Microns: Connection Subsetting for Microservices in Shared Clusters

Microservice applications typically employ a technique known as connection subsetting to ensure resource-efficient and stable communication. In this technique, upstream containers selectively route requests to a limited subset of downstream counterparts via persistent connections. However, the interdependency in microservice applications along with the complex runtime environments pose significant challenges for effective connection subsetting, rendering traditional strategies notably inefficient.
In this paper, we present Microns, a connection subsetting framework designated for microservices in shared clusters. At the application level, Microns effectively handles the complex call dependencies in applications and meticulously determines the number of connections maintained by each pair of dependent microservices. At the microservice level, Microns manages the connection relationships between dependent containers according to their respective contributions on end-to-end latency. Experiments across microservice benchmarks and large scale simulations demonstrate that Microns achieves a significant reduction on end-to-end latency by over 74.4% compared with the state-of-the-art strategies.

1 INTRODUCTION

In recent years, there has been a growing trend towards developing user-facing applications with the microservice architecture [2, 4, 17]. This architecture decomposes applications into tens to hundreds of independent, loosely-coupled microservices [19, 25, 29, 31], each providing specific functionality and collectively forming a complex dependency graph [6, 32]. The microservice architecture offers advantages such as ease of maintenance, efficient resource management, and high reliability. Consequently, leading technology companies have increasingly embraced microservice architectures for the development of their online applications.

In a microservice application, microservices continuously interact with their dependent counterparts to handle user requests throughout their lifetime. To avoid the overhead of re-establishing connections for each interaction, persistent connections are typically established between dependent containers at startup, ensuring seamless and efficient communication [11, 16]. However, given the inherent complexity and large scale of microservices, establishing connections indiscriminately between all containers with dependencies can result in significant resource (e.g., CPU, Memory) overheads to maintain these connections, which degrades both performance and scalability. To mitigate the overhead, production clusters often implement *connection subsetting* [44], configuring each upstream container to maintain connections with only a limited subset of downstream containers. Our analysis of Alibaba's production traces reveals that the majority of upstream containers connect to fewer than 20% of their downstream counterparts. Additionally, as the scale of microservices increases, the number of connections established by each upstream container stabilizes (§ 2.3).

Connection subsetting has been widely implemented by internet giants such as Google [18, 43], Twitter [40], Uber [41], and Netflix to manage connections in their large scale clusters. Existing strategies [14, 18, 35, 40–43] primarily focus on balancing the workloads or connections allocated to containers. Although these approaches effectively mitigate connection overheads, their benefits can not be fully materialized in the microservice era. First, in a single microservice, containers are often distributed across multiple physical hosts, each exposed to varying levels of resource contention from co-located microservices or best-effort applications (e.g., batch analytics) running on shared hosts. This uneven resource contention can lead to significant performance disparities between containers of the microservice. If such imbalances fail to be addressed when establishing connections between dependent containers, it can result in substantial performance degradation, as highly interfered containers can become bottlenecks in request handling. Second, in a microservice application, the end-to-end latency is interdependent with each pair of dependent microservices.

As the number of connections maintained by these pairs greatly affects their individual latency, fluctuations in their connection quantity can lead to notable changes in the overall end-to-end performance. Furthermore, given the disparities in latency and call dependencies between different microservice pairs, adjusting the number of connections for specific pairs can have distinct impacts on the end-to-end latency. However, existing strategies manage connections for each microservice pair separately, failing to account for the interdependency between individual pairs and the whole application. This can result in significant performance degradation, as microservice pairs with minimal influence on end-to-end latency may still consume excessive connections.

Given the limitations of existing strategies, we present Microns, a connection subsetting framework designed to optimize end-to-end latency for microservice applications in shared clusters. Microns models the connection subsetting as a joint optimization problem, aiming to optimize both the number of connections and the connection relationships between containers for each microservice pair. However, this problem is intractable due to its vast search space and the complex interdependency between the two knobs. To address this, Microns employs an efficient iterative approach that decouples the optimization process. It begins by independently adjusting the number of connections for each microservice pair using a learning-based method to enhance end-to-end latency. Following this, Microns efficiently manages the connection relationships between dependent containers for each microservice pair, addressing performance disparities among containers. By executing these two steps interactively, Microns progressively refines the connection subsetting solution, ultimately achieving the optimal configuration for minimizing end-to-end latency. Notably, Microns ensures high convergence efficiency with minimal overheads, even in large-scale deployments, thereby minimizing interference with the execution of microservices.

We implement a prototype of Microns on the widely-adopted container orchestration framework Kubernetes [24] in our local cluster and comprehensively evaluate it with real-world applications in DeathStarbench [16] and Alibaba production traces [3] to demonstrate Microns' effectiveness. Empirical findings reveal that Microns is capable of substantially diminishing end-to-end latency by a notable margin of over 74.4%. In summary, Microns makes the following main contributions:

- **In-depth analysis of connection subsetting.** Microns is the first system to comprehensively analyze the deployment of connection subsetting and the limitations of existing strategies within the microservice architecture.
- **Efficient optimization of connection subsetting.** Microns develops an efficient optimization scheme for connection subsetting to significantly enhance the end-to-end latency of microservice applications while incurring minimal computational overheads.
- **Extensive evaluations.** We conduct real-world experiments and large scale simulations to demonstrate the effectiveness of Microns and its scalability in large scale deployments.

2 BACKGROUND

2.1 Microservices Background

A microservice application consists of numerous loosely coupled microservices, each running independently in hundreds of containers and dedicated to performing specific operations to handle user requests. Typically, a user request is initially routed to an entry-point microservice (e.g., Nginx) and subsequently triggers a series of inter-microservice calls, either serially or in parallel, throughout its lifetime. Microservices that initiate these calls are known as *upstream microservices*, while those receiving the calls are referred to as *downstream microservices*. Together, an upstream microservice and its corresponding downstream microservice constitute a *microservice pair*. These calls, along with the microservices involved, collectively form a *dependency graph*, where each

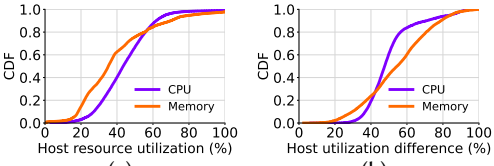


Fig. 1. (a) Cumulative distribution function (CDF) of the host resource utilization for each container. (b) CDF of difference in resource utilization between the residing hosts with the highest and lowest utilization for each microservice.

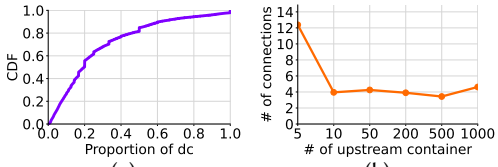


Fig. 2. (a) CDF of the proportion of downstream containers (dc) connected by individual upstream containers. (b) The average number of connections established by each upstream container for different groups of microservice pairs, with each group categorized by the number of upstream containers.

vertex indicates a microservice, and each directed edge represents a microservice pair. The end-to-end latency of the request is measured as the time elapsed from the moment the user initiates the request to when the response is received. To ensure a consistent and reliable user experience, the Service Level Agreement (SLA) is typically defined based on the tail end-to-end latency, such as the 95th percentile.

2.2 Resource Contention

Nowadays, shared clusters, which colocate latency-sensitive applications (e.g., microservices [12]) with best-effort applications such as batch jobs [37] on the same physical hosts to improve the overall resource utilization, have become the de-facto production practice [38, 39, 45, 48]. However, due to the resource-intensive nature of batch jobs, this colocation can incur significant resource contention, negatively impacting the performance of microservices sharing the same infrastructure. Moreover, given the diverse resource demands of different batch jobs, containers from the same microservice but running on different physical hosts may experience different levels of resource contention [28, 39, 49].

To investigate the resource contention, we conduct an analysis over ten thousand microservices on Alibaba production traces [3]. We quantify resource contention by measuring the CPU and memory utilization of the physical hosts on which microservice containers are deployed. As depicted in Fig.1a, a notable proportion of containers experiences high resource contention, with over 60% running on hosts with CPU and memory utilization ranging from 30% to 80%. Additionally, for each microservice, we assess the disparity in resource utilization between its residing host with the highest and lowest utilization to assess the extent of imbalanced resource contention experienced by its containers. As illustrated in Fig. 1b, for half of the microservices, the utilization difference exceeds 50%, indicating significant variability in resource contention among containers of the same microservice. This uneven resource contention causes substantial performance disparities among containers, as container performance is highly sensitive to resource contention. Experimental results show that when the resource utilization of physical hosts reaches 60%, container latency can increase by more than 5× compared to containers unaffected by resource contention..

2.3 Connection Subsetting for Microservices

Microservices typically rely on connection-oriented protocols like RPCs [26, 27] to facilitate smooth inter-microservice communication. With these protocols, persistent connections are established between containers from dependent microservices during the startup of containers, streamlining future interactions without the need to repeatedly establish connections for each individual query. However, maintaining these persistent connections incurs non-negligible resource overheads. Each

connection requires memory for data buffering, as well as CPU and network bandwidth for periodic health check packets [7, 44]. These overheads become especially significant in large-scale production environments, where microservices dynamically spawn thousands of containers, resulting in the establishment of millions of inter-container connections. The sheer volume of these connections can result in substantial resource consumption and performance degradation.

Connection subsetting [43, 44], a technique where each upstream container connects with only a small subset of its downstream containers, offers a promising approach to reduce the overall connection overheads. Its effectiveness has spurred the development and implementation of multiple subsetting strategies in production clusters [14, 35, 40, 41, 43, 44] to optimize overall system performance and resource usage. For example, Google leverages the Rocksteadier Subsetting strategy, which randomly partitions downstream containers into equally sized subsets and assigns them to corresponding upstream containers. To comprehensively investigate the connection subsetting in production clusters, we conduct an analysis on Alibaba traces [3]. Specifically, we track the number of connections established by each upstream container to calculate the proportion of connected downstream containers. As depicted in Fig. 2a, most upstream containers establish connections with only a small fraction of their potential downstream counterparts, with over 50% connecting to fewer than 20% of downstream containers. Furthermore, we categorize microservice pairs based on the number of their upstream containers and analyze the average number of connections established by each upstream container within each group. As illustrated in Fig. 2b, when the number of upstream containers in a microservice pair exceeds ten, the number of connections established by individual upstream containers stabilizes, with each typically connecting to four downstream containers. This result indicates that the total number of connections maintained by individual microservice pairs is typically four times the number of their upstream containers.

To delve into the effectiveness of connection subsetting in reducing connection overheads and enhancing application performance, we conduct an experiment using the Social Network application from DeathStarBench [16], comparing the Rocksteadier Subsetting strategy implemented in Google with the Fully Connected strategy, where each upstream container connects to all its downstream counterparts. We deploy different number of containers respectively, ranging from 64 to 512, to evaluate these two strategies under different deployment scales. The containers of each microservice are configured using the default Kubernetes autoscaler [5], which scales containers based on the average resource utilization of microservices. For the Rocksteadier Subsetting strategy, the number of connections maintained by each microservice pair is four times the number of their upstream containers, aligning with our observation in § 2.3. Then we generate workloads in proportion to the total number of containers and evaluate the 95th percentile end-to-end latency.

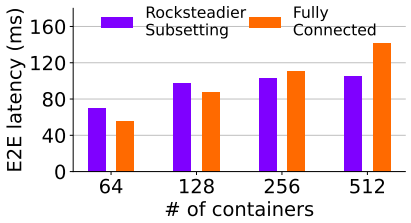
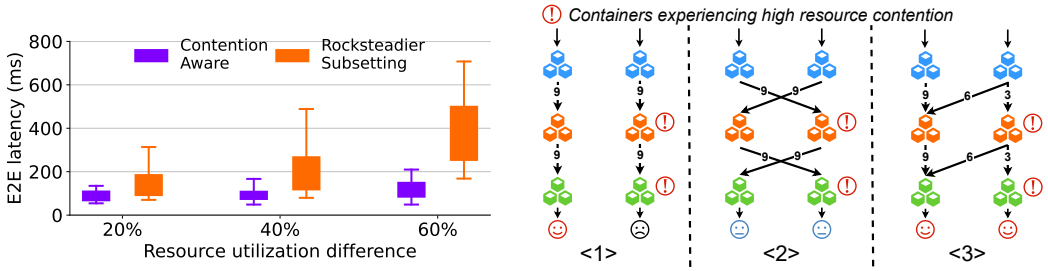


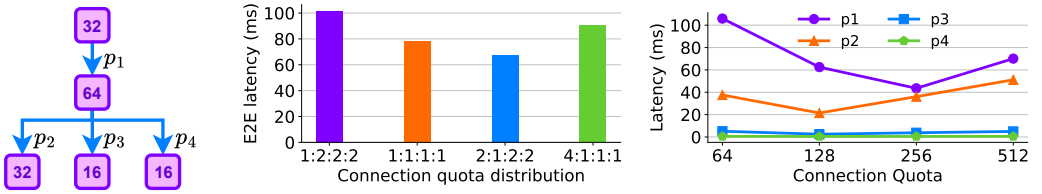
Fig. 3. Comparison of end-to-end (E2E) latency between fully connected and connection subsetting approaches across varying numbers of container deployments.

As illustrated in Fig. 3, when deploying 64 containers, which represents a small deployment scale, the Fully Connected strategy reduces end-to-end latency by 20.1% compared to the Rocksteadier Subsetting strategy. This is achieved by introducing more request traversal pathways, which helps alleviate bottlenecks caused by abnormal containers. However, as the deployment scale increases, the connection maintenance overhead becomes substantial, leading to a significant impact on the application's end-to-end latency. Particularly in a large scale where 512 containers are deployed, using the Fully Connected strategy, which establishes tens of thousands of connections between containers, can lead to a 35.1% increase in end-to-end latency. Furthermore, our experimental results indicate that using the Fully Connected strategy can result in a 14.7% increase in resource



(a) (b)

Fig. 4. (a) End-to-end (E2E) latency across different resource contention settings. (b) A schematic diagram illustrating the impact of various connection relationships between containers. Numbers on edges denote the number of connections established between containers.



(a) (b) (c)

Fig. 5. (a) Microservices and dependencies within the top four microservice pairs in Social Network. Numbers indicate the number of containers for each microservice. (b) End-to-end latency under different connection quota distributions. (c) Latency of individual microservice pairs under varying connection quotas.

usage compared to the Rocksteadier Subsetting strategy when deploying 512 containers. These results highlight the importance of mitigating connection overheads with connection subsetting.

3 MOTIVATIONS

3.1 Limitation of existing subsetting strategies

Existing connection subsetting strategies [14, 35, 40, 41, 43, 44] distribute connections or workloads evenly among different containers to achieve balanced container usage. Despite their effectiveness in reducing maintained connections, these strategies overlook the resource contention and the interdependency in microservice applications, resulting in significant performance degradation.

Contention-unaware connection management. As discussed in § 2.2, containers within the same microservice often encounter significant imbalance in resource contention, resulting in notable performance disparities. However, existing strategies connect dependent containers without accounting for these disparities, potentially causing significant degradation and fluctuations in the end-to-end latency. To examine the limitation, we implement an alternative subsetting strategy (Contention Aware), where the number of connections established by each container is configured based on the resource contention it experiences. Within this strategy, less interfered containers will maintain more connections with their counterparts, while keeping the total number of connections consistent with the Rocksteadier Subsetting. Then we conduct an experiment similar to the one described in Fig. 3. To generate resource contention, we employ iBench [13] to deploy best-effort jobs with varying resource requirements including CPU, memory bandwidth and network bandwidth. The difference in resource utilization between hosts is configured from 20% to 60%.

Fig. 4a depicts the end-to-end latency of the application across different resource contention settings. On average, the Rocksteadier Subsetting strategy results in a 1.31× higher end-to-end latency compared to the Contention Aware strategy, with the increase reaching over 2.16× when

the difference in resource utilization between hosts reaches 60%. This outcome is attributed to the contention-unaware nature of the Rocksteadier Subsetting strategy, which connects containers without factoring in imbalanced resource contention. First, when connections are unintentionally established between highly interfered containers, as depicted in Fig. 4b <1>, part of user requests may continuously traverse through containers with poor performance, causing a significant degradation in end-to-end latency. Second, even if connections between poorly performing containers are prevented, as depicted in Fig. 4b <2>, balancing the connections or workloads allocated to containers fails to mitigate the adverse effects of containers with high resource contention, leading to suboptimal end-to-end latency. Furthermore, the Rocksteadier Subsetting strategy demonstrates greater variability in end-to-end latency compared to the Contention-Aware strategy due to the high randomness in connecting containers with varying performance, which can easily result in a poor and inconsistent user experience. In contrast, the Contention Aware strategy unevenly distributes connections among containers based on their respective resource contention, as shown in Fig. 4b <3>, effectively addressing performance imbalance and significantly enhancing end-to-end latency. Therefore, it is essential to manage connections between dependent containers in a contention-aware manner to minimize performance degradation.

Interdependency-unaware connection quota assignment. Existing strategies typically adopt an unified approach to assign the number of connections, referred to as the **connection quota**, to each microservice pair respectively, aiming to independently manage connections for each pair. For example, our analysis of Alibaba traces [3] reveals that the connection quota of microservice pairs is typically four times the number of upstream containers. Nevertheless, while enhancing connection management with a connection-aware approach can effectively address the imbalanced performance among containers, assigning connection quotas without accounting for the interdependency between individual microservice pairs and the whole application may still result in suboptimal end-to-end latency. To examine this, we investigate the impact of different connection quota distributions among microservice pairs on the end-to-end latency in the Social Network application. We apply the Contention Aware strategy to manage inter-container connection relationships and configure the resource utilization difference between hosts to 30%. For illustration, we focus on the top four microservice pairs with the highest contribution on the end-to-end latency, labeled as p_1 to p_4, adjusting the connection quotas assigned to these pairs while ensuring the total quota remains 224, which is four times the number of upstream containers in the pairs. Fig. 5a depicts the microservices and dependencies of these pairs.

As illustrated in Fig. 5b, when the connection quota distributes differently among microservice pairs, the end-to-end latency exhibits a notable fluctuation. Compared to the default setting 1:2:2:2, where connection quotas are assigned based on the number of upstream containers in microservice pairs, adjusting the distribution to 2:1:2:2 results in a reduction of 32.8%. These results suggest that meticulously distributing connection quotas among microservice pairs presents another promising opportunity to further improve end-to-end latency. Note that this example involves only four microservice pairs. As the complexity of microservice applications increases, optimizing the connection quota distribution could lead to even greater improvements.

To explore the rationale behind the benefit of optimally distributing connection quotas, we evaluate the 95th percentile latency of each microservice pair, which is determined by the latency of downstream microservices, under varying connection quotas. As shown in Fig. 5c, the latency of microservice pairs is significantly influenced by the assigned connection quota. Initially, latency reduces as higher connection quota enables a more uneven distribution of connections, helping to address performance disparities among containers. However, once the quota exceeds a certain threshold, containers with high resource contention are allocated more connections and workloads, which degrades the latency. Furthermore, we observe a notable disparity in latency variation

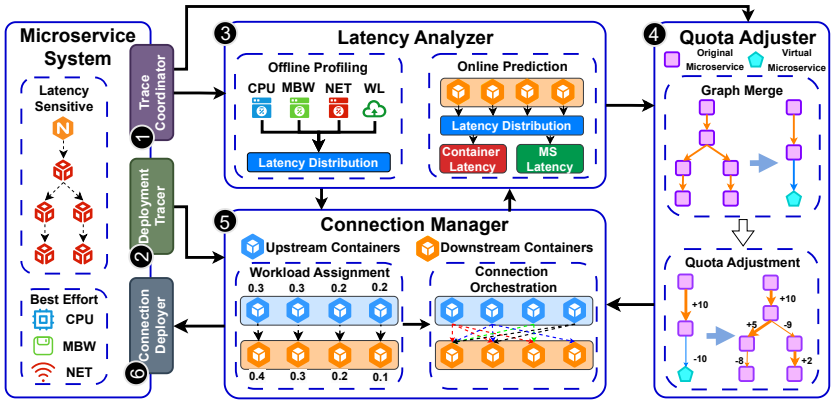


Fig. 6. The system architecture of Microns

between microservice pairs under different connection quotas: increasing the connection quota for p_1 can result in a reduction in latency by up to 64.6 ms, while the latency variation for p_4 typically remains below 0.5 ms. Thus, by factoring in the interdependency between individual microservice pairs and the whole application, and redistributing more connection quotas to microservice pairs that benefit the most, the end-to-end latency can be significantly improved.

3.2 Opportunities and challenges

We propose that connection subsetting in microservice applications should be contention-aware and interdependency-aware. Optimally distributing the connection quota among microservice pairs and managing connection relationships to address performance imbalance among containers can result in a notable reduction in end-to-end latency. However, due to the large-scale nature and complex dependencies of microservices, optimizing these two aspects presents significant challenges, including the vast search space and the complexity of modeling the impact of connection quota distribution and connection relationships on end-to-end latency. Moreover, the lightweight startup of containers and the frequent scaling of resources in microservices necessitate that both aspects should be optimized within a short period. Consequently, there is a critical need for a meticulously designed and efficient connection subsetting strategy for microservice applications.

4 THE MICRONS FRAMEWORK

In this section, we introduce the overall architecture of Microns, as illustrated in Fig. 6. Microns is a cluster-wide connection subsetting framework that executes when a scaling decision is made by the resource scaling framework, e.g., Kubernetes [24], to manage connections between dependent containers. It mainly consists of the following components:

The *Trace Coordinator* ❶ extracts latency statistics and the graph topology of applications from request traces. The *Deployment Tracer* ❷ monitors the statistics of microservice containers and the resource contention data including CPU, memory bandwidth (MBW) and network (NET) within each host. It triggers the connection subsetting procedure when a change in the resource allocation of microservices is detected.

The *Latency Analyzer* ❸ proceeds to profile and analyze the latency of containers and microservices. It executes an offline profiling process to collect latency samples from the *Trace Coordinator* and model the latency distribution of containers. At runtime, the *Latency Analyzer* queries the latency distribution of individual containers utilizing workloads and resource contention statistics to predict the latency of containers and microservice pairs (MS Latency).

Notation	Definition
E	Set of microservice pairs in the application
C	Total connection quota of the application
W	Workload arrival of the application
U_e (D_e)	Set of upstream (downstream) containers in microservice pair e
L	Latency of microservice pairs
G	Latency gradient of microservice pairs
y	Connection quota of microservice pairs
x	Connection relationships between containers
w	Workload distribution among containers

Table 1. Notations for Connection Subsetting

The *Quota Adjuster* ❹ attains the graph topology of the application from the *Trace Coordinator* and takes charge of distributing connection quotas among microservice pairs. This entails a graph merge algorithm to handle the call dependencies between microservices and leveraging the latency variation of microservices under fluctuating connection quotas collected from the *Trace Coordinator* to refine these quotas. The resulting connection quota distribution will serve as a pivotal input for the *Connection Manager*.

The *Connection Manager* ❺ is responsible for managing the connection relationships between containers. With statistics collected from the *Deployment Tracer*, it initiates a workload assignment process to derive the optimal workload distribution among containers within each microservice pair. Subsequently, it dynamically resolves the connection relationships between dependent containers. The optimal connection subsetting solution is deployed through the *Connection Deployer* ❻.

5 DESIGN DETAILS

5.1 Overview

For a specific microservice application, Microns minimizes its end-to-end latency by optimally assigning connection quotas among microservice pairs and managing the connection relationships between containers, while adhering to a predefined total connection quota for the entire application. Specifically, given the set of microservice pairs E in the application, $e \in E$ is a specific microservice pair and U_e and D_e denote the set of its upstream and downstream containers, respectively. The binary variable $x^e_{i,j}$ indicates whether the upstream container $i \in U_e$ in pair e connects to the downstream container $j \in D_e$ ($x^e_{i,j} = 1$) or not ($x^e_{i,j} = 0$). For the application, given the total amount of connection quota C, we introduce the variable y_e to denote the amount of connection quota allocated to microservice pair e. According to the empirical study in § 2.3, C is typically set to four times of total number of upstream containers in microservice pairs. Given the workload arrival W of the application, the function $Workload(x, W)$ derives the workload distribution w among containers based on the connection relationships x. Then Microns optimizes x and y to minimize the objective function $E2E(w, E)$, which captures the end-to-end latency of the application according to the workload distribution w and the call dependencies in microservice pairs E:

$$\min \quad E2E(w, E) \tag{1}$$

$$\text{s.t.} \quad \sum_{i \in U_e} \sum_{j \in D_e} x^e_{i,j} = y_e, \quad \forall e \in E \tag{2}$$

$$\sum_{e \in E} y_e = C \tag{3}$$

$$w = Workload(x, W) \tag{4}$$

$$x^e_{i,j} \in \{0, 1\}, y_e \in N^*, \quad \forall e \in E, i \in U_e, j \in D_e \tag{5}$$

Here, constraint (2) ensures the number of connections in pair e matches its connection quota y_e, while constraint (3) ensures the overall allocated quota is C.

Jointly optimizing the allocation of connection quotas across microservice pairs and inter-container connection relationships introduces significant complexity due to their interdependency. The allocation of connection quotas depends on the latency of microservice pairs, which, in turn, is influenced by the connection relationships between containers. Conversely, managing these relationships is constrained by the available connection quotas. To address this complexity, Microns adopts an iterative approach that decouples the optimization of these two aspects, solving them separately in each iteration.

Algorithm 1 presents the overall connection subsetting procedure of Microns. Initially, Microns derives the optimal workload distribution $\hat{w}$ across downstream containers without considering the limit of the connection quotas. This will be utilized for efficient and decoupled management of connection relationships, as detailed in § 5.3. In each iteration, Microns handles the complex dependencies between microservices using a graph merge algorithm. This algorithm computes the latency gradients G of microservice pairs with respect to connection quotas, as well as the overall end-to-end latency $E2E$. If the optimal end-to-end latency stabilizes within θ iterations (set to 5 by default), Microns terminates the procedure and outputs the optimal connection subsetting solution. Otherwise, Microns refines the distribution of connection quotas, y, among microservice pairs based on latency gradients. Leveraging the updated quota distribution and optimal workload assignments, Microns efficiently manages the connection relationships x between containers and solves the actual workload distribution w among containers. At the end of each iteration, Microns uses actual workloads to predict the latency of microservice pairs, L^{t+1}, for the next iteration.

Algorithm 1: Connection Subsetting

```
1  Procedure Subsetting(E, C, W):
2      Initialize L^0, y^0 and G to 0;
3      ŵ = OptimalWorkloadDistribution(E, W);
4      for t = 0, 1, 2, ... do
5          E_merge, E2E = GraphMerge(t, E, L, y, G);
6          if optimal E2E stagnates for θ iterations then
7              Output y and x that optimize E2E;
8          QuotaAdjustment(t, E_merge, C, y, G);
9          x, w = ConnectionOrchestration(t, E, y, W, ŵ);
10         Predict latency of microservice pairs L^{t+1} with w;
```

5.2 Quota Adjustment

For each iteration, the *Quota Adjuster* adjusts the connection quota distribution y among microservice pairs, aiming to minimize the end-to-end latency. Solving for the optimal distribution requires accurately modeling the latency of microservice pairs under varying connection quota distributions, which is a complex task intertwined with multiple factors, including container workloads, resource contention and call dependencies between microservices. To mitigate the complexity, the *Quota Adjuster* employs a gradient-driven learning methodology to mitigate the need for a closed-form formula of microservice pair latency with respect to connection quotas. Specifically, given the connection quota y_e^t of microservice pair e at iteration t and the corresponding latency L_e^t, the *Quota Adjuster* capitalizes on the estimated partial derivatives of microservice latency with respect

Algorithm 2: Graph Merge

Function `GraphMerge`(t, E, L, y, G)**:**
  E_{merge} = empty set of microservice pairs;
  if $t = 0$ **then**
    return E, $+\infty$;
  for $e \in E$ **do**
    if *e denotes a serial dependency* **then**
      Calculate G_e;
      $E_{merge}.insert(e)$
    else
      S = the set of subgraphs in the parallel dependency;
      e_p = MergeParallel(S, L, y, G);
      $E_{merge}.insert(e_p)$
  return E_{merge}, $\sum_{e \in E_{merge}} L_e^t$;
Function `MergeParallel`(S, L, y, G)**:**
  for $E_S \in S$ **do**
    E_S, $L_{E_S}^t$ = GraphMerge(t, E_S, L, y, G);
    Merge E_S into microservice pair e_s and calculate G_{e_s};
  Merge S into microservice pair e_p and calculate G_{e_p};
  $G_{e_s} = \frac{L_{e_s}^t}{L_{e_p}^t} G_{e_s}$, for $e_s \in S$;
  return e_p;

to connection quotas as the latency gradient, formulated as:

$$G_e = \frac{L_e^t - L_e^{t-1}}{y_e^t - y_e^{t-1}} \tag{6}$$

The *Quota Adjuster* then strategically assigns larger quotas to microservice pairs with higher latency gradients, as they present a greater potential for minimizing overall end-to-end latency.

However, relying solely on these gradients for quota adjustment can inadvertently yield suboptimal results. This is because end-to-end latency is a complex accumulation of microservice pair latency, influenced by the underlying call dependencies between microservices. Reductions in the latency of individual microservice pairs do not necessarily translate into improved end-to-end performance. Consequently, neglecting call dependencies can lead to assigning quotas to microservice pairs that have minimal impact on the overall end-to-end latency. To incorporate the call dependencies, the *Quota Adjuster* employs a graph merge algorithm to simplify the dependency graph of the application and compute the latency gradients for microservice pairs.

Algorithm 2 details the graph merge algorithm. At the initialization iteration (t=0), as latency gradients are not needed for quota adjustment, the algorithm directly outputs the original dependency graph and sets the end-to-end latency to $+\infty$. In the subsequent iteration t, the *Quota Adjuster* examines all microservice pairs e involved in the application. If pair e denotes a serial dependency between microservices, the *Quota Adjuster* calculates its latency gradient G_e based on equation (6) and inserts it into the merged graph. In contrast, for a parallel dependency, where an upstream microservice calls a set of microservice subgraphs S in parallel, the *Quota Adjuster*

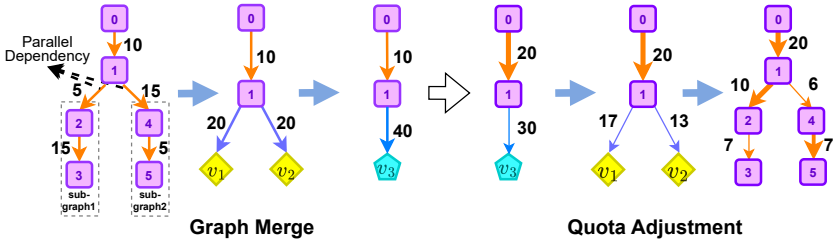


Fig. 7. An example of the algorithm of graph merge and quota adjustment in Microns. The numbers on edges denote the connection quotas of microservice pairs.

merges it. Specifically, for each subgraph $E_S \in S$, the *Quota Adjuster* first simplifies the subgraph, derives its latency and merges it into a new microservice pair e_s. The connection quota of e_s is the cumulative quotas of microservice pairs involved in E_S, represented as $y^t_{e_s} = \sum_{e \in E_S} y^t_e$, and the latency gradient G_{e_s} is calculated according to equation (6). Next, the *Quota Adjuster* merges the set of subgraphs S into a new microservice pair e_p to represent the parallel dependency. Since these subgraphs are executed in parallel, the latency of e_p is the maximum execution time among subgraphs, formulated as $L^t_{e_p} = max_{e_s \in S} L^t_{e_s}$. To account for the fact that subgraphs with lower latency in parallel dependencies have a weakened impact on end-to-end latency, the latency gradients of these subgraphs are scaled based on the difference between their latency and the overall latency of the parallel dependency. This adjustment helps refine the quota allocation by emphasizing the subgraphs that most significantly affect the overall performance. Once all parallel dependencies have been merged, the algorithm outputs a simplified dependency graph E_{merge}, which consists of only serial dependencies, and the latency gradients G of microservice pairs. The end-to-end latency of the application is then computed as the cumulative latency of the microservice pairs in this simplified graph. This algorithm takes both serial and parallel dependencies into account to facilitate more accurate latency gradients and optimal end-to-end latency.

Fig. 7 illustrates an example of the graph merge process in Microns. In this example, microservice 0 calls 1 in a serial manner, while 1 calls two subgraphs {2,3} and {4,5} in parallel. To simplify this parallel dependency, the *Quota Adjuster* first merges these two subgraphs into virtual microservices v_1 and v_2 respectively, and creates new microservice pairs $1 \rightarrow v_1$ and $1 \rightarrow v_2$. The connection quotas of these new pairs are the cumulative quotas of their respective original microservice pairs. Subsequently, the *Quota Adjuster* merges the whole parallel dependency into a new microservice pair $1 \rightarrow v_3$. Through merging all parallel dependencies, the microservice dependency graph is simplified to involve only serial dependencies.

With the simplified dependency graph, the *Quota Adjuster* leverages the gradient-driven learning methodology to adjust the connection quota allocation, as outlined in algorithm 3. For the initial iteration $t = 0$, the connection quota of each microservice pair e is initialized proportional to the number of its upstream containers, calculated as $y^{t+1}_e = \frac{|U_e|C}{\sum_{p \in E_{merge}} |U_p|}$. In each subsequent iteration t, the *Quota Adjuster* first deducts the maximum gradient value from each latency gradient to align the sign of gradient values (line 4). Then it adjusts the connection quotas based on the difference between individual latency gradients G and the average value $\overline{G}$, aiming to balance the gradient values across all microservice pairs. The rationale is that when latency gradients are equal across microservice pairs, adjusting the connection quota allocated to each microservice pair impacts end-to-end latency equally. Therefore, further refinement of the quota distribution will not improve end-to-end latency. The adjustment of connection quotas is scaled by a learning rate η, which is set to 0.5 and discussed in § 7.4. After each adjustment, the connection quota distribution is normalized to ensure that the total quota across all microservice pairs matches C. Finally, the *Quota Adjuster*

Algorithm 3: Connection Quota Adjustment

Function `QuotaAdjustment`(t, E_{merge}, C, y, G)**:**
  if $t = 0$ **then**
    Initialize y^{t+1} based on the number of upstream containers in microservice pairs.
  $G_e = G_e - \max_{p \in E_{merge}} G_p$ for $e \in E_{merge}$; `// Align the sign of gradient values`
  $\overline{G} = \frac{\sum_{p \in E_{merge}} G_p}{|E_{merge}|}$; `//` $|E_{merge}|$ `is the cardinality of set` E_{merge}
  for $e \in E_{merge}$ **do**
    $y_e^{t+1} = y_e^t * (1 + \eta \frac{\overline{G} - G_e}{\overline{G}})$;
  Normalize y^{t+1} to ensure $\sum_{e \in E_{merge}} y_e^{t+1} = C$;
  for $e \in E_{merge}$ **do**
    if *e denotes a parallel dependency* **then**
      S = subgraph set of the parallel dependency;
      QuotaAdjustment(t, S, y_e^{t+1}, y, G);
    else if *e denotes a subgraph* **then**
      E = microservice pair set of the subgraph;
      QuotaAdjustment(t, E, y_e^{t+1}, y, G);

decouples the merged microservice pairs (representing parallel dependencies or subgraphs) and applies the quota adjustment within them, as illustrated in Figure 7. After applying a depth-first decomposition to these merged microservice pairs, the connection quotas of all microservice pairs in the original dependency graph are updated.

5.3 Connection Management

Based on the connection quota distribution provided by the *Quota Adjuster*, the *Connection Manager* manages the connection relationships between dependent containers to achieve optimal workload distribution that minimizes the latency of each microservice pair. To this end, a naive approach is to first formulate the optimization of connection management as a mixed integer non-linear programming (MINLP) that minimizes the latency difference among the downstream containers of each microservice pair. In this formulation, the constraints regulate the number of connections does not exceed the allocated quota generated by the *Quota Adjuster*. Additionally, the workload distribution must align with the connection decisions: workload can only be assigned to downstream containers that are connected to the upstream container. However, applying this approach in practice presents two challenges. First, the large scale of microservice applications makes connection management computationally expensive, as it can involve establishing millions of connections. Second, connection management between different microservice pairs has cascading effects. The connections between containers in an upstream microservice pair affect the workload distribution of its downstream containers, which then impacts the connection management and latency of downstream microservice pairs. This leads to uncertainty in estimating latency gradients during quota adjustment, reducing the convergence efficiency of Microns' connection subsetting procedure.

Given these challenges, the *Connection Manager* aims to mitigate cascade effects and ensure efficient connection management. It starts by deriving the optimal workload distribution among downstream containers, without considering the limit of the connection quota. Using this optimal workload distribution as a reference effectively decouples the connection management between

microservice pairs, as it is independent from the connection relationships between containers. Besides, it alleviates the need to model the relationships between inter-container connections and microservice pair latency, streamlining and facilitating efficient connection management. Subsequently, the *Connection Manager* refines the connection relationships between containers to ensure that the actual workload assignment closely aligns the optimal distribution, while adhering to the connection quota limit.

Optimal Workload Distribution. For each microservice pair e, the *Connection Manager* first derives the optimal workload distribution $\hat{w}_j$ for all downstream containers $j \in D_e$ to achieve balanced latency among them, without considering the constraint of the connection quota. Mathematically, this problem can be formulated as:

$$\min \sum_{j \in D_e} (l_j(\hat{w}_j) - \bar{l})^2, \quad \text{s.t.} \sum_{j \in D_e} \hat{w}_j = W. \tag{7}$$

Here, $\hat{w}_j$ denotes the amount of workloads assigned to downstream container $j \in D_e$, and $l_j(\hat{w}_j)$ represents the latency of container j under workload $\hat{w}_j$. The mean latency $\bar{l}$ is defined as $\bar{l} = \frac{\sum_{j \in D_e}(l_j(\hat{w}_j))}{|D_e|}$, where $|D_e|$ is the number of downstream containers in microservice pair e. By minimizing the sum of squared differences between individual container latency and the mean latency, the above optimization problem seeks to balance the latency across all downstream containers in the microservice pair.

In production systems, the relationship between container latency, workload, and underlying resource contention of the physical host is highly complex, making the exact form of the function $l_j(\hat{w}_j)$ difficult to obtain. As a result, the optimization problem in Eq. (8) cannot be solved directly using standard methods, such as convex optimization tools like CVXPY. To address this challenge, the *Connection Manager* employs the Newton's method [23], which iteratively approximates the solution by updating the workload distribution in a way that better aligns with the trailed gradient, rather than relying on the exact form of the latency function $l_j(\hat{w}_j)$. The process begins by evenly assigning the total workload W across all downstream containers in microservice pair e, i.e., $\hat{w}_j = \frac{W}{|D_e|}$ for all $j \in D_e$. Then, the *Connection Manager* progressively refines this initial workload distribution. In each k-th refinement step, it queries the *Latency Analyzer* for the latency $l_j(\hat{w}_j)$ of each downstream container $j \in D_e$ under its current workload $\hat{w}_j$. Newton's method is then used to update the workload distribution, as formulated in the following equation:

$$\hat{w}_{j,k+1} = \hat{w}_{j,k} + \frac{(\bar{l} - l_j(\hat{w}_{j,k}))(\hat{w}_{j,k} - \hat{w}_{j,k-1})}{(l_j(\hat{w}_{j,k}) - l_j(\hat{w}_{j,k-1}))}, \forall j \in D_e \tag{8}$$

After each refinement, the updated workloads are normalized so that their cumulative sum equals the total workload W. The refinement process continues until the latency difference between containers falls within a specified threshold, formulated as: $\frac{\sum_{j \in D_e}(\bar{l} - l_j(\hat{w}_{j,k}))}{|D_e|\bar{l}} < \beta$. By default, the termination threshold β is empirically set to 0.1, ensuring that the container latency is sufficiently balanced with fast convergence.

Connection Orchestration. While the optimal workload assignment derived by Eq. (8) effectively balances the latency of downstream containers, it may not be feasible in practice, as it does not account for connection quotas and could potentially violate the allocated quota constraints. To address this issue, the *Connection Manager* refines the connections between dependent containers to ensure that the actual workload assignment closely approximates the optimal distribution (from Eq. (8)), while adhering to the connection quota limits. Fig. 8 illustrates this procedure. Initially, the actual workload w_j of each downstream container $j \in D_e$ is initialized to 0. For each upstream container $i \in U_e$ of microservice pair e (outlined with a red border in Fig. 8), given its optimal

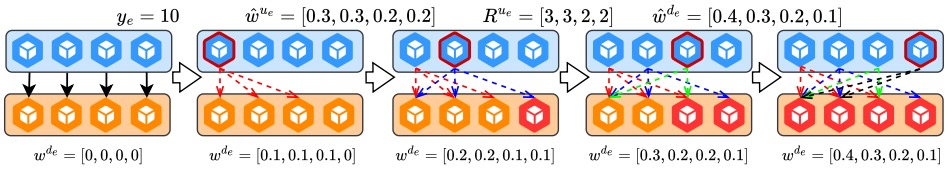


Fig. 8. An example of connection orchestration in Microns.

workload assignment $\hat{w}_i$, the number of its connections is determined by: $R_i = \lfloor \frac{\hat{w}_i y_e^{t+1}}{\sum_{i \in U_e} \hat{w}_i} \rfloor$. To select R_i downstream containers in D_e to connect with each upstream container $i \in U_e$, the *Connection Manager* sequentially selects a downstream container $j \in D_e$ to be included in this subset, establishes a connection by setting $x_{ij}^e = 1$ and increments the actual workload of j as $w_j = w_j + \frac{w_i}{R_i}$. If the workload of container j exceeds its designated optimal value $\hat{w}_j$ (highlighted in red in Fig. 8), it will no longer be selected by other upstream containers in subsequent rounds. This connection management process continues until the total number of established connections reaches the connection quota y_e^{t+1} for microservice pair e. Finally, the *Connection Manager* outputs the updated connection relationships x and the individual workloads w of all containers in each microservice pair. Notably, the time complexity of the connection management procedure for microservice pair e at iteration t is quantified as $O(y_e^{t+1})$, which represents a relatively low computational overhead.

5.4 Latency Prediction

When calculating the latency gradients for quota adjustment by Eq. (6) and deriving the optimal workload assignment by Eq. (8), it is necessary to estimate the latency of individual containers and microservice pairs under varying workloads and resource contention. Previous studies [9, 22, 30, 34, 47, 49] model the latency of microservices as a function of workloads and resource contention. While these methods effectively predict the latency of individual containers, they struggle to accurately predict microservice pair latency. The main limitation is their reliance on average workloads and resource contention of containers to estimate overall latency of microservice pairs, overlooking the uneven impacts of containers with imbalanced workloads and resource contention. This simplification results in poor accuracy in predicting microservice pair latency.

In this paper, the *Latency Analyzer* takes a more granular approach by considering the specific contribution of individual containers to the latency of microservice pairs, thereby achieving more accurate predictions. The *Latency Analyzer* begins by analyzing the latency distribution of containers for each microservice. As shown in Fig. 9, the latency distribution of a given container—determined by its workload and resource contention—follows a log-normal distribution [15, 46]. This distribution is characterized by two parameters: the mean (μ) and the standard deviation (σ). To validate this observation, we apply the Kolmogorov-Smirnov test [33] to compare the observed latency distribution with the log-normal distribution for different applications from DeathStarbench [16]. The average P-value from the test, as shown in Table 2, exceeds 0.12 for all applications, well above the conventional threshold of 0.05. This result indicates that there is no statistically significant difference between the observed latency distributions and the log-normal distribution, further validating the suitability of this model for container-level latency prediction.

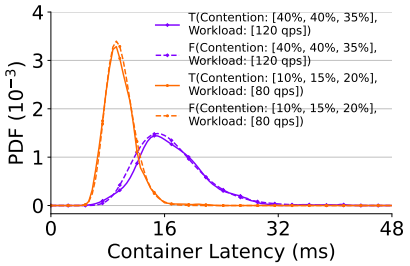


Fig. 9. Latency distributions of containers follow a log-normal distribution. T denotes the ground truth of latency distribution and F denotes the fitting results. Resource contention includes the CPU utilization, memory bandwidth utilization and network bandwidth utilization.

Application	Social Network	Hotel Reservation	Media Service
P-value	0.123	0.169	0.217

Table 2. P-value from the Kolmogorov-Smirnov test between the latency distribution of containers and the log-normal distribution in different applications.

Building on these findings, the *Latency Analyzer* first incorporates an offline profiling process to profile the latency distribution of containers for each microservice. Specifically, it collects data samples across different workloads and resource contention, where each sample captures the workload, resource contention, and the mean μ and standard deviation σ of the log-normal distribution for each container. The resource contention factors include CPU utilization, memory bandwidth utilization, and network bandwidth utilization of the physical hosts. Using the collected data samples, the *Latency Analyzer* trains two XGBoost models [10] for each microservice : one to learn the mapping from workloads and resource contention to the mean μ, and another to learn the mapping to the standard deviation σ. For a specific container j of microservice m, the mapping relationship can be formulated as $\tau_m(I_j, w_j) \longrightarrow \mu_j$ and $\pi_m(I_j, w_j) \longrightarrow \sigma_j$, where $I_j = (CPU_j, MBW_j, NET_j)$ represents the resource utilization of the physical host hosting container j, and w_j denotes the workload of container j.

With the workload distribution generated by the *Connection Manager* and resource contention statistics of containers, the *Latency Analyzer* predicts the latency of individual containers for optimal workload distribution and the latency of microservice pairs for quota adjustment. Specifically, for downstream container j in microservice pair e, the *Latency Analyzer* queries the corresponding XGBoost models for the mean μ_j and standard deviation σ_j of the latency distribution. It then solves equation $\phi(\frac{ln(l_j)-\mu_j}{\sigma_j}) = P$ to estimate the latency l_j, where ϕ represents the cumulative distribution function of the standard normal distribution, and P is the target quantile of the latency, e.g. 95th percentile. For microservice pair e, the *Latency Analyzer* models its latency distribution as a weighted mixture of the latency distribution from all its downstream containers. The cumulative distribution function of microservice pair e is $F_e = \sum_{j \in D_e} F_j \ w_j$, where F_j and w_j is the cumulative distribution function and workloads of downstream container $j \in D_e$. The *Latency Analyzer* then employs Brent's method [8] to efficiently solve the target latency L_e^{t+1} of microservice pair e in next iteration $t + 1$, using the cumulative distribution function of the mixture latency distribution.

6 IMPLEMENTATION

We develop a prototype of Microns on top of Kubernetes [24], which provides comprehensive container orchestration functionalities to deploy microservices.

For the offline profiling process, we deploy the best-effort jobs to generate diverse types of resource contention with IBench[13]. The *Trace Coordinator* queries a widely-adopted tracing system Jaeger [21] to extract latency data of microservices from traces of requests and analyze the graph structure of applications. With collected latency samples, the *Latency Analyzer* employs the python scikit-learn library [1] to fit the latency distribution and train XGBoost models.

For the online connection subsetting procedure, we design an interface for Microns' operators to specify the total connection quota as a function of the number of upstream and downstream containers. We implement the *Deployment Tracer* with the Kubernetes Python client library to monitor the real-time statistics of microservices including the number of containers within each microservice, the virtual IP address and the residing physical host of each container. The *Deployment Tracer* also queries the Prometheus [36] to attain resource contention statistics of hosts. To accelerate the procedure, we leverage multi-core parallelization on the CPU to parallelize the connection management of multiple microservice pairs. Furthermore, we implement the *Connection Deployer* to apply the optimal connection subsetting solution to corresponding applications in the network

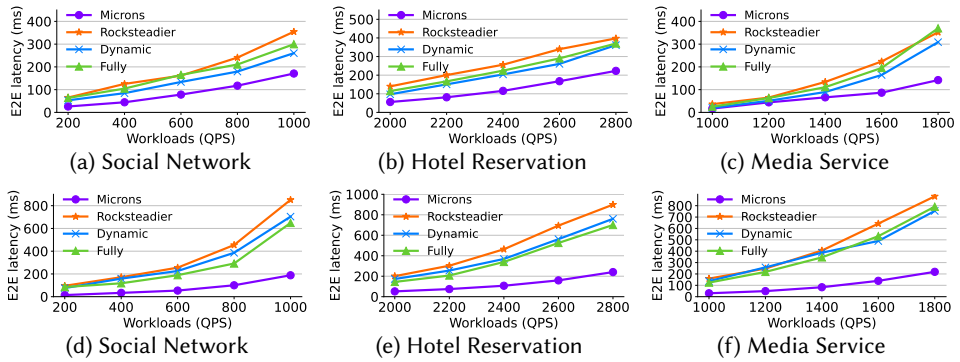


Fig. 10. End-to-end latency under static deployments. (a)-(c) Low imbalance. (d)-(f) High imbalance.

layer. The adjustment of inter-container connections is achieved by modifying forwarding rules between containers based on their virtual IP addresses in Linux *Iptables*.

7 EVALUATION

7.1 Evaluation Setup

Cluster Setup: We implement Microns in a local cluster consisting of eight two-socket physical hosts, where each host is equipped with 48 CPU cores and 64GB of RAM. Each microservice container is configured with 0.1 CPU core and 200MB of RAM.

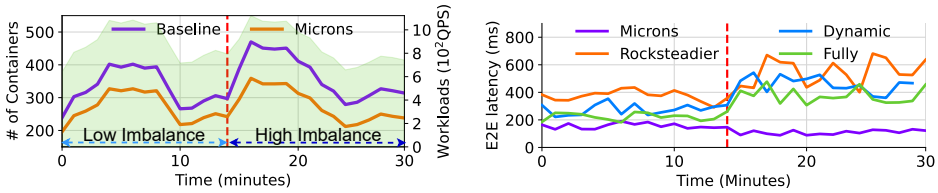
Benchmarks: We evaluate Microns with three applications including Social Network, Media Service and Hotel Reservation in the widely-adopted microservice benchmark DeathStarBench [16]. These applications contain 36, 38, and 15 microservices respectively.

Baseline Schemes: We evaluate the design of Microns with three mainstream connection subsetting strategies including the Rocksteadier Subsetting (Rocksteadier) in Google [43], the Real-Time Dynamic Subsetting (Dynamic) deployed in Uber [41], and the Fully Connected strategy (Fully). The Rocksteadier Subsetting randomly classifies downstream containers into equally sized subsets and subsequently assigns each subset to individual upstream containers to evenly distribute connections among containers. The Dynamic Subsetting tracks the workloads received by each container and establishes connections proportional to these workloads to achieve balanced workload distribution.

7.2 Performance Analysis

7.2.1 Static Deployment. We first demonstrate the effectiveness of Microns in improving the end-to-end latency of applications under static deployments. We identify that deploying 512 containers reaches the capability of our cluster under high imbalance settings. Therefore, we deploy 512 containers based on scaling results of the default autoscaler in Kubernetes. The total connection quota is set to four times the number of upstream containers in all microservice pairs, aligning with our observations in § 2.3. Then we generate workloads and evaluate the 95th percentile end-to-end latency for all applications. For illustration, we categorize the results into two groups: 'High Imbalance' indicates that the difference in resource utilization between hosts, including CPU, memory bandwidth and network bandwidth, exceeds 30%, while 'Low Imbalance' indicates that this difference is 30% or less.

As shown in Figure.10, Microns consistently outperforms the baseline schemes. The Rocksteadier Subsetting assigns an identical number of connections to both upstream and downstream containers without accounting for their individual workloads, resulting in greater performance imbalance among containers. The Dynamic Subsetting allocates connections to containers based on their



(a) Variation of workloads and containers (b) End-to-end latency over time

Fig. 11. Experimental results under dynamic deployments.

workload distribution, but it can inadvertently establish connections between containers experiencing high resource contention. The Fully Connected strategy connects all dependent containers to prevent requests from continuously passing through containers with poor performance at the cost of increased connection overheads. Moreover, it fails to balance the performance among downstream containers. In contrast, Microns factors in the imbalanced resource contention encountered by containers, carefully establishing connections to balance latency of downstream containers, and further optimizes the overall end-to-end latency through dynamic adjustment of connection quotas. Therefore, it achieves an average reduction on end-to-end latency by 67.2%, 57.5%, 60.1%, compared to Rocksteadier, Dynamic and Fully, respectively. As the workload increases, the latency imbalance among containers becomes more pronounced. Consequently, Microns achieves an average reduction of 63.2% in end-to-end latency compared to all baseline schemes. Furthermore, in high imbalance settings, where the resource contention difference between physical hosts surpasses 30%, Microns reduces the end-to-end latency by up to 74.4% compared with all baseline schemes, underscoring its superior effectiveness in improving end-to-end latency of applications.

7.2.2 Dynamic Deployment. We further assess Microns under dynamic deployments using the SocialNetwork application. We generate real-time workloads collected from production traces [3] for 30 minutes and dynamically deploy containers using the Kubernetes autoscaler. The target resource utilization of microservices is set to 30% for resource scaling. For the first 15 minutes, we configure a 20% difference in resource contention between hosts (Low Imbalance) and increase it to 40% (High Imbalance) for the remainder period. Fig. 11a illustrates the fluctuation in the number of containers over time, with the shaded area representing the workload variation. With contention-aware connection management, Microns strategically balances the resource utilization among containers, thereby reducing the overall resource usage by 21.2%. Furthermore, as shown in Fig. 11b, even with fewer containers, Microns still achieves a notable reduction by 56% in end-to-end latency. These results demonstrate that Microns can seamlessly execute with existing autoscalers, providing substantial improvements in both resource efficiency and end-to-end performance in dynamic deployment environments.

7.2.3 Effectiveness of individual modules. In this part, we demonstrate the importance of individual modules in Microns. Specifically, we implement additional connection subsetting strategies by incorporating part of the components in Microns: The Quota strategy integrates the *Quota Adjuster* and manages the connection relationships between dependent containers with the goal of balancing the workloads among containers. The Connect strategy integrates the *Connection Manager* and distributes connection quotas based on the number of upstream containers in microservice pairs. We then experiment with these strategies and evaluate the end-to-end latency of each application.

As depicted in Fig. 12, all components of Microns are indispensable in optimizing the end-to-end latency. When incorporating the *Connection Manager*, Microns achieves an average reduction of 54.6% in end-to-end latency compared to the Quota strategy, with up to a 65.5% reduction under highly imbalanced settings, demonstrating its effectiveness in addressing performance disparities among containers to improve end-to-end latency. By strategically assigning connection quotas

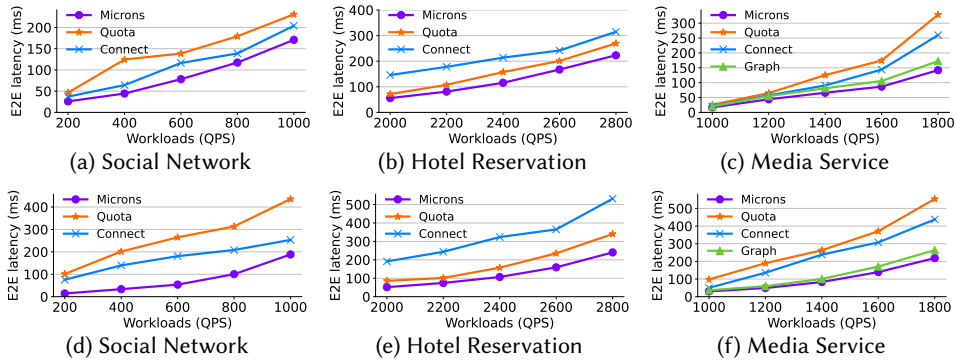


Fig. 12. Analysis of individual modules. (a)-(c) Low imbalance. (d)-(f) High imbalance.

to critical microservice pairs that significantly impact the end-to-end latency with the *Quota Adjuster*, Microns reduces the end-to-end latency by 39.2% compared to the Connect strategy. For the most complex Media Service application, this reduction is even more pronounced to 47.7%, underscoring the importance of optimally distributing connection quotas in highly complex applications. Furthermore, we implement an alternative strategy called Graph by disabling the graph merge algorithm in Microns and adjusting connection quotas without handling the call dependencies between microservices. We then evaluate its end-to-end latency in the Media Service, which involves four subgraphs executed in parallel. As shown in Fig.12c and 12f, by leveraging the graph merge algorithm to shift connection quotas from non-dominant microservice pairs in parallel dependencies to the dominant ones, Microns reduces end-to-end latency by 17.8% compared to the Graph strategy. These results underscore the importance of integrating graph merge, quota adjustment, and connection management in tandem.

7.3 Prediction Accuracy Analysis

In this section, we assess the prediction accuracy of Microns using the data samples collected from local clusters. Specifically, we deploy different combinations of resource contention and workloads in each physical host every 30 minutes and collect data samples every 30 seconds. Each sample captures the latency of microservice pairs and statistics of individual containers as mentioned in § 5.4. This process lasts for 8 hours for each application, ensuring sufficient data samples to evaluate the prediction accuracy.

We first investigate the accuracy of predicting the mean μ and standard deviation σ in latency distributions of containers based on workloads and resource contention. We train the XGBoost models with 70% data samples and evaluate them with the remaining ones for each application. As shown in Table.3, the prediction accuracy of σ ranges from 85.3% to 87.1%, while the prediction of μ achieves an accuracy exceeding 91.9%. These results indicate high precision in predicting the latency distribution across all applications.

Application	Social Network	Hotel Reservation	Media Service
μ	93.6%	94.6%	91.9%
σ	87.1%	85.3%	86.8%

Table 3. Prediction accuracy of container latency distribution.

We further assess the accuracy of predicting microservice pair latency using the predicted latency distributions of individual containers under different settings. For comparison, we implement three baseline schemes: Erms [30] models the latency of microservice pairs as a piece-wise linear function.

Parslo [34] profiles the latency distribution of microservice pairs across varying workloads. Derm [9] employs the exponential distribution to estimate the latency of microservice pairs.

As illustrated in Table .4, all baseline schemes exhibit a prediction accuracy below 80% as they overlook imbalanced workloads and resource contention among individual containers. Instead, they rely on the mean values of these statistics across containers to predict the latency of microservice pairs, leading to underestimation. Particularly in high imbalance settings, the prediction accuracy of baseline schemes dramatically drops to 70%. In contrast, Microns consistently achieves high prediction accuracy, up to 86.74% across all settings, by carefully accounting for the contribution of individual containers on microservice pair latency. These results demonstrate the effectiveness of Microns' design in achieving accurate latency predictions.

Contention Setting	Microns	Parslo	Erms	Derm
Low Imbalance	88.85%	77.27%	74.78%	76.51%
High Imbalance	86.74%	68.70%	65.19%	69.84%

Table 4. Prediction accuracy of microservice pair latency.

7.4 Sensitivity Analysis

In this section, we analyze Microns's sensitivity to related parameters and prediction accuracy. Specifically, we conduct experiments similar to § 7.2.1, varying the parameter values and introducing different levels of prediction error. We then evaluate the resulting end-to-end latency and report the average results across all applications.

Learning rate. We assess the impact of learning rate η in Microns's quota adjustment on the end-to-end latency and convergence efficiency. As shown in Fig. 13a, increasing η initially reduces end-to-end latency by preventing the algorithm from getting stuck in local optima and facilitates convergence with larger step size. However, an excessively high η can result in convergence on suboptimal solutions. Notably, the end-to-end latency varies by less than 10% when η changes, demonstrating the robustness of Microns with respect to the learning rate.

Stagnation threshold. Microns employs a stagnation threshold θ to control the termination of its iterative connection subsetting procedure. A higher threshold enables Microns to search for better solutions, but increases the number of iterations required to converge. As illustrated in Fig. 13b, the end-to-end latency exhibits a notable improvement when θ increases from 2 to 5. However, further increasing θ to 15 yields only minor improvements at the cost of reduced convergence efficiency. We set θ to 5 by default to strike a balance between convergence efficiency and end-to-end latency.

Termination threshold. The workload assignment process, which determines the optimal workload distribution among containers, is regulated by the termination threshold β. As illustrated in Fig. 13c, reducing β allows for an accurate workload distribution that better balances performance across containers, thereby enhancing the end-to-end latency. But it requires additional refinement iterations and increases computational overheads. The default value of β is set to 0.1, aiming to balance the overheads and end-to-end latency.

Maximum connection quota. The total connection quota C determines the overall connections maintained within an application. As shown in Fig.13d, increasing C from 2 to 8 times the number of upstream containers significantly reduces end-to-end latency by 27%. This stems from the increased opportunity for Microns to redistribute connection quotas among microservice pairs and address the performance disparities among containers. However, when C further increases to 16 times the number of upstream containers, the benefit is outweighed by higher connection maintenance overheads, leading to an increase in latency. We configure the default total connection quota based on our analysis of production traces [3] and provide an interface to customize it.

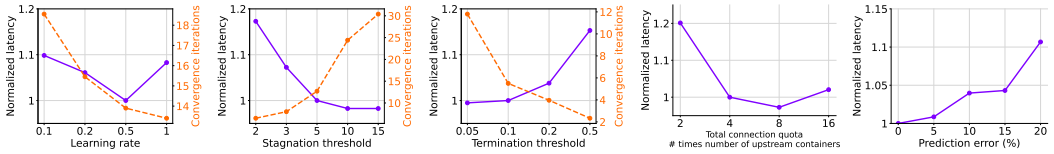


Fig. 13. End-to-end latency under different parameter values and prediction error. (a) Learning rate η in quota adjustment. (b) Stagnation threshold θ in the connection subsetting procedure. (c) Termination threshold β in workload assignment. (d) Maximum connection quota C. (e) Prediction error.

Prediction accuracy. Recognizing potential errors in latency prediction, we introduce varying levels of error to the prediction results of the *Latency Analyzer* during the connection subsetting procedure and evaluate the resulting end-to-end latency of applications. As shown in Fig. 13e, even with a 20% drop in prediction accuracy, the end-to-end latency increases by only 10.7%. This demonstrates the robustness of Microns against prediction errors.

7.5 Scalability of Microns

To evaluate Microns's scalability, we conduct experiments using highly complex microservice applications in large-scale clusters. Specifically, we leverage production traces from Alibaba [3] to analyze the latency of a complex microservice application, which consists of 425 microservices, under varying workloads and resource contention. We then simulate the deployment and execution of this application within 2,000 physical hosts, with each microservice deployed with an average of 100 containers. Within the deployment, Microns can solve the connection subsetting solution within 1.06 seconds using an Intel i7 CPU. This computation time is remarkably efficient compared to typical container startup times, which range from several to tens of seconds, demonstrating Microns's ability to scale effectively in large, complex environments.

8 RELATED WORK

Connection subsetting [14, 18, 20, 35, 40, 41, 43] has been widely adopted in production clusters. Google [43] leverages the Rocksteadier subsetting algorithm to assign equally sized subsets of downstream containers to corresponding upstream containers. Uber [41] leverages the Real-Time Dynamic Subsetting algorithm to dynamically adjust the size of upstream container subsets based on current workload distribution, ensuring efficient resource use and adaptability to fluctuating workloads. Twitter [40] develops Deterministic Aperture to enhance workload distribution by using a ring coordinate system combined with the Pick of 2 Choices (P2C) algorithm. This system maps both upstream and downstream instances onto rings, and then directs requests to downstream instances according to their positions. Envoy [14] requires tagging downstream containers with user defined metadata to allow the load balancer to effectively select the correct subset. Netflix [35] efficiently manages server pools by filtering servers to form a stable subset based on metrics like failure rates and concurrent connections. Despite the effectiveness of these techniques to regulate connection overheads, they fail to account for the unique characteristics of microservices and result in suboptimal end-to-end latency.

9 CONCLUSION

In this paper, we propose Microns, an innovative connection subsetting framework for microservices in shared clusters. The key concept is to develop a contention-aware and interdependency-aware approach, involving optimally distributing connection quotas among microservice pairs and managing connection relationships between dependent containers. Extensive evaluations reveal that Microns significantly improves the end-to-end performance while ensuring high efficiency.

REFERENCES

[1] Sklearn. https://scikit-learn.org/stable/.

[2] Alibaba cloud microservices engine. https://www.alibabacloud.com/product/microservices-engine, 2024.

[3] Alibaba microservices cluster traces. https://github.com/alibaba/clusterdata/tree/master/cluster-trace-microservices-v2022, 2022.

[4] Azure Cloud Container Apps. https://azure.microsoft.com/en-us/services/container-apps/, 2024.

[5] Kubernetes's Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2024.

[6] Rohan Basu Roy and Devesh Tiwari. Starship: Mitigating i/o bottlenecks in serverless computing for scientific workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, pages 1–29, 2024.

[7] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: How Google runs production systems.* " O'Reilly Media, Inc.", 2016.

[8] Richard P. Brent. An algorithm with guaranteed convergence for finding a zero of a function. *The computer journal*, 1971.

[9] Liao Chen, Shutian Luo, Chenyu Lin, Zizhao Mo, Huanle Xu, Kejiang Ye, and Chengzhong Xu. Derm: Sla-aware resource management for highly dynamic microservices. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 424–436. IEEE, 2024.

[10] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of SIGKDD*, 2016.

[11] Zuzhi Chen, Fuxin Jiang, Binbin Chen, Yu Li, Yunkai Zhang, Chao Huang, Rui Yang, Fan Jiang, Jianjun Chen, Wu Xiang, et al. Resource allocation with service affinity in large-scale cloud environments. In *Proc. of IEEE ICDE*, 2024.

[12] Ka-Ho Chow, Umesh Deshpande, Veera Deenadhayalan, Sangeetha Seshadri, and Ling Liu. Scad: Scalability advisor for interactive microservices on hybrid clouds. In *Companion of ACM SIGMOD*, 2023.

[13] Christina Delimitrou and Christos Kozyrakis. Ibench: Quantifying interference for datacenter applications. In *Proceedings of IISWC*, 2013.

[14] Envoy: Load balancer subsets. https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/load_balancing/subsets, 2024.

[15] Yu Gan, Guiyang Liu, Xin Zhang, Qi Zhou, Jiesheng Wu, and Jiangwei Jiang. Sleuth: A trace-based root cause analysis system for large-scale microservices with graph neural networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2023.

[16] Yu Gan, Yanqi Zhang, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of ASPLOS*, 2019.

[17] Google kubernetes engine. https://cloud.google.com/kubernetes-engine, 2024.

[18] Google site reliability engineering. https://sre.google/sre-book/load-balancing-datacenter/, 2024.

[19] Md Rajib Hossen. Pema+: A comprehensive resource manager for microservices. *ACM SIGMETRICS Performance Evaluation Review*, pages 10–12, 2024.

[20] Istio. https://istio.io/latest/about/service-mesh/, 2024.

[21] Jaeger. https://jaegertracing.io/, 2024.

[22] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, and Jason Mars. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of Eurosys*, 2019.

[23] Carl T Kelley. *Solving nonlinear equations with Newton's method*. SIAM, 2003.

[24] Kubernetes. https://kubernetes.io., 2024.

[25] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. Data management in microservices: State of the practice, challenges, and research directions. *Proceedings of the VLDB Endowment*, 2021.

[26] I-Ting Angelina Lee, Zhizhou Zhang, Abhishek Parwal, and Milind Chabbi. The tale of errors in microservices. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2024.

[27] Qian Li, Peter Kraft, Michael Cafarella, Çağatay Demiralp, Goetz Graefe, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. R3: Record-replay-retroaction for database-backed applications. *Proceedings of the VLDB Endowment*, 2023.

[28] Chengzhi Lu, Huanle Xu, Keying Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. Understanding and optimizing workloads for unified resource management in large cloud platforms. In *Proceedings of Eurosys*, 2023.

[29] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of ACM SoCC*, 2021.

[30] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Erms: Efficient resource management for shared microservices with sla guarantees. In *Proceedings of ASPLOS*, 2023.

[31] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Cheng-Zhong Xu. An in-depth study of microservice call graph and runtime performance. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[32] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2022.

[33] Frank J Massey Jr. The kolmogorov-smirnov test for goodness of fit. *Journal of the American statistical Association*, 1951.

[34] Amirhossein Mirhosseini, Sameh Elnikety, and Thomas F Wenisch. Parslo: A gradient descent-based approach for near-optimal partial slo allotment in microservices. In *Proceedings of ACM SoCC*, 2021.

[35] Serverlistsubsetfilter in netflix. https://javadoc.io/doc/com.netflix.ribbon/ribbon-loadbalancer/2.4.3/com/netflix/loadbalancer/ServerListSubsetFilter.html, 2024.

[36] Prometheus. https://prometheus.io/, 2024.

[37] Huajie Qian, Qingsong Wen, Liang Sun, Jing Gu, Qiulin Niu, and Zhimin Tang. Robustscaler: Qos-aware autoscaling for complex workloads. In *Proc. of IEEE ICDE*, 2022.

[38] Krzysztof Rzadca, Pawel Findeisen, et al. Autopilot: workload autoscaling at google. In *Proceedings of EuroSys*, 2020.

[39] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, and Jonathan Kaldor, et al. Twine: A unified cluster management system for shared infrastructure. In *Proceedings of OSDI*, 2020.

[40] Deterministic aperture: A distributed, load balancing algorithm in twitter. https://blog.x.com/engineering/en_us/topics/infrastructure/2019/daperture-load-balancer, 2019.

[41] Better load balancing: Real-time dynamic subsetting. https://www.uber.com/en-HK/blog/better-load-balancing-real-time-dynamic-subsetting/, 2024.

[42] Load balancing: Handling heterogeneous hardware. https://www.uber.com/en-DO/blog/load-balancing-handling-heterogeneous-hardware/, 2024.

[43] Peter Ward, Paul Wankadia, and Kavita Guliani. Reinventing backend subsetting at google: Designing an algorithm with reduced connection churn that could replace deterministic subsetting. *Queue*, 2022.

[44] Peter Ward, Paul Wankadia, and Kavita Guliani. Reinventing backend subsetting at google. *Communications of the ACM*, 66(5):40–47, 2023.

[45] Wu Xiang, Yakun Li, Yuquan Ren, Fan Jiang, Chaohui Xin, Varun Gupta, Chao Xiang, Xinyi Song, Meng Liu, Bing Li, et al. Gödel: Unified large-scale resource management and scheduling at bytedance. In *Proc. of ACM SoCC*, pages 308–323, 2023.

[46] Zhe Xie, Changhua Pei, Wanxue Li, Huai Jiang, Liangfei Su, Jianhui Li, Gaogang Xie, and Dan Pei. From point-wise to group-wise: A fast and accurate microservice trace anomaly detection approach. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[47] 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 ASPLOS*, 2021.

[48] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proceedings of the VLDB Endowment*, pages 1393–1404, 2014.

[49] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of EuroSys*, 2020.