service relationships and resource constraints, improving service placement to reduce end-to-end latency. The idea behind LEAD is straightforward: keep the cooperating services close to each other to exploit faster in-node communication and automate this process. The proposed idea is realized by leveraging a scoring algorithm and monitoring framework to achieve dynamic improvement of service placement. Our experimental results show an average 20% improvement in the 99th percentile latency compared to Kubernetes default scheduler.

***Index Terms*—Cloud computing, Microservices architecture, Tail-latency, Resource management, Container placement**

I. Introduction

The microservices architecture is transforming the creation of user-centric services through its dynamic and modular approach. Characterized by a multitude of small, independent microservices working in unison, this architectural style facilitates agile and autonomous development, allowing it to scale and adapt to varying workloads [1], [2], [3]. Moreover, it streamlines troubleshooting and enhances the ability to pinpoint and address performance and reliability concerns [4], [5].

In a microservices architecture, everything flows over the network. A prominent challenge within this architecture lies in the appropriate scheduling of these services. The proximity of interrelated services is crucial for decreasing communication latency and boosting the overall efficiency of service responses. Managing many small service connections and changing network needs make this scheduling task harder [6] [7].

Numerous studies have focused on resource utilization challenges, often evaluating their scheduling solutions using Kubernetes. By default, the Kubernetes scheduler does not consider the dependencies between microservices. Among the recent academic solutions, SHOWAR and SINAN represent two significant but distinct approaches[8], [9]. SINAN uses machine learning to predict and mitigate service-level objective violations, which can be resource-intensive and require extensive data. SHOWAR, on the other hand, uses autoscaling based on historical data to manage resources dynamically. However, neither approach primarily focuses on pod (i.e., smallest unit of a Kubernetes application) placement based on service interactions.

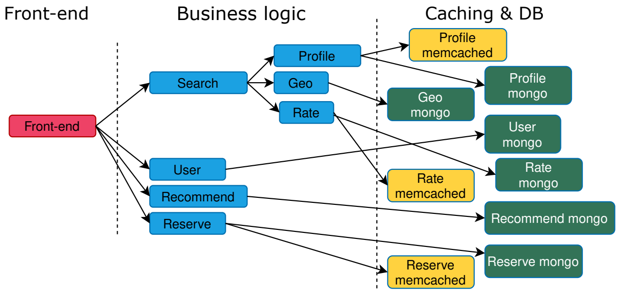

Fig. 1. A visual representation of the microservices employed in LEAD performance tests, focusing on service relationships and data flow [10].

LEAD operates orthogonally to prior approaches, focusing on auto-scaling and resource management within Kubernetes environments. The idea behind LEAD is simple: keeping related and interacting services as close to each other as possible within the necessary constraints and resources. For example, in Figure 1, if the ”Recommend” service communicates frequently with its respective database, it is better to place them on the same node to utilize the loopback mechanism, which is much faster. Kubernetes already has a built-in feature for this purpose—affinity rules—which allows developers to define how the Kubernetes scheduler should act on certain services. However, managing this manually, especially with many services and servers, can be challenging. Moreover, in the culture of DevOps, there is a strong preference for automating everything.

LEAD automates the finding of appropriate placements using three algorithms. The first algorithm identifies critical paths in the microservices architecture graph. LEAD then continuously monitors the services by collecting real-time data from monitoring agents for further deployment by the second algorithm in the event of performance issues, such as tail-latency violations or resource overuse. The third algorithm generates and applies affinity rules to the configuration files of these services. Therefore, the Kubernetes scheduler considers these rules for pod placement of the services. It was all done without modifying Kubernetes' default scheduler.

We evaluate LEAD by deploying an interactive microservice application on a cluster of virtual machines on a public cloud. We compare the performance of LEAD against that of the Kubernetes default scheduler. Our results of an experimental analysis using real-world production workloads show that LEAD outperforms this baseline regarding the tail distribution of end-to-end request latency. In particular, on average, LEAD improves the 99th percentile end-to-end user request latency by 20%.

In summary, the main contributions of LEAD are:

- Automated placement: LEAD automatically improves service placement, reducing tail latency significantly.
- Dynamic improvement: LEAD monitors application performance in real-time to handle sudden tail-latency violations.

II. Background and Related Work

A. Background

Microservices architecture has become a dominant design choice for modern software applications due to its scalability and flexibility. In this architectural style, applications comprise small, independent units known as microservices, each performing a specific function and communicating over the network. For example, Figure 1 shows the Hotel Reservation application, highlighting its microservice-based structure. Each microservice, encapsulated within its container [11], interacts with others via HTTP API requests or RPC calls [12], [13]. User requests enter through the front end and traverse through various microservices that span across different tiers—front-end, business logic, and data storage. This forms a dependency graph where specific microservices rely on others.

Kubernetes is an open-source platform designed to automate application container deployment, scaling, and operation [14]. It groups containers that make up an application into logical units for easy management and discovery—known as pods. However, the Kubernetes default scheduler does not consider service communication and dependency patterns when deploying pods. This oversight can lead to suboptimal pod placement, increasing the end-to-end tail latency of applications, as the network communication between dependent services across distant nodes introduces latency.

One of Kubernetes features to address this challenge is the built-in affinity rules, which developers can use to influence pod scheduling. These rules can be specified as 'preferred' or 'required' [15]. 'Preferred' rules suggest to Kubernetes where it might be best to place pods but do not strictly enforce these suggestions, allowing for flexibility in scheduling. In contrast, 'required' rules are enforced, ensuring pods are placed according to the specified policies, which could be critical for performance or regulatory compliance. Within the preferred rules, developers can assign a weighting from 0 to 100. This weighting allows developers to specify the relative importance of each rule. A weight of 100 indicates that the scheduler should treat the rule as nearly mandatory, while a weight closer to 0 suggests that the rule is more of a preference and less critical.

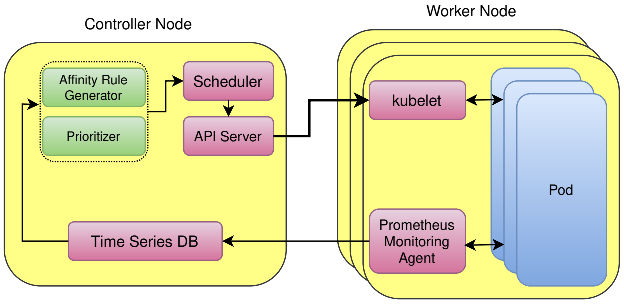

Fig. 2. LEAD architecture overview: Resource usage logs are collected via corresponding agents on each node and saved into a time series database. LEAD then utilizes these collected metrics to schedule affinity rules by interfacing with the Kubernetes scheduler.

B. Related works

Here, we give a brief overview of statstate-of-the-artd to autoscaling and resource management for microservices. Autoscaling has been studied in the context of public cloud resources [16], [17], [18], [19] including different types of workloads [20], [21]. The autoscaling framework for microservices, which is widely used by practitioners in industry, are those of Kubernetes [14] and Google Autopilot autoscaler [22]. Google's *Autopilot autoscaler* and Kubernetes' *Horizontal Pod Autoscaler* (HPA) also play crucial roles in this ecosystem. Both aim to automate the scaling of pods based on observed CPU usage and other metrics but do not consider the service interaction graph complexity. *SHOWAR* [8] and *SINAN* [9] represent two distinct approaches in the current landscape of microservices deployment and management. *SHOWAR* employs empirical variances in historical resource usage for vertical scaling and integrates control theory for horizontal scaling. Its innovative approach extends to generating affinity rules to assist Kubernetes in making informed scheduling decisions, consequently enhancing resource utilization and reducing tail latency.

SINAN, a machine learning-based system, stands out for its predictive capabilities, mitigating potential service level objective violations. While resource-intensive and reliant on extensive data, SINAN does not focus on placing pods using affinity or anti-affinity rules, distinguishing it from other methods.

Our approach distinguishes itself from these existing methods by not only considering the scaling of services but also by emphasizing the improvement of pod placement through an analysis of service interaction graphs. This ensures that related services are preferably co-located to reduce communication delays, thereby enhancing the overall performance of the application.

III. LEAD Design

LEAD is a framework integrated into Kubernetes. It utilizes three main algorithms to analyze service interaction graphs for critical path discovery, considering resource requirements and dynamic performance metrics. We first provide an overview of LEAD and then explain its main components.

A. Overview

In a large-scale environment with numerous servers and a vast array of applications, it becomes crucial to prioritize which pods are preferred for co-location and to establish the level of preference. This prioritization helps efficiently manage the myriad of possible pod placements, significantly impacting the network latency and overall system performance. LEAD architecture, as illustrated in Figure 2, comprises two main components: (i) the Prioritizer and (ii) the Affinity Rule Generator. At the initial deployment, the dependency graph (example shown in Figure 3) detailing the relationships between microservices—which includes the length of each path and the number of incoming vertices— is provided to the Prioritizer. The number of replicas specified by the developers is also provided. Another crucial metric, RPS, is collected during the monitoring phase and is applied to the algorithm. Below, we describe the reason behind each metric:

- Length of each path: This metric helps identify critical paths that may impact overall application performance due to their complexity or length, thus requiring careful placement.
- Number of incoming vertices: These services are often crucial nodes that manage more data or requests.
- Number of replicas: Understanding the required number of service instances ensures the application can handle expected loads during high-traffic periods.
- RPS: Services that form part of a critical path typically have higher RPS values, as these paths handle core business logic or user-facing functionalities.

Using this information, the Prioritizer can identify the critical paths and sort a normalized version, considering a calculated score that resembles the Kubernetes affinity rule weighting mechanism. This score is then applied to the configuration files of each service by the Affinity Rule Generator. It plays a crucial role in the system by applying the calculated weights to the configuration files, which are then handed over to the Kubernetes scheduler. LEAD then continuously monitors the application performance, responding to tail-latency violations or bottlenecks using collected real-time information with the help of third-party monitoring agents like Prometheus.

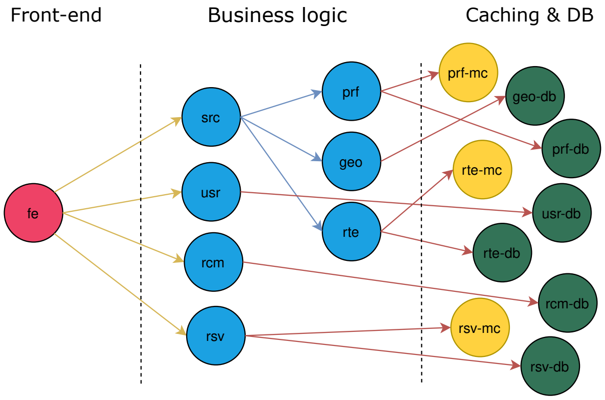

Fig. 3. Graph of relations for Hotel Reservation application

B. Algorithms for critical path detection and monitoring

The heart of LEAD is the Prioritizer and its scoring algorithm.

Algorithm 1 Scoring Algorithm

```
Require: $G = (V, E)$: Directed graph representing the service mesh
Require: apiGateway: The entry point of the service mesh
Ensure: scheduledPaths: List of paths with applied scheduling rules
paths ← findAllPaths($G$, apiGateway)
for each path in paths do
    pathLength ← findPathLength(path)
    podCount ← countPodsInPath(path)
    serviceEdgeCount ← countServiceEdges(path)
    score ← calculateScore(pathLength, podCount, serviceEdgeCount, RPS)
    path.setScore(score)
end for
sortedPathsByScore ← sortByScore(paths)
weight ← 100
for each path in sortedPathsByScore do
    applySchedulingRules(path, weight)
    weight ← weight − 1
end for
return sortedPathsByScore
```

1) Scoring algorithm:

1) Scoring algorithm: This algorithm is tasked with finding the critical paths through the service mesh by identifying routes that reduce latency. Each path is scored based on several factors: the length of the path, the number of pods it includes, the number of service-to-service interactions and RPS. The score effectively reflects the path potential impact on the overall application performance and stability.

By evaluating these aspects, the algorithm prioritizes paths that are most critical for maintaining efficient service operations. The scored paths are then normalized to a scale of 0 to 100 using the formula:

$$s_i' = \frac{s_i - \min(\{s_1, s_2, \ldots, s_n\})}{\max(\{s_1, s_2, \ldots, s_n\}) - \min(\{s_1, s_2, \ldots, s_n\})} \cdot 100$$

This normalization allows for weighting the paths in a manner that resembles the Kubernetes affinity rules' weighting mechanism. The paths are then sorted, with the most critical paths receiving the highest priority in the scheduling algorithm.

This integration of normalization ensures that the prioritization is both meaningful and effective, allowing for dynamic adjustments in scheduling as dictated by real-time service requirements.

2) Performance monitoring algorithm: Another component of the Prioritizer is the performance monitoring algorithm. This algorithm is designed to continuously monitor services in the application and dynamically scale out any service that becomes a bottleneck.

Algorithm 2 Real-time monitoring

```
Require: None
Ensure: None (side effects: scales out services based on
    monitoring data)
while true do
    if detectLatencyViolation() then
        bottleneckService ← analyzeAndIdentifyBottleneck()
        if bottleneckService is not None then
            re-run scoring algorithm
            scaleOut(bottleneckService)
        end if
    end if
end while
function analyzeAndIdentifyBottleneck():
    serviceData = getServiceData() {Get data for all services}
    bottleneckServices = []

    for service in serviceData:
        totalUsage = getResourceUsage()
        if totalUsage > threshold:
            bottleneckService.add(service)
    return bottleneckService
```

At each iteration, the system checks for a latency violation; if a latency violation is detected, the algorithm identifies the service causing the bottleneck, and if the total usage for any service exceeds a set threshold, that service is marked as a bottleneck. If at least one bottleneck service is identified, then the scoring algorithm comes into play again. The real-time performance feedback loop and tail latency-focused monitoring are pivotal to LEAD dynamic scoring algorithm, helping the application respond effectively to performance changes.

C. Affinity rule generator

After prioritizing paths, the "Affinity Rule Generator" component applies Kubernetes' affinity rules in 'Preferred' mode to co-locate interacting services, thereby reducing network latency and improving overall system performance using the algorithm below. The choice of 'Preferred' mode allows for flexibility in pod placement rather than enforcing strict co-location, which could potentially delay or halt the deployment process if specific conditions are not met.

The algorithm requires a path and a weight which indicates the priority of the path. The services within the path are sorted in ascending order. The algorithm iterates over each service in the sorted list, and during each iteration, the algorithm checks if the index is even; if it is, then it generates an affinity rule tailored explicitly to the current service and applies this affinity rule to the current service and its immediate successor in the list. The modulo operation ensures that affinity rules are used only to adjacent pairs of services, such as between the first and second, third and fourth, and so on. This method means that the rule is applied starting with the first service (an even index) and affects every second service after that, effectively omitting services that fall at odd positions in the sequence from initiating these rules.

In summary, LEAD enhances Kubernetes pod scheduling by utilizing a specific set of algorithms that analyze service interaction graphs to uncover critical paths, prioritize them based on the mentioned metrics. By continuously monitoring the application and upon detection of tail-latency violations or resource bottlenecks, LEAD can respond by scaling out or re-prioritizing the paths.

Algorithm 3 Apply Scheduling Rules

```
Require: path: A single path in the service mesh
Require: weight: Weight of priority
Ensure: None (side effects: applies affinity rules to services)

services ← sortedASC(path)
for each service from $i = 0$ to $servicesLength - 1$ do
    if $i$ mod 2 = 0 then
        affinityRule ← generateAffinityRule(service)
        applyAffinityRule(service[$i$], service[$i + 1$], weight)
    end if
end for
```

IV. Evaluation

A. Evaluation Environment

In this paper, we sought to replicate operational scenarios by deploying and executing our proposed algorithms on real-world servers. We utilized five servers located across different geographical locations worldwide to comprehensively test and evaluate our algorithm.

B. Cluster Setup

We conducted all our evaluations on Cloudzy. The virtual machine (VM) instances were deployed in different geographical locations such as New York, Dallas, London, Frankfurt, and Amsterdam. The VM instance in New York served as the master server and had 6 vCPUs and 12 GB of memory. In addition, we provisioned 4 worker VM instances, each with 2 vCPUs and 2 GB of memory. All VMs were running Ubuntu 21.04 TLS.

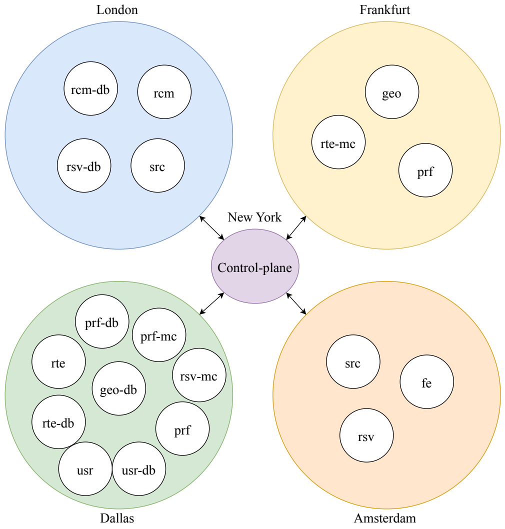

Fig. 4. Kubernetes pod placement

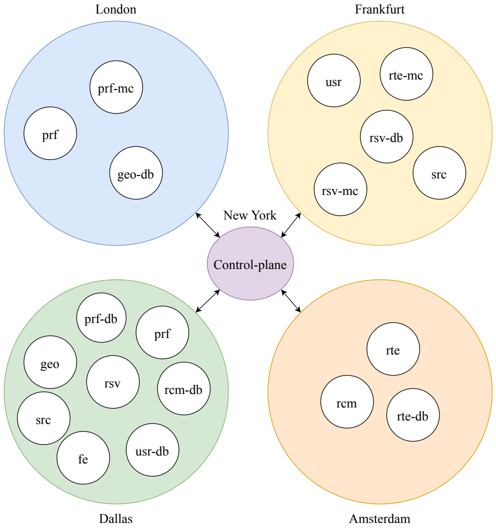

Fig. 5. LEAD pod placement

C. Death-Star Benchmark

For our evaluation, we utilized the Hotel Reservation application shown in Figure 1 comprising 17 microservices. This application encompasses critical processes such as hotel reservation and hotel recommendation, each relying on various services within the application architecture. Provided by DeathStarBench, a comprehensive benchmarking suite designed specifically for assessing microservices. The workload for the Hotel Reservation application encompasses diverse user interactions typical of a hotel booking platform. It includes a mix of read-heavy queries (such as searching for hotels and checking availability) and write-heavy operations (such as making reservations).

DeathStarBench load generator was employed to produce these workloads. It generates requests that mimic actual user behavior, including Poisson arrival times to simulate random traffic and burstiness to test system performance under high load conditions. The generator can distribute the load across multiple client instances, thereby maintaining a realistic environment for the microservices.

1) Baselines: We compare LEAD's performance with Kubernetes' default scheduler. We also compare the performance of the default Kubernetes algorithm in the initial deployment of services on servers with our proposed algorithm. Subsequently, we compare the end-to-end latency in two deployment scenarios based on the default and proposed algorithms. Based on Figure4 and 5, it is evident that using our proposed Kubernetes algorithm, Kubernetes tries to deploy services alongside each other based on their inter-relationships.

As it is shown in Figure 6, in the 8-hour time frame, if proper deployment does not occur, the waiting time for requests may increase, especially in the default deployment scenario, under increased workload. Moreover, as depicted in Figure 7, all requests complete their processes faster when services are deployed according to our proposed algorithm compared to the default scenario. We can understand from the figure that all requests under the LEAD algorithm can complete their business in a smaller time range than default deployment. Although we do not have enough space to add another chart, it is noteworthy that the number of timeout requests decreases in our algorithm. In essence, when services are not under high pressure, they can respond to requests immediately, leading to decreased timeouts and improved system responsiveness.

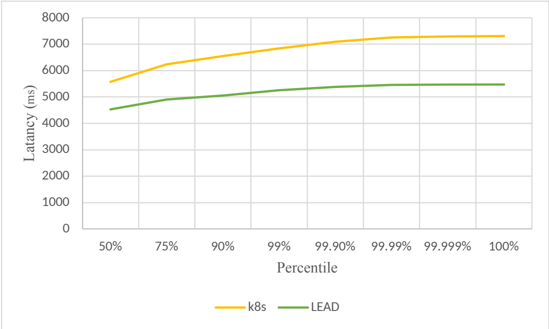

Fig. 6. Latency by percentile distribution in the eight-hour pursuer.

Another important evaluation metric is average latency over each workload run. The results in Figure 8 indicate that proper service deployment significantly reduce the average latency of the application. The performance of the proposed algorithm has been shown to reduce the average latency by over 20%.

The findings, as illustrated in Figure 8, are promising. They show that our proposed algorithm significantly enhances the system's capability to handle requests, making it more responsive. This improvement is particularly notable in scenarios where multiple services collaborate, leading to faster completion of business processes. It's a clear indication that efficient collaboration between services can greatly enhance overall system performance.

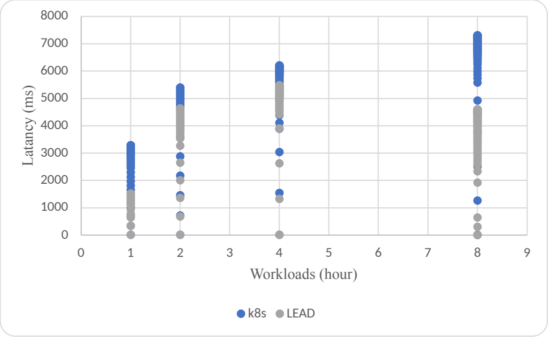

Fig. 7. 99th percentile latency distribution for four different workload durations.

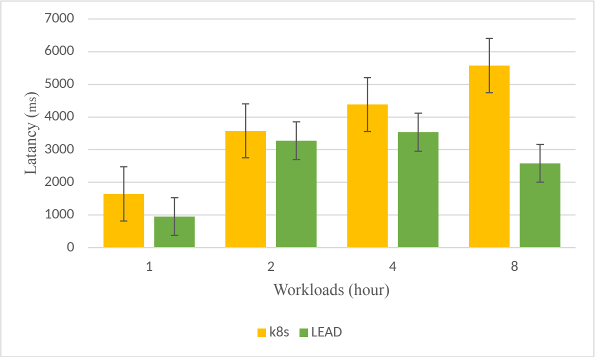

Fig. 8. End-to-End Average Latency

One of the critical challenges for cloud applications is minimizing the 99th percentile response time of the system. Failure to meet this metric may result in penalties for cloud service providers. Our experiments demonstrate that the system can consistently maintain a response time below the specified threshold with proper service deployment. When latency remains below the threshold, cloud providers do not need to assign new resources to avoid failing in the metric, resulting in customer cost savings.

Additionally, our evaluation revealed the identification of critical paths within the application before the initial deployment, which could assist in allocating resources more efficiently. The proposed algorithm demonstrated its ability to identify and highlight critical and congested paths, demonstrating the accuracy and effectiveness of the selected criteria before the algorithm initial deployment.

V. Conclusion

We presented LEAD, a Latency-Efficient Application Deployment framework integrated into Kubernetes that aims to improve application performance with ideal service placement. LEAD works by leveraging two main components: (i) Prioritizer, the heart of the framework with two algorithms, scoring and dynamic performance monitoring, and (ii) Affinity Rule Generator, which applies the affinity rules in the Kubernetes syntax to the configuration files and hands it over to the Scheduler. We found that using LEAD reduces tail latency and beats the Kubernetes default scheduler under heavy pressure. For a web application like Hotel Reservation by DeathStarBench, LEAD matches the required baselines of 99th percentile tail-latency, reducing it by 20%

References

[1] N. E. Team, “Microservices at netflix: Lessons for architectural design,” 2015. [Online]. Available: https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

[2] Microsoft, “Microservices architecture style.” [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

[3] Apigee, “Adapt or die: A microservices story at google.” [Online]. Available: https://www.slideshare.net/apigee/adapt-or-die-a-microservices-story-at-google

[4] E. Wolff, *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 2016.

[5] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang, “Overload control for scaling wechat microservices,” in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 149–161.

[6] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai, “Imbalance in the cloud: An analysis on alibaba cluster trace,” in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 2884–2892. [Online]. Available: https://www.researchgate.net/publication/262326398_Heterogeneity_and_dynamicity_of_clouds_at_scale_Google_trace_analysis

[7] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, “Heterogeneity and dynamicity of clouds at scale: Google trace analysis,” in *Proceedings of the third ACM symposium on cloud computing*. ACM, 2012, pp. 1–13. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8258257/

[8] A. F. Baarzi and G. Kesidis, “Showar: Right-sizing and efficient scheduling of microservices,” pp. 427–441, 2021.

[9] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, “Sinan: Ml-based and qos-aware resource management for cloud microservices,” pp. 167–181, 2021.

[10] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, “An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems,” pp. 3–18, 2019.

[11] “Docker containers,” https://www.docker.com/, accessed: 2023-04-27.

[12] “Why grpc?” https://grpc.io/, accessed: 2023-04-27.

[13] Microsoft, “Rest api guidelines,” https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design, 2023, accessed: 2023-04-27.

[14] “Kubernetes: Production-grade container orchestration,” https://kubernetes.io/, accessed: 2023-04-27.

[15] “Assign pod node,” https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/.

[16] “Autoscaling in amazon web services cloud,” https://aws.amazon.com/autoscaling/.

[17] “Autoscaling in google cloud platform,” https://cloud.google.com/compute/docs/load-balancing-andautoscaling.

[18] “Autoscaling in microsoft azure,” https://azure.microsoft.com/en-us/services/autoscaling/.

[19] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, “A review of auto-scaling techniques for elastic applications in cloud environments,” *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014. [Online]. Available: https://link.springer.com/article/10.1007/s10723-014-9314-8

[20] A. U. Gias, G. Casale, and M. Woodside, “Atom: Model-driven autoscaling for microservices,” in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 1994–2004.

[21] “Kubernetes autoscalers,” https://github.com/kubernetes/autoscaler.

[22] K. Rzadca, P. Findeisen, J. Świderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, “Title of the article,” *Journal or Conference Name*, vol. Volume Number, no. Issue Number, p. Page Numbers, April 2020, additional Information.