LEAD: Latency-Efficient Application Deployment for Microservices Architecture

1st Mohammad-Ali Mirzaei*
Department of Computer Engineering
Amirkabir University of Technology
Tehran, Iran
momirzaei@aut.ac.ir

2nd Parsa Poorsistani*
Department of Math and Computer Science
Amirkabir University of Technology
Tehran, Iran
parsa.ps@aut.ac.ir

3rd Seyyed Ahmad Javadi
Department of Computer Engineering
Amirkabir University of Technology
Tehran, Iran
sajavadi@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 facilitates agile, modular development, allowing scalability and efficient troubleshooting [1]–[3]. However, scheduling these services to optimize communication latency and resource utilization remains challenging [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. While Kubernetes is widely used for container orchestration, its default scheduler does not consider service dependencies. Solutions like SINAN [8] and Nodens [9] address resource management but overlook pod placement based on service interactions. Other solutions such as SHOWAR [10], although provides hints for pod placement, these features are not as robust or primary, as it focuses more on resource allocation and autoscaling.

*Mohammad-Ali Mirzaei and Parsa Poorsistani are co-first authors.

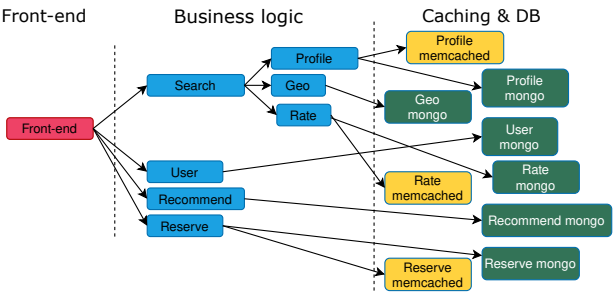

Fig. 1. Hotel reservation application [11] consists of multiple services and databases. Client requests are first routed to a front-end service, which directs them to appropriate layers based on request type. The diagram illustrates processing requests through microservices, focusing on retrieving hotel profiles and rates, recommending hotels, and completing reservations.

LEAD operates orthogonally to prior approaches, focusing on pod placement 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 that inter-node communication. 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 two components. One identifies critical paths in a microservices architecture graph. LEAD then continuously monitors the deployed services by collecting real-time data from monitoring agents for further deployment in the event of performance issues, such as tail-latency violations or resource overuse. The Other component 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. 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 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 service placement: LEAD automatically improves service placement, reducing tail latency.
- Dynamic performance 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. It comprises small, independent microservices that communicate over the network. For example, Figure 1 illustrates a Hotel Reservation application's microservice structure. These microservices, each in its container [12], interact via HTTP API requests or RPC calls [13], [14], forming a dependency graph.

Kubernetes is an open-source platform designed to automate application container deployment, scaling, and life-cycle [15]. 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.

Affinity rules in Kubernetes help influence pod placement, offering flexibility with "preferred" rules and strict enforcement with "required" rules [16]. Preferred rules can be weighted 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.

B. Related works

Here, we provide a brief overview of state-of-the-art approaches to autoscaling and resource management for microservices. Autoscaling has been studied in the context of public cloud resources [17]–[19] including different types of workloads [20], [21]. The autoscaling frameworks for microservices, which are widely used by practitioners in the industry, include those of Kubernetes [15] and Google Autopilot autoscaler [22]. Google's *Autopilot autoscaler* and Kubernetes *Horizontal Pod Autoscaler* (HPA) aim to automate the scaling of pods based on observed CPU usage and other metrics but do not consider the service interaction graph complexity.

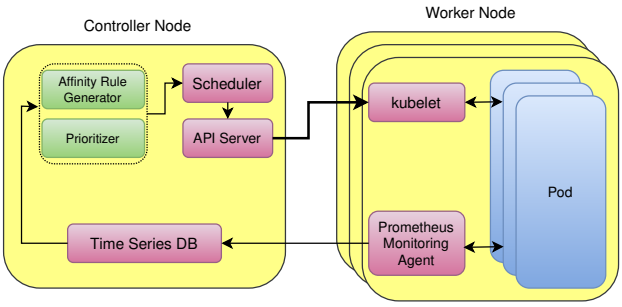

Fig. 2. LEAD architecture overview: Resource usage logs are collected via corresponding agents on each node and saved into a time series database. LEAD then utilizes these collected metrics to determine affinity rules by interfacing with the Kubernetes scheduler.

SHOWAR [10] uses historical resource data for vertical scaling and control theory for horizontal scaling, generating affinity rules based on simple resource-utilization correlations. SINAN [8] employs machine learning to predict and mitigate service level objective violations which is very resource-intensive and does not focus on affinity rules. Nodens [9] addresses dynamic microservice challenges by monitoring network bandwidth usage, updating microservice loads, and efficiently draining queued queries. It ensures fast resource adjustments and QoS recovery.

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 graph. 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

LEAD architecture, as illustrated in Figure 2, comprises two main components: (i) the Prioritizer and (ii) the Affinity Rule Generator. At the initial deployment, a 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:

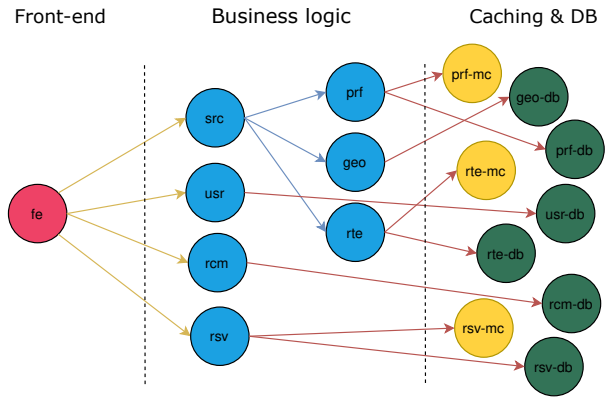

Fig. 3. The image illustrates a sample graph extracted from the hotel reservation application's architecture, which is essential for initiating the algorithm. It represents the data structure and relationships required by the algorithm to operate effectively.

- 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 edges: These services are often crucial vertices 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 above information, the Prioritizer can identify the critical paths in the service interaction graph. These critical paths are sorted and assigned a normalized score, which is meant to resemble Kubernetes affinity rule weighting mechanism. The Affinity Rule Generator takes these scores and applies them to the configuration files of each service. Finally, these modified configuration files are handed over to the Kubernetes scheduler for deployment. 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.

B. Critical path detection and monitoring algorithms

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

1) Scoring: Algorithm 1 evaluates critical paths through the service mesh based on several key factors to identify paths that reduce latency. The length of a path reflects the number of vertices a request must traverse; longer paths indicate potential bottlenecks, as failures or delays at any node affect the entire chain. The number of pods per node shows the anticipated request load; nodes with more pods are designed to handle higher traffic and can scale accordingly. The number of service-to-service interactions highlights nodes with higher dependencies, signaling that these nodes are central to many services and might experience higher request volumes.

Algorithm 1 Scoring

Require: $G = (V, E)$: Directed graph representing the service mesh
Require: Gateway: 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
```

RPS, though not available at the initial deployment, would be gathered by the monitoring system for the next phases of scoring. 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) Real-time monitoring : Another component of the Prioritizer is the performance monitoring algorithm. Algorithm 2 is designed to continuously monitor services in the application and dynamically scale out any service that becomes a bottleneck.

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. In order to avoid additional overhead, LEAD is designed

Algorithm 2 Real-time monitoring

Require: None
Ensure: None (side effects: scales out services based on monitoring data)

```
while true do
    if detectLatencyViolation() then
        bottleneckServices ← analyzeAndIdentifyBottleneck()
        if bottleneckServices is not None then
            call Algorithm 1
            scaleOut(bottleneckServices)
        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:
            bottleneckServices.add(service)
    return bottleneckServices
```

to be lazy; it re-prioritizes the paths only when a violation is detected. Currently, LEAD only supports monitoring via Prometheus standards.

C. Affinity rule generator

After prioritizing paths, the "Affinity Rule Generator" component applies Kubernetes affinity rules in "Preferred" mode to co-locate interacting services, 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.

Algorithm 3 requires a path and a weight which indicates the priority of the path. The services within the path are sorted in ascending order to ensure that child services are placed behind their parent services. This sorting is crucial because each parent service has multiple child services, and it is important to determine the placement of child services relative to their parent services. The algorithm iterates over each service in the sorted list, and during each iteration, it 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 for 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.

Algorithm 3 Affinity rule generator

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
```

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.

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.

C. 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 [11], 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

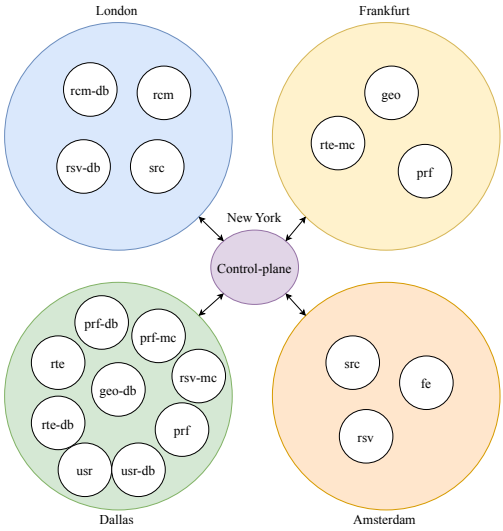

Fig. 4. Kubernetes pod placement

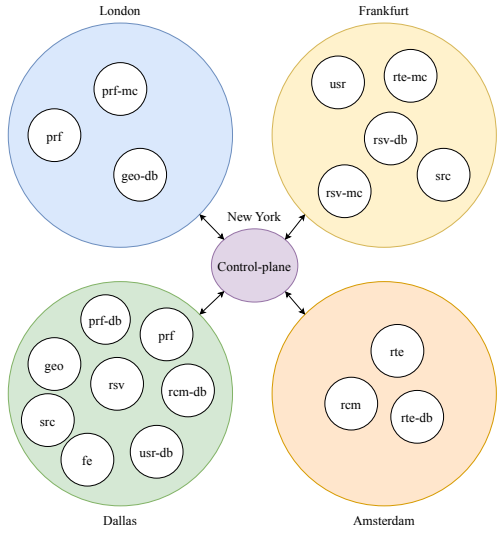

Fig. 5. LEAD pod placement

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 Figures 4 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.

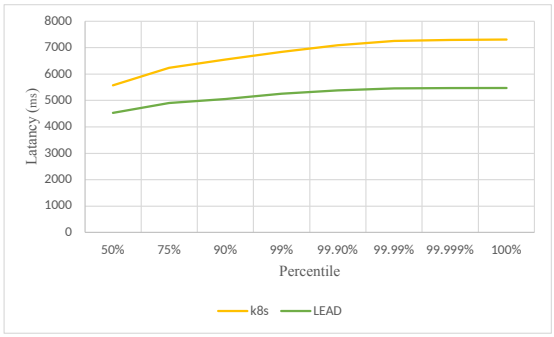

Fig. 6. Latency by percentile distribution in the eight-hour pursuer.

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%.

Our results, as illustrated in Figure 8, show that LEAD 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.

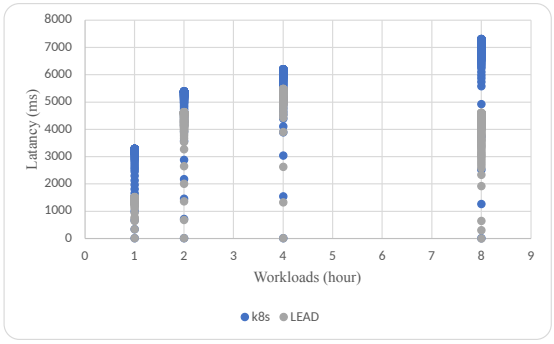

Fig. 7. 99^{th} percentile latency distribution for four different workload durations.

One of the critical challenges for cloud applications is minimizing the 99^{th} 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

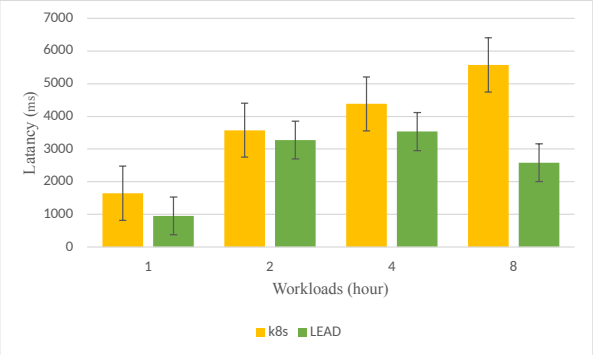

Fig. 8. Average end-to-end latency

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. LEAD demonstrated its ability to identify and highlight critical and congested paths, demonstrating the accuracy and effectiveness of the selected criteria before initial deployment.

V. Discussion

While LEAD demonstrates improvements in reducing end-to-end latency, it faces some challenges. Although continuous real-time monitoring is essential for modern systems and is a must, it inherently introduces computational and network overhead. Additionally, the time required to detect performance issues and recompute placements can introduce delays in dynamic environments with rapid service updates or roll-outs. While focusing on latency reduction, other aspects such as availability may be compromised, requiring careful balancing of performance trade-offs.

Future work will focus on developing more efficient algorithms for critical path analysis and adaptive monitoring techniques. The framework reliance on Kubernetes affinity rules may become challenging due to inherent scheduling constraints and complexities in rule management. Additionally, strategies to enhance fault tolerance and resilience will be investigated to ensure that latency improvements do not compromise system reliability and availability; for example, incorporating advanced network topology awareness.

VI. Conclusion

We presented LEAD, a Latency-Efficient Application Deployment framework integrated into Kubernetes that aims to improve application performance with better 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 99^{th} 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] 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.

[9] J. Shi, H. Zhang, Z. Tong, Q. Chen, K. Fu, and M. Guo, "Nodens: Enabling resource efficient and fast {QoS} recovery of dynamic microservice applications in datacenters," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 403–417.

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

[11] 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.

[12] "Docker containers," https://www.docker.com/, accessed: 2023-04-27.

[13] "Why grpc?" https://grpc.io/, accessed: 2023-04-27.

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

[15] "Kubernetes: Production-grade container orchestration," https://kubernetes.io/, accessed: 2023-04-27.

[16] "Assign pod node," https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/.

[17] "Autoscaling in amazon web services cloud," https://aws.amazon.com/autoscaling/.

[18] "Autoscaling in google cloud platform," https://cloud.google.com/compute/docs/load-balancing-andautoscaling.

[19] "Autoscaling in microsoft azure," https://azure.microsoft.com/en-us/services/autoscaling/.

[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.