

# TraDE: Network and Traffic-Aware Adaptive Scheduling for Microservices Under Dynamics

Ming Chen , Muhammed Tawfiqul Islam , Maria Rodriguez Read , and Rajkumar Buyya , *Fellow, IEEE*

**Abstract**—The transition from monolithic architecture to microservices has enhanced flexibility in application design and its scalable execution. This approach typically uses a computing cluster managed by a container orchestration platform to deploy microservices. However, this shift introduces significant challenges, particularly in the efficient scheduling of containerized services. These challenges are compounded by unpredictable scenarios such as dynamic incoming workloads with various execution traffic and variable communication delays among cluster nodes. Existing works often overlook the real-time traffic impacts of dynamic requests on running microservices, as well as the varied communication delays across cluster nodes. Consequently, even optimally deployed microservices could suffer from significant performance degradation over time. To address these issues, we propose a network and traffic-aware adaptive scheduling framework, TraDE, which can adaptively redeploy microservice instances to maintain desired performance amid changing traffic and network conditions within the hosting cluster. We have implemented TraDE as an extension to the Kubernetes platform. Additionally, we deployed realistic microservice applications in a real compute cluster and conducted extensive experiments to assess our framework’s performance in various scenarios. The results demonstrate the effectiveness of TraDE in rescheduling running microservices to enhance end-to-end performance while maintaining a high goodput ratio. Compared with the existing method NetMARKS, TraDE outperforms it by reducing the average response time of the application by up to 48.3%, and improving the throughput by up to 1.2–1.5  $\times$  across workloads while maintaining a goodput ratio of 95.36%, and showing robust adaptive capability to meet QoS targets under sustained workloads and dynamic networking conditions.

**Index Terms**—Microservice, scheduling, performance optimization.

## I. INTRODUCTION

IN THE pervasive cloud computing domain, microservices have emerged as a key architecture, revolutionizing how cloud-based applications are designed and implemented. Characterized by the modular and decentralized design, microservices provide good flexibility and scalability, catering to the

dynamic demands of modern cloud-deployed applications [1], [2], [3]. However, this transition also introduces significant challenges in microservice management and performance optimization, particularly microservices with complex workflow and dependencies [4], [5], [6], [7]. Uncertainty in user requests, combined with varying execution paths across microservices and differing communication delays between nodes, significantly increases the difficulty of ensuring Quality of Service (QoS) compliance for microservices deployed on cluster nodes. In cloud environments, numerous microservices are co-located, interact with each other, and are often shared among various applications [8], [9]. This can lead to worsened performance degradation due to resource contention and cascading delays in the execution paths of requests. These issues directly affect latency and throughput, both critical for meeting QoS targets.

Although existing works have proposed different methods to improve the running performances of containerized microservices, there are still some limitations to these methods. Opt-Traffic [10] optimizes the traffic transmission of containerized microservices across cluster nodes, which fails to consider the cross-node delays under dynamic networking traffic and also introduces complexity by calculating every dependent container replica pair. NetMARKS [11] determines Kubernetes pod scheduling by the dynamic network metrics collected by Istio Service Mesh [12], which have limitations on analyzing the bidirectional metrics between dependent microservices and also may introduce imbalanced load distributions across the cluster nodes. Other existing methods [13], [14], [15], [16] also have similar limitations on tackling dynamic workloads, the awareness of cross-node delays, and imbalanced load distributions in the cluster.

To solve the aforementioned challenges, this paper proposes TraDE, a novel framework that utilizes a traffic and network-aware rescheduling approach. TraDE is designed to adaptively redeploy containerized microservices within the cluster by analyzing real-time traffic stress between dependent microservices along with the variations of cross-node delays. By doing so, it seeks to mitigate QoS target violations amid fluctuating user requests and network variations. To implement and evaluate the proposed TraDE framework, this paper seeks to address several crucial challenges in dealing with network dynamics for meeting application services’ QoS targets: (1) How to quantitatively map dynamic bidirectional traffic patterns into traffic stress between upstream and downstream microservices, including all the corresponding replicas?, (2) What approach should be employed to build the traffic stress graph under dynamic workloads

Received 31 October 2024; revised 14 October 2025; accepted 23 October 2025. Date of publication 28 October 2025; date of current version 20 November 2025. This work was supported in part by the Australian Research Council (ARC) grants: Discovery Project under Grant DP240102088 and in part by Linkage Infrastructure, Equipment and Facilities under Grant LE200100049. Recommended for acceptance by J. Lofstead. (*Corresponding author: Ming Chen.*)

The authors are with the Quantum Cloud Computing and Distributed Systems (qCLOUDS) Laboratory, School of Computing and Information Systems, University of Melbourne, Parkville, VIC 3010, Australia (e-mail: mingc4@student.unimelb.edu.au; tawfiqul.islam@unimelb.edu.au; maria.read@unimelb.edu.au; rbuyya@unimelb.edu.au).

Digital Object Identifier 10.1109/TPDS.2025.3626424

within a specific time interval?, (3) How to design a controllable network-dynamics manager to thoroughly evaluate the proposed method via efficiently injecting dynamic cross-node delays and accurately measuring the node-delay matrix?, (4) With the constructed traffic-stress graph and the measured cross-node delay matrix, how to determine the service-to-node mapping under minimal exploration time and a balanced load goal?, (5) With the explored service-to-node mapping result, what strategies should be adopted to ensure zero downtime of the running services when migrating related microservice containers?

The proposed TraDE framework resolves these challenges and could adapt to the changing traffic conditions and redeploy the running microservice containers to server nodes when QoS violation is detected. Specifically, TraDE builds a traffic stress graph for dependent microservices, a lightweight cross-node delay monitor, a low overhead service-node mapper, and a microservice rescheduler with guarantees of service availability when migrating containers. We demonstrate the effectiveness of our approach through extensive evaluation using practical microservice applications. The results of our experiments, conducted under a variety of scenarios, demonstrate the ability of TraDE to maintain the desired QoS target of deployed microservice applications when QoS violations happen. In summary, our main contributions can be summarized as follows.

- We propose a traffic analyzer that dynamically constructs a traffic stress graph. This graph not only illustrates the latest microservice call graphs (dependencies) but also identifies the microservice pairs experiencing higher stress via bidirectional traffic analysis.
- We design and implement a network-dynamics manager which mainly consists of injecting customized cross-node delays in a controllable way via packet-level tagging and also accurately measuring the node communication delays via cluster-level daemon agents.
- We introduce a parallel algorithm for service-node mapping to minimize the total traffic transmission overhead with guaranteed balanced task chunks and fast convergence.
- We design a microservice rescheduler to migrate microservice instances that experience QoS violations. Specifically, when migrating the microservices, we employ various scheduling schemes to guarantee service availability and resource availability in the cluster.
- We develop a prototype system of the TraDE framework as an extension to the Kubernetes platform, and demonstrate system performance in a real computing cluster.

*What is new compared to prior work?* TraDE (i) models *bidirectional* traffic to expose overloaded pairs as rescheduling targets; (ii) introduces a *controllable*, destination-specific delay generator paired with a multi-agent measurer for accurate cross-node latency under dynamics; and (iii) couples these with a service-node mapper that embeds an overload penalty and preserves availability during service migrations. Together, these enable fast, practical, and repeatable QoS-target compliance under changing traffic and network conditions.

The rest of the paper is structured as follows: Section II discusses related work, providing a comprehensive background on existing methods. The motivation and problem statement

are introduced in Section III. Section IV presents the proposed TraDE framework, explaining its main components. Section V focuses on the design of the traffic analyzer. Section VI details the design of the dynamics manager, which consists of the dynamic delay generator and the cross-node delay measurer. Section VII introduces the proposed PGA algorithm for microservice placement, along with an overhead analysis. Section VIII offers a performance evaluation and analysis of the proposed TraDE framework. Finally, Section IX concludes the paper and discusses future work.

## II. RELATED WORK

The scheduling of microservices in cluster environments has been extensively studied from various perspectives. This section reviews the existing literature, focusing on microservice management, graph analysis, and network-aware scheduling.

### A. Microservice Management

FIRM [17] leverages online telemetry metrics data and machine learning-based models to manage microservices in a fine-grained way by localizing the SLO violations, identifying resource contentions, and then taking reprovision measures to mitigate the SLO violations. Erms [8] builds resource scaling models to calculate the latency objectives for shared microservices with large calling graphs. GrandSLAM [18] estimates the completion time of the requests for individual microservice execution stages and leverages the estimated time to batch and reorder the requests dynamically. However, these existing methods have limitations in considering the dynamic impacts of cross-node communication delays and the changing bidirectional traffic between upstream and downstream microservices with multiple replicas.

### B. Graph Analysis

Sage [19] builds a graphical-based Bayesian model to analyze the root cause of cascading QoS violations for interactive microservices focusing on practicality and scalability. Tian et al. [20] develop a workload generator to synthesize the DAG jobs with graph workflow by analyzing large-scale cluster traces. Furtherly, Luo et al. [21] characterize the call graph of dependent microservices by analyzing Alibaba cluster data and reveal three types of calling dependency graphs for microservice applications. Parslo [22] introduces a gradient descent-based method by breaking the end-to-end SLO budget into smaller units to assign partial SLOs among nodes in a microservice graph under an end-to-end latency SLO. However, these methods are generally time-consuming with high overheads to build the graph and not suitable for dynamic incoming user requests.

### C. Network-Aware Scheduling

NetMARKS [11] introduces a network-aware approach to schedule the Kubernetes pods from different 5G edge applications by using the collected dynamic metrics from Istio Service Mesh. Marchese et al. [16] introduce a network-aware scheduling extension for the default Kubernetes scheduler by considering the infrastructure network conditions and the interactions

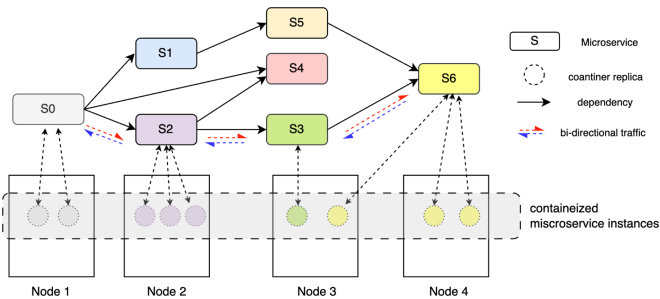


Fig. 1. An envisioned workflow of containerized microservice executions and communications in a cluster with four nodes.

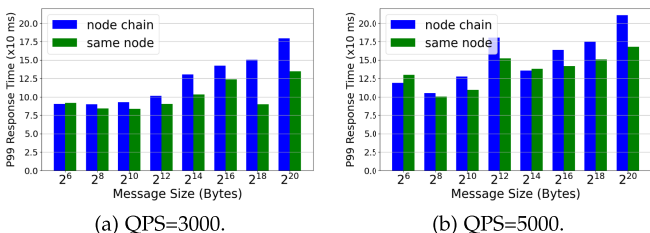


Fig. 2. P99 response time of different QPS from an *upstream* microservice client and a *downstream* microservice server under scenarios where they are either colocated on the same node or running across a node chain.

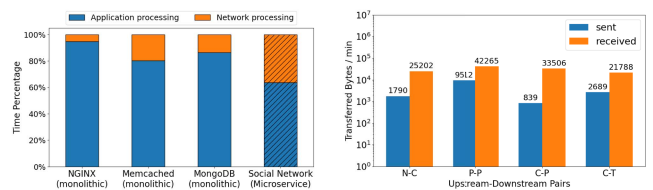
among microservices. OptTraffic [10] develops a network-aware scheduling framework by optimizing the cross-machine traffic scheduling for multi-replica microservice containers by migrating the containers with dependent relations. However, these networking-aware scheduling schemes designed for microservices still have limitations on bidirectional traffic analysis for dependent microservice replicas and the changing infrastructure-level conditions, i.e., varied cross-node communication delays.

### III. MOTIVATION AND PROBLEM STATEMENT

#### A. Background

An increasing number of modern cloud applications have evolved into microservice-based architectures, which manage applications through a collection of containerized, loosely coupled, fine-grained services [21], [23]. As discussed in previous work [23], transitioning from monolithic designs to microservice designs for cloud applications leads to a higher proportion of processing time being spent in the networking stack compared to monolithic applications. The processing time percentages of different cloud applications are shown in Fig. 3(a).

Deployed as a set of containerized microservice instances, the application processes user requests through dependent microservice replicas and returns the processed results in the reverse direction. As shown in Fig. 1, a microservice-based application is decoupled into a collection of containerized microservices along with the bidirectional traffic flows between dependent microservices. Every microservice is supported by one or multiple corresponding container instances to provide the application functionality. These container instances launch from pre-defined



(a) Processing comparisons of monolithic and microservice applications (data source [23]). (b) Bidirectional traffic between UM-DM pairs in Social Network in one minute.

Fig. 3. (a) Comparison of processing time percentage between monolithic and microservice applications. (b) Traffic of different UM-DM pairs in one minute.

container images specified in the corresponding microservice deployment file.

#### B. Motivation

We deployed synthetic microservice applications in a running cluster to motivate our design and tested them under different scenarios. The deployed applications include a server/client application and a benchmark application. Fig. 2 shows the performance of two dependent microservices: one functions as a client container sending `PUT` requests with varying message sizes from  $2^6$  Bytes to  $2^{20}$  Bytes, and the other operates as the corresponding server container to receive the requests. From Fig. 2(a) and (b), we observe that colocating two dependent microservice containers on the same node notably improves the p99 response time. Additionally, the QPS (Queries Per Second) has a significant impact on dependent microservice containers. As shown in Fig. 2, when QPS increases from 3000 to 5000, the p99 response times for each transferred message size also increase, with a maximum 78.2% performance degradation when the sent message size is  $2^{12}$  Bytes. This implies that, for an *upstream-downstream* pair, the transferred message size, QPS, and cross-node communications have notable impacts on end-to-end performance.

Additionally, we deployed the Social Network benchmark released with DeathStarBench [23] to quantify the transmitted traffic difference between dependent UM-DM pairs. Under different request workloads, we observed notable traffic differences between different UM-DM pairs.

As shown in Fig. 3(b), there are significant differences between sent and received traffic for certain UM-DM pairs, including N-C (Nginx  $\rightarrow$  Compose-post), P-P (Post-storage-service  $\rightarrow$  Post-storage-MongoDB), C-P (Compose-post  $\rightarrow$  Post-storage-service) and C-T (Compose-post  $\rightarrow$  Text-service) [23]. In our testing scenario, for example, the bidirectional traffic between the C-P pair shows that the received traffic could be approximately 40x more than the sent traffic, implying that a slight delay increase in transmission among the C-P pair could degrade the pair's end-to-end performance by up to 40x. Thus, it is significant to analyze the real-time bidirectional traffic for dependent microservices.

From the above observations, we can conclude that: (1) Unlike monolithic cloud applications, microservice-based applications spend significantly more time on the networking processing

stack. (2) Cross-node communication can impact the performance of microservice pairs, particularly for pairs with high volumes of traffic transmission. (3) For dependent *upstream* and *downstream* microservice pairs, the transmitted bidirectional (sent and received) traffic can exhibit significant differences, such as small requests but large payloads.

### C. Problem Definition

In dynamic networking and traffic environments, the deployment of microservice instances across different server nodes plays a critical role in determining the end-to-end performance of distributed applications. In particular, poor placement decisions may incur high communication overhead due to inter-node delays and traffic volumes. To address this, we formulate the optimization task as a *Service-Node Mapping Problem*, where the objective is to assign a set of microservices to a set of server nodes in a manner that minimizes overall communication cost—capturing both communication latency and traffic-induced overhead—while satisfying server resource constraints.

Let  $M = \{m_1, \dots, m_k\}$  be the set of microservices,  $N = \{n_1, \dots, n_p\}$  the set of server nodes, and  $P : M \rightarrow N$  the service–node mapping. Over a measurement window  $\Delta t$ , let  $T_{i \rightarrow j} \geq 0$  and  $T_{j \rightarrow i} \geq 0$  denote the forward and reverse traffic (bytes or rate) between  $m_i$  and  $m_j$ . Let  $D_{a,b} \geq 0$  be the one-way (ping-like) delay from node  $a$  to node  $b$  (not necessarily symmetric). We use direction weights  $w_f, w_b \in [0, 1]$  (default  $w_f = w_b = 0.5$ ).

The pairwise communication cost for  $(i, j)$  under a placement  $P$  is

$$\begin{cases} C_{i,j}(P) = w_f T_{i \rightarrow j} D_{P(m_i), P(m_j)} \\ \quad + w_b T_{j \rightarrow i} D_{P(m_j), P(m_i)}, \\ (w_f, w_b) \in [0, 1]^2. \end{cases} \quad (1)$$

and the total communication cost aggregates pairwise costs:

$$\text{TotalCost}(P) = \sum_{i=1}^k \sum_{j=1}^k C_{i,j}(P). \quad (2)$$

Each node  $n \in N$  has a capacity vector  $C_n$  (e.g., CPU, memory, GPU), and each microservice  $i$  has a resource demand vector  $R_i$ . A placement is feasible if, element-wise,

$$\sum_{i: P(m_i)=n} R_i \leq C_n \quad \forall n \in N. \quad (3)$$

*Penalty form used in our solver:* We enforce (3) via a soft penalty, consistent with our implementation. Let  $\Phi(P) = \sum_{n \in N} \max\{0, \sum_{i: P(i)=n} R_i - C_n\}$  denote the aggregated overflow (applied element-wise and summed across resources), and let  $\lambda > 0$  be a large coefficient. The penalised objective is

$$\min_P \text{TotalCost}(P) + \lambda \Phi(P), \quad (4)$$

which is equivalent in practice to the constrained form for sufficiently large  $\lambda$ .

*Why bidirectional traffic?* An RPC is typically a request–response round trip; even with one-way (ping-like) delays, both directions (forward  $i \rightarrow j$ , return  $j \rightarrow i$ ) contribute

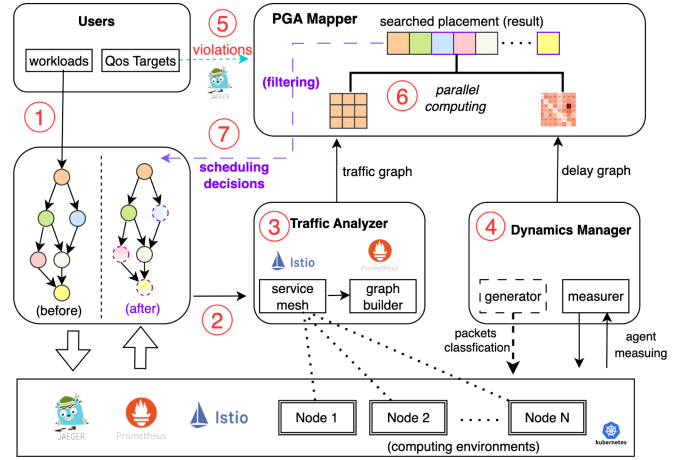


Fig. 4. The proposed TraDE framework.

to user-perceived time and to total network time over  $\Delta t$ . Equation (1) therefore weights both directions. When one-way delays are symmetric ( $D_{a,b} = D_{b,a}$ ), (1) reduces to  $(w_f T_{i \rightarrow j} + w_b T_{j \rightarrow i}) D_{P(m_i), P(m_j)}$ ; setting  $(w_f, w_b) = (1, 0)$  yields the one-direction variant.

*Implementation in TraDE:* In our prototype system,  $T_{i \rightarrow j}$  and  $T_{j \rightarrow i}$  are computed from Istio byte counters over  $\Delta t$ ; we take  $w_f = w_b = 0.5$  by default and optimise (4) using a parallel greedy search.

The problem is thus to find a mapping  $P$  that minimizes (2) (or equivalently (4)) subject to (3), promoting traffic locality and reducing cross-node latency under dynamic workloads and network conditions.

## IV. FRAMEWORK OF TraDE

Based on our observations, we were motivated to design TraDE, a network and traffic-aware rescheduling framework for containerized microservices when the deployed service experiences QoS violations due to dynamic requests and varying cross-node communication delays in dynamic computing environments. As shown in Fig. 4, the figure illustrates the main modules and how each module works together at different stages to complete the adaptive scheduling process. The key modules of the proposed framework are as follows:

- *Traffic Analyzer:* This module consists of a service mesh and a graph builder, which are designed for analyzing the real-time bidirectional traffic flows between the dependent *upstream* microservice containers (including all corresponding replicas) and *downstream* microservice containers through the service mesh and the proposed graph builder algorithm.
- *Dynamics Manager:* This module consists of a *generator* and a *measurer* to manage the cross-node communication delays in the computing environments. The *generator* is designed to generate different practical delays to the computing nodes to validate the proposed TraDE through packet-level manipulation. The *measurer* is designed to measure the communication delays between different

nodes across the computing environments through multiple daemon agents.

- *PGA Mapper*: This module is designed to tackle QoS violations of the running applications. Specifically, when there are violations to the predefined QoS targets, the PGA (Parallel Greedy Algorithm) mapper computes the service placements on the computing nodes to achieve the lowest overhead as defined in (2).

As shown in Fig. 4, at the beginning stage ①, users define the QoS targets and send different workloads (i.e., different types of requests and QPS) to the deployed microservice applications. At stages ② and ③, the performance metrics of the running microservices are collected by Jaeger and Istio. Within a predefined monitoring time interval, the traffic analyzer analyzes the bidirectional traffic between dependent microservices and builds the traffic graph, which is then sent to the PGA mapper. Meanwhile, at stage ④, the node dynamics manager measures the cross-node communication delay graph, which is sent as another graph to the PGA mapper. As the microservice application runs, if there are any QoS violations (at stage ⑤) to the predefined QoS targets, TraDE is triggered to run the PGA mapper. As shown at stage ⑥, the PGA mapper finds the service-node placement through a constructed traffic graph and delay graph from the traffic analyzer and dynamics manager, respectively. Designed to run in a parallel computing manner to speed up the process, the PGA mapper provides the searched placement result, specifying the placement of each microservice to a node. Placement results are filtered before being used as the scheduling decision, as some microservices are already placed in the optimal node, and some microservice instances, such as Jaeger agent instances, are not supposed to be migrated. At the final stage ⑦, the filtered placement results are taken as the adaptive scheduling decision in response to the current dynamic computing environments.

## V. DESIGN OF TRAFFIC ANALYZER

### A. Service Mesh

The key aspect of implementing the Traffic Analyzer for TraDE lies in efficiently and minimally analyzing the bidirectional traffic between dependent container pairs, i.e., the upstream and downstream microservice containers. When there is only one microservice instance for both upstream and downstream services, it is straightforward to collect and analyze bidirectional traffic metrics from the Linux *proc* file system. However, when multiple replicas exist for either the downstream or upstream microservices, analyzing the bidirectional traffic between all upstream and downstream microservice replicas becomes time-consuming. Thus, we implemented the service mesh to better observe and manage the complex traffic flows.

1) *Istio Service Mesh Implementation*: To obtain finer-grained metrics of bidirectional traffic between upstream microservices and their downstream counterparts, along with their corresponding replicas, we implemented *Traffic Analyzer* by analyzing traffic metrics with Istio Service Mesh [12]. The service mesh is designed as a dedicated infrastructure layer that can be added to containerized microservice applications. With

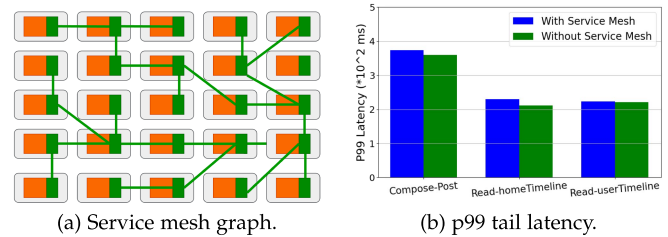


Fig. 5. (a) An overview of how service mesh with sidecar containers (green) works with microservice containers (orange). (b) Overhead comparisons of p99 tail latency of different workloads to Social Network applications with and without service mesh.

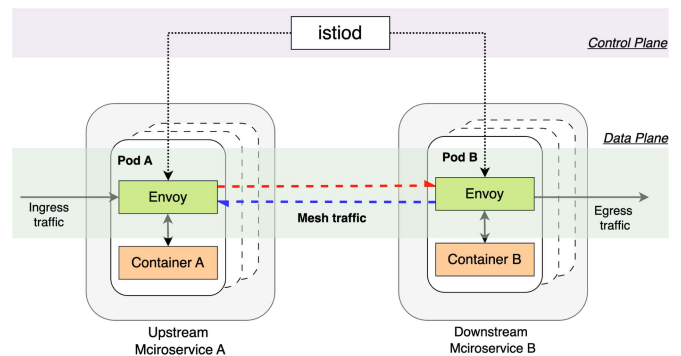


Fig. 6. An architecture of how upstream microservice A interacts with downstream microservice B with Istio service mesh enabled.

the service mesh deployed, the traffic for each microservice container is proxied by an injected sidecar container.

In a Kubernetes-based cluster with the deployed Istio service mesh, a microservice pod comprises both the application and sidecar containers. As depicted in Fig. 5(a), the dedicated service mesh structure proxies the ingress and egress traffic among the microservice containers. The orange rectangles represent the containerized microservices, and the green rectangles represent the sidecar containers, which function as a dedicated layer managing traffic among various microservices. The green links illustrate the data traffic connections and dependency relationships between microservices. Specifically, as depicted in Fig. 6, the Envoy [24] containers (acting as sidecar containers) in Pod A and Pod B proxy the bidirectional traffic between Container A and Container B. Within each pod, the Envoy container communicates with its corresponding microservice containers.

2) *Overhead Analysis*: We realize that adding an additional service mesh layer to the microservice application may introduce extra delays in the traffic flow. However, the introduced delay only contributes a slight proportion to tail latency. Based on [25], within the Istio 1.21.2 service mesh, utilizing telemetry v2, each request is processed by both a client-side and server-side Envoy proxy. These proxies collectively increase latency at the 90th percentile by approximately 0.182 milliseconds and at the 99th percentile by about 0.248 milliseconds, compared to the baseline latency of the data plane. These findings were conducted with a 1 kB payload, a rate of 1000 requests per second, and varying

client connections (2, 4, 8, 16, 32, 64) at the CNCF Community Infrastructure Lab [26].

Additionally, we evaluated and compared the performance of the Social Network [23] with and without the implemented service mesh. We separately deployed the Social Network application in two different namespaces to guarantee the requests and services would not interfere with each other. Three types of requests (i.e., Compose-Post, Read-homeTimeline, and Read-userTimeline) were sent as the workloads to the deployed Social Network application, each with a QPS of 200. As depicted in Fig. 5(b), it can be observed that the service mesh introduces only minimal delays. Therefore, the service mesh layer does not add significant overhead to the network traffic of the microservice application. As the tradeoff to better manage the complex traffic flows, the introduced minimal delays are acceptable.

### B. Traffic Stress Graph

One of the main goals of the proposed TraDE is to construct real-time traffic flows along the triggered call graphs of the deployed microservices. Multiple request types and varied QPS would trigger different structures of call-graph with different amounts of traffic flows, which complicates the adaptive scheduling of microservices in response to QoS violations. In our work, we define the following terminology.

*Stress Element*: Stress Element (SE) defines the dependencies between two dependent microservices, i.e., the UM (Upstream Microservice) and DM (the Downstream Microservice), and the traffic stress between the two corresponding microservices. The traffic stress of a SE is calculated by the average traffic of *sent* and *received* in a given time interval, as we believe that the traffic of *sent* and *received* between the dependent microservices both contribute significant stress to the end-to-end performance of the deployed application. Written mathematically, the stress of a Stress Element can be expressed by:

$$\text{stress}_\sigma^\mu(\mu^{UM}, \sigma^{DM}, \Delta t) = \frac{\text{Bi-direction\_traffic}(\mu^{UM}, \sigma^{DM})}{2\Delta t} \quad (5)$$

In (5),  $\mu$  and  $\sigma$  refer to the dependent upstream and downstream microservices.  $\Delta t$  is to define the measurement time interval for microservices  $\mu$  and  $\sigma$ , and  $\text{Bi-Direction\_traffic}(\mu^{UM}, \sigma^{DM})$  measures the bidirectional traffic transmitted between  $\mu$  and  $\sigma$  during time interval  $\Delta t$ . Thus, a Stress Element can be referred to as  $SE(\mu^{UM}, \sigma^{DM}, \text{stress}_\sigma^\mu)$ . In our implementation, TraDE analyzes the metrics data retrieved from `istio_tcp_sent_bytes_total` and `istio_tcp_received_bytes_total`,<sup>1</sup> which respectively measures the size of total bytes sent during response and the size of total bytes received during request.

Before analyzing the traffic flow patterns among microservices, it is essential to build a stress graph that represents the dependencies between microservices and the associated stress on those dependencies. Additionally, sorting the microservice pairs from the constructed traffic graph is important for efficient

---

### Algorithm 1: Build Traffic Stress Graph.

---

```

1: Input: List of Stress_Elements with SE objects
2: Output: A traffic Graph with dependencies and stress
3:  $MS \leftarrow \text{Stress\_Elements}$   $\triangleright$  list of deployed
   microservices
4: Initialize  $Graph \leftarrow \text{zeros}(|MS| \times |MS|)$ 
5: for each  $\mu$  in  $MS$  do  $\triangleright$  upstream microservices
6:   for each  $\sigma$  in  $MS$  do  $\triangleright$  downstream microservices
7:     if  $\mu \neq \sigma$  then
8:        $\text{stress} \leftarrow \text{Bi\_traffic}(\mu, \sigma, \Delta t)$ 
9:        $Graph[\mu][\sigma] \leftarrow \text{stress}$ 
10:    end if
11:  end for
12: end for
13: return  $Graph$ 

```

---

service-node mapping in subsequent sections. Therefore, we propose algorithms for constructing a traffic stress graph and sorting the microservice pairs based on the constructed stress graph.

1) *Building the Traffic Stress Graph*: Algorithm 1 constructs the traffic stress graph,  $Graph$ , using the stress elements defined by (5). The algorithm begins by retrieving a list of deployed microservices at the current monitoring time. It then initializes the  $Graph$  with zeros for each row and column. In the next step, it iterates through all deployed microservices and calculates the traffic stress between each pair of microservices over a given time interval. Once the matrix iteration is complete, the traffic stress graph is obtained.

2) *Sorting Microservice Pairs by Stress Level*: After constructing the traffic stress graph, the next step is to identify the stress level among the microservice pairs in the stress graph. Algorithm 2 is responsible for sorting all the microservice pairs with traffic values from the stress graph  $Graph$  in descending order to identify the microservice pairs with the higher stress and also the pairs with lower stress, which will be used for designing scheduling policies in the proposed TraDE. The microservice pairs with higher stress are the pairs that contribute more to the total communication cost in (2). A quicker localization of the microservice pairs under higher stress would be good for a quicker convergence to find the satisfied microservices ( $M$ ) to nodes ( $N$ ) mapping ( $P : M \rightarrow N$ ).

## VI. DESIGN OF DYNAMICS MANAGER

### A. Dynamic Delay Generator

*Customized Delay Generation*: Injecting different cross-node delays to the cluster nodes is crucial for evaluating the effectiveness of TraDE. However, implementing various communication delays from one source node to multiple destination nodes poses significant challenges. To the best of our knowledge, no existing work has proposed a method to inject customized communication delays in a controllable manner. Some related works [17], [27], [28], [29], [30] mention delay injection to server nodes, but these works either use a static delay matrix

<sup>1</sup>Different version of Istio service mesh may use different standard metrics.

**Algorithm 2:** Sort Microservice Pairs by Traffic Stress.

---

```

1: Input: Traffic Stress Graph  $Graph$  (traffic stress matrix)
2: Output: Sorted list of microservice pairs by traffic stress
3: Initialize  $pairs \leftarrow \{\}$   $\triangleright$  empty list of pairs
4: for each  $\mu$  in  $Graph$  do
5:   for each  $\sigma$  in  $Graph[\mu]$  do  $\triangleright$  corresponding DM
6:     if  $Graph[\mu][\sigma] > 0$  then
7:       Append  $(\mu, \sigma, Graph[\mu][\sigma])$  to  $pairs$ 
8:     end if
9:   end for
10: end for
11: SortPairs( $pairs$ )  $\triangleright$  sorting in descending order
12: return Sorted  $pairs$ 

```

---

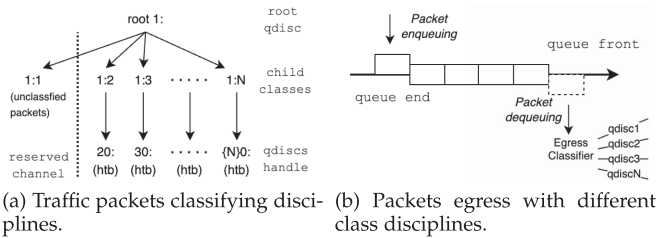


Fig. 7. (a) Unclassified packets (not sent to certain destinations) are transmitted through a reserved channel without injected delays, while classified packets are assigned different delays for different destinations. (b) Classified packets are sent to different destinations with varying delays.

or implement uniform communication delays for every egress traffic packet on the node’s network interface. This approach leads to two main limitations in evaluation experiments: (1) Uniform communication delays from one source node to all destination nodes prevent distinguishing differences between node pairs, as delays from one source node to all other nodes share the same settings; (2) All other networking services not involving correlated nodes are degraded because the injected delays affect all egress traffic.

To address these limitations in current research, as shown in Fig. 7(a), we propose a customized delay injection scheme that classifies packets using filters to differentiate egress packets and distribute them based on their IP destinations. Additionally, an extra channel is reserved for default packet transmission without injected delays, ensuring that the performance of other services is not affected. We implemented this scheme using the `tc` networking tool and `htb` (Hierarchical Token Bucket).

### B. Cross-Node Delay Measurer

In a large-scale computing cluster with high traffic, communication delays among infrastructure nodes are not negligible, as microservice applications are decoupled and distributed across different connected computing nodes. When significant delays occur among cluster nodes, the communication among microservice containers running on these nodes will be negatively impacted. Consequently, some microservices may experience QoS violations.

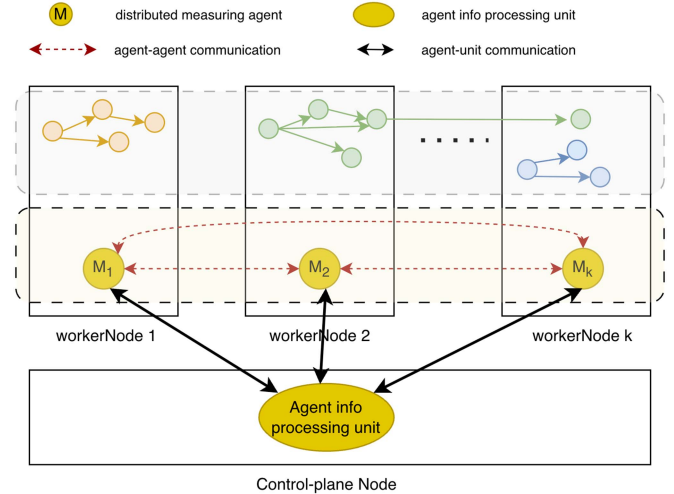


Fig. 8. The implementation of *Cross-node Delay Measurer* across cluster nodes.

*Design of Delay Measurer:* To address this problem, we designed a lightweight module called the *Cross-node Delay Measurer* for our proposed `TRADE` framework. The module consists of a centralized information processing unit and multiple distributed measuring agents. To keep the module lightweight, the processing unit maintains minimal connections with all measuring agents and summarizes the measured communication delays. The measuring agents run as lightweight containers, maintaining only a simple communication function for measuring delays. Additionally, to enhance the module’s robustness during cluster-level upgrades or downgrades, an auto-scaling mechanism for the distributed measuring agents is implemented. Specifically, when the cluster adds more nodes, a measuring agent is automatically added to the new nodes to maintain the consistency and accuracy of communication delay measurements across the cluster. Similarly, when the cluster removes nodes, the corresponding measuring agents are automatically removed as well.

*Implementation of Delay Measurer:* We implemented an efficient, lightweight, and auto-scalable cluster-level measuring scheme that analyzes cross-communication delays across infrastructure nodes. As shown in Fig. 8, we introduced the design schemes of the *Cross-node Delay Measurer*, consisting of an agent information processing unit and a set of distributed measuring agents.

Specifically, we designed the agent information processing unit to operate as a running plugin on the control-plane node, collecting measured cross-node delay information from all active agents via TCP messages. Meanwhile, the distributed measuring agents are implemented as a group of interconnected running pods, managed by a `DaemonSet` deployment with a lightweight pre-configured image. In this way, the measuring agents on worker nodes continuously measure cross-node delays with each other and send the measured information to the information processing unit plugin on the control-plane node. Additionally, when the number of cluster nodes changes, the agents are automatically added when a new node joins or deleted when

TABLE I

INJECTED DELAY VERSUS MEASURED CROSS-NODE COMMUNICATION DELAY (MS). IN EACH CELL, THE FIRST NUMBER REPRESENTS THE INJECTED DELAY, AND THE SECOND NUMBER REPRESENTS THE MEASURED ACTUAL DELAY. THIS TABLE DEMONSTRATES THE SUCCESSFUL INJECTION OF DIFFERENT DELAYS FROM ONE SOURCE NODE TO VARIOUS DESTINATION NODES, AS WELL AS THE RESERVED CHANNEL FOR DESTINATIONS (I.E., CONTROL-PLANE NODE AND GOOGLE.COM) WITHOUT INJECTED DELAYS.

Source Node	Destinations										
	Node1	Node2	Node3	Node4	Node5	Node6	Node7	Node8	Node9	Control-plane Node	google.com
Node1	- / -	3.00 / 3.21	8.00 / 8.51	10.00 / 11.01	14.00 / 14.21	6.00 / 6.73	27.00 / 28.98	13.00 / 13.89	21.00 / 22.31	- / 0.64	- / 14.10
Node2	8.00 / 8.22	- / -	4.00 / 4.52	13.00 / 14.02	14.00 / 15.21	18.00 / 19.02	38.00 / 38.22	31.00 / 32.13	29.00 / 31.02	- / 0.89	- / 14.20
Node3	4.00 / 4.03	12.00 / 12.37	- / -	8.00 / 9.05	18.00 / 18.66	11.00 / 11.89	4.00 / 5.75	12.00 / 13.41	15.00 / 16.89	- / 0.43	- / 14.20
Node4	17.00 / 17.04	15.00 / 15.11	7.00 / 7.21	- / -	5.00 / 5.87	6.00 / 6.79	22.00 / 23.15	13.00 / 13.76	25.00 / 26.82	- / 0.51	- / 14.30
Node5	20.00 / 21.02	12.00 / 12.23	11.00 / 11.68	10.00 / 11.27	- / -	9.00 / 9.73	7.00 / 7.13	4.00 / 4.11	9.00 / 9.23	- / 0.45	- / 14.10
Node6	17.00 / 17.12	26.00 / 26.06	18.00 / 18.87	16.00 / 17.08	6.00 / 6.11	- / -	5.00 / 5.71	10.00 / 10.19	5.00 / 5.86	- / 0.53	- / 14.20
Node7	20.00 / 20.96	10.00 / 10.52	10.00 / 10.91	9.00 / 9.86	11.00 / 12.92	5.00 / 5.67	- / -	5.00 / 5.71	9.00 / 9.39	- / 0.45	- / 14.10
Node8	21.00 / 21.11	25.00 / 25.72	4.00 / 4.53	10.00 / 10.71	12.00 / 12.73	15.00 / 16.02	10.00 / 10.13	- / -	6.00 / 7.08	- / 0.34	- / 14.10
Node9	36.00 / 36.93	22.00 / 22.70	40.00 / 40.39	9.00 / 10.08	25.00 / 25.69	8.00 / 8.16	7.00 / 7.23	6.00 / 6.18	- / -	- / 0.30	- / 14.30

a node drains from the existing cluster through the DaemonSet deployment mechanism. With this mechanism, the *Cross-node Delay Measurer* ensures that each node will have a dedicated pod for delay measurement, regardless of changes in the number of cluster nodes.

Fig. 8 shows how the distributed measuring agents communicate with each other and how the agent information processing unit interacts with each measuring agent container.

*Overhead analysis of Delay Measurer:* Regarding the overhead of delay measurement in a real system, we optimized the measurement using parallel processing. The latency measurement tasks are designed to run concurrently, and the results are aggregated into a latency results dictionary, completing the process in just a few seconds. Additionally, the distributed measuring agents are lightweight (about 0.2 MiB) and stable (running over 6 months without any failure). Each node runs a measuring agent developed from the `curlimages/curl` image, consuming around 0.2 MiB of memory per node. Thus, the cluster-level delay measurement consumes minimal memory and completes within a few seconds.

### C. Effectiveness of Dynamics Manager

Injecting different communication delays from one source node to various destination nodes and accurately measuring these delays is challenging. By adopting the proposed injection scheme shown in Fig. 7, we injected different delays to various destination worker nodes and measured the actual communication delays from the source node to these destinations. As shown in Table I, the measured delays in each cell exhibit high accuracy, with only around a 1 ms difference, which can be attributed to the randomness of data packet transfers in cloud networking stacks. Additionally, to demonstrate that other traffic is not influenced, we tested the communication delays from all worker nodes to other destinations. One destination is the control-plane node in the same cluster, and the other is the Google host (google.com), showing average delays of 0.50 ms and 14.20 ms, respectively.

From these measured data, we can confirm the effectiveness of the proposed schemes and implementation for the *Dynamics Manager* in TraDE. It should be noted that the delay injection scheme is optional and can be switched off when TraDE is deployed in actual computing environments. The primary design aim of the customized delay injection scheme is to generate various dynamics to validate and evaluate our proposed adaptive scheduling framework TraDE. Additionally, the proposed

```

ParallelPlace(T, D, P, res, cap, workers)
|-- Sorted_MS_pairs(T)
|-- for each chunk task:
|   |-- PlaceWorker(T, D, P, res, cap, tasks)
|   |   |-- CalcCost(T, P, D, res, cap)
|   |   |-- CalcCost(T, new_P, D, res, cap)
|-- Choose best result from all workers

```

Fig. 9. Function call hierarchy of the PGA placement Algorithm 3.

delay injection mechanism with customized delays can be easily adopted to evaluate systems in different computing environments, such as cloud-edge continuum and pervasive computing.

## VII. PGA MAPPER FOR MICROSERVICE PLACEMENT

To solve the defined problem in Section III-C, we proposed the following PGA (Parallel Greedy Algorithm) parallel algorithm to address the problem. We propose a Parallel Greedy Algorithm to optimize microservice placement by leveraging parallel computing to minimize communication costs. This method ensures efficient and effective placement of microservices, reducing inter-node communication overhead and improving system performance.

### A. PGA Algorithm Explanation

The algorithm aims to iteratively refine the placement of microservices to minimize the total communication cost, as defined in (2). It leverages parallel processing to handle multiple microservices concurrently, improving the efficiency of the optimization process. The algorithm integrates three core functions: calculating communication costs, optimizing placements for chunks of microservices, and iteratively refining the overall placement until no further improvement is achievable. As shown in Fig. 9, the *ParallelPlace* function coordinates the execution by calling *PlaceWorker* on distributed microservice pairs, which in turn invokes *CalcCost* to evaluate each candidate placement.

The key steps of the proposed method for microservice placement are as follows:

- 1) *Initialization:* The algorithm begins with an initial placement of microservices across server nodes, alongside the input of a traffic stress graph and a cross-node communication delay graph.
- 2) *Resource List Definition:* As outlined in (3), resource constraints such as CPU, memory, and GPU availability

---

**Algorithm 3:** PGA Algorithm for Microservice Placement Under Dynamic Traffics and Cross-Node Delays.

---

```

1: Input: Matrix  $T$  for microservice traffic stress graph,
   Matrix  $D$  for cross-node delay graph, Placement  $P$  for
   service to node mapping list.
2: Output: Microservice-node placement, Lowest cost
3: Define  $res\_list \leftarrow \{cpu, memory, gpu, \dots\}$ 
4:  $res \leftarrow get\_ms\_demands(res\_list, T)$ 
5:  $cap \leftarrow get\_node\_capacities(res\_list)$ 
6: // Cost and Overloads Penalty.
7: function CalcCost $T, P, D, res, cap$ 
8:    $cost \leftarrow 0$ 
9:   for all  $(u, v)$  in  $T$  do
10:     $cost \leftarrow cost + T[u][v] \times D[P[u]][P[v]]$ 
11:   end for
12: Initialize penalty factor  $pf \quad \triangleright pf$  can be adaptive
13:  $loads \leftarrow [0] \times len(cap)$ 
14: for each  $u$  in  $P$  do  $\triangleright$  Calculate server resource load
15:    $loads[P[u]] \leftarrow loads[P[u]] + res[u]$ 
16: end for
17:  $penalty \leftarrow 0$ 
18: for each server  $j$  in  $loads$  do  $\triangleright$  Check for overloads
19:   if  $loads[j] > cap[j]$  then
20:      $penalty \leftarrow penalty + (loads[j] - cap[j]) \times pf$ 
21:   end if
22: end for
23: return  $cost + penalty$ 
24: end Function
25: // Microservice Placement Worker.
26: function PlaceWorker $T, D, P, res, cap, tasks$ 
27:    $current\_cost \leftarrow CalcCost(T, P, D, res, cap)$ 
28:    $nodes\_num \leftarrow |D| \quad \triangleright$  number of server nodes
29:   for each  $(u, v, stress)$  in  $tasks$  do
30:      $new\_P \leftarrow$  other nodes for  $u$  and  $v$ 
31:      $\_cost \leftarrow CalcCost(T, new\_P, D, res, cap)$ 
32:     if  $\_cost < current\_cost$  then
33:       Update  $P$  and  $current\_cost$ 
34:     end if
35:   end for
36:   return  $P, current\_cost$ 
37: end Function
38: // Parallel Placement Computing.
39: function ParallelPlace $T, D, P, res, cap, workers$ 
40:    $pairs \leftarrow Sorted\_MS\_pairs(T) \quad \triangleright$  Algorithm 2
41:    $num\_pairs \leftarrow |pairs| \quad \triangleright$  Get the number of pairs
42:    $num\_workers \leftarrow |workers|$ 
43:    $size \leftarrow \lceil \frac{num\_pairs + num\_workers - 1}{num\_workers} \rceil$ 
44:   /* Distribute tasks to workers */
45:   Initialize  $tasks \leftarrow [] \quad \triangleright$  Initialize chunk tasks
46:   for  $i = 0$  to  $num\_workers - 1$  do
47:      $tasks \leftarrow pairs[i \times size : (i + 1) \times size]$ 
48:   end for
49:    $results \leftarrow PlaceWorker(T, D, P, res, cap, tasks)$ 
50:   Choose the best  $\{P, cost\}$  from  $results$ 
51:   return  $best\_P, best\_cost$ 
52: end Function

```

---

are critical when migrating microservices. A resource list  $res\_list$  is defined at the start, specifying the resources to be considered throughout the placement process.

- 3) *Cost Calculation:* The total communication cost for the initial placement is computed using the  $CalcCost(T, P, D, res, cap)$  function. This function also accounts for server overloads by introducing a penalty factor, which adds to the communication cost if any server's capacity is exceeded.
- 4) *Parallel Processing:* The set of microservices is divided into chunks, and the  $PlaceWorker$  function is applied to each chunk in parallel via  $ParallelPlace$  function.
  - In the  $PlaceWorker$  function, for each microservice pair  $(u, v)$ , the algorithm attempts to reassign  $u$  and  $v$  to every possible pair of server nodes (excluding their current assignments) across the cluster. This is done exhaustively over all node pairs  $(i, j)$ , ensuring that all candidate placements are evaluated. The new placement is only accepted if it results in a strictly lower total cost (including penalty for resource violations) compared to the current placement.
  - In the  $ParallelPlace$  function, the full set of sorted microservice communication pairs is first partitioned into disjoint chunks, where each chunk is assigned to a separate worker (e.g., CPU thread). Each worker executes its own instance of the  $PlaceWorker$  function independently. These workers operate in parallel, concurrently exploring local placement refinements for their assigned microservice pairs. After all workers (CPU threads) complete, their outputs—each representing a locally optimized placement and cost—are gathered, and the best result (i.e., the one with the lowest overall cost) is selected as the final placement. This design improves scalability by enabling simultaneous local search across multiple parts of the system.
- 5) *Iterative Refinement:* The results from all parallel workers (i.e., CPU cores) are gathered, and the overall placement is updated if a better solution is found. This process continues until no further improvement can be made in the placement.
- 6) *Output:* The algorithm concludes by returning the satisfied microservice placement across the server nodes, along with the minimized communication cost.

This approach leverages parallel processing to efficiently handle large-scale microservice deployments, ensuring efficient placement with reduced communication overhead.

### B. Balanced Chunks and Fast Convergence

*Balanced Task Chunks:* To balance the computation tasks at each parallel worker, we adopted the chunk size calculated by  $size = \lceil \frac{num\_pairs + num\_workers - 1}{num\_workers} \rceil$  to ensure a more accurate and even distribution of tasks among workers, especially when the number of tasks is not perfectly divisible by the number of workers. This approach correctly handles the remainder and avoids overestimating the chunk size, ensuring a more balanced task load distribution for the parallel computing process.

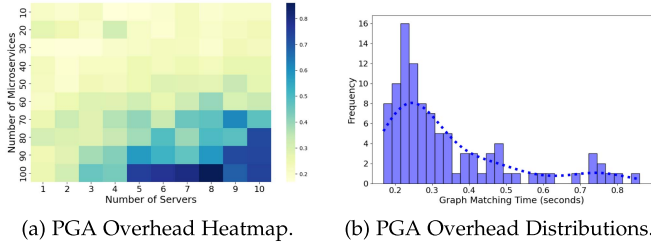


Fig. 10. Processing time overheads for exploring satisfied placements via the PGA algorithm under different numbers of server nodes and microservices.

*Fast Convergence:* Not all microservices contribute equally to the total communication cost. Some pairs of microservices may have significantly higher traffic between them compared to others. As shown in line 40 of the Algorithm 3, high-traffic microservice pairs will be prioritized. By focusing on these high-traffic pairs first, the algorithm addresses the microservice placements that contribute the most to the total cost. Thus, improvements in these placements will have a disproportionately large impact on reducing the overall cost compared to optimizing low-traffic pairs.

### C. Overhead Analysis

Exploring the satisfied placement result is time-consuming. To address this, we designed and implemented the proposed algorithm as a parallel algorithm and quantified the overheads in our cluster. When high-impact microservice pairs are grouped and processed in parallel, it reduces the overhead of synchronization between parallel tasks, thus reducing the need for frequent inter-worker communications. As shown in Fig. 10(a), the running time of the proposed PGA algorithm is able to complete the matching process in less than 1 second. In Fig. 10(b), it can be observed that the majority of the graph matching time is approximately 0.3 seconds, which indicates a promising scheduling decision time with a quick response to dynamic changes.

### D. Adaptive Scheduler

*Adaptive Scheduler* is responsible for rescheduling the deployed microservices containers based on the satisfied placement results from PGA mapper. When there are traffic stresses on the running application or notable communication delays among certain server nodes, the pre-defined QoS targets might be violated in practical computing environments. Thus, designing adaptive scheduling schemes to tackle QoS violations is significant.

The rescheduling process involves microservice instance migration and eviction across different server nodes in the cluster. However, this will lead to the following problems: (1) how to guarantee the performance of microservice instances that are not affected during the rescheduling process?, (2) when service consistency is guaranteed, how to avoid the overloads on certain server loads?

*Asynchronous launching:* At the beginning of microservice migration, it is crucial to ensure the service availability of the affected microservices. As shown in Fig. 11, we initially launch

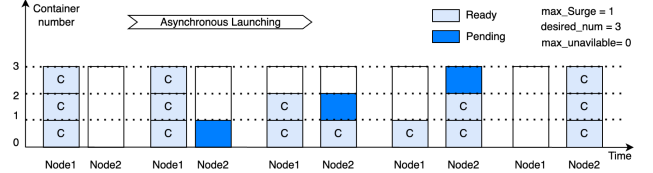


Fig. 11. The process of asynchronous launching for container migration (evicting old while launching new containers) between two nodes to guarantee service high availability.

new microservice containers on the target cluster node, and once these new containers are in ready states, the old containers are evicted from the previous nodes. This approach guarantees zero downtime for specific microservices, whose backend-supported microservice instances require migration.

*Constraints-based scheduling:* The proposed rescheduling scheme computes a service–node placement based on microservice dependencies and the cross-node delay matrix, and also enforces server-node resource constraints (e.g., CPU, memory, GPU), as mathematically summarized in (3). In line 2 of the Algorithm 3, a resource constraints list is defined for consideration during the whole rescheduling process. Additionally, the microservice demands (e.g., requests and limits in deployment yaml file) and server node resource capacities are considered as conditions at Algorithm 3.

## VIII. PERFORMANCE EVALUATION

### A. Testbed Setup

*Cluster Setup:* We validated our design by implementing TraDE using the de facto standard container orchestration platform, Kubernetes [31]. We deployed TraDE on 10 server nodes without any preset anti-colocation rules, such as taints for server nodes and affinity for pods. The Kubernetes cluster configuration includes one control-plane node and nine worker nodes. The control-plane node features 32 CPU cores with x86\_64 AMD EPYC 7763 series processors, 32GiB of RAM, and a network bandwidth of 16 Gbps. Each of the nine worker nodes is equipped with 4 CPU cores from the same AMD series as the control-plane node, 32 GiB of RAM, and a network bandwidth of 16 Gbps. Regarding software versions, the cluster runs Kubernetes v1.27.4, the Container Network Interface (CNI) plugin Calico v3.26.1, the service mesh Istio v1.20.3, and uses CRI-O v1.27.1 as the container runtime. All server nodes operate on Ubuntu 22.04.2 LTS with the Linux kernel 5.15.0.

Besides, each of the cluster nodes is running on a Virtual Machine instance at the dedicated research cloud platform from the University, thus the typical communication delay among each of the nodes is ultra-low, usually ranging from 0.2 to 1 milliseconds (ms), tested by sending ICMP messages between nodes. To make our evaluations more realistic and emulate the changing cluster networking environment, the cross-node delay matrix can be automatically updated every  $t$  (i.e.,  $t = 5$ ) minutes. In Table I, we demonstrated the customized cross-node communication delays to the worker node destinations and also avoided injecting delays to other destinations like the control-plane node and external sites.

*Benchmark Application:* We adopted Social Network from DeathStarBench [23] as the microservice application to evaluate our proposed scheduling framework. Social Network benchmark emulates a simplified social media platform similar to popular social networking services. It is structured to replicate the intricate interactions and communication patterns in such type of applications. The benchmark comprises 27 different microservices that collectively offer functionalities such as composing a post, writing a user timeline, and writing a home timeline.

## B. Workload Generator

We adopted wrk2 [32] as the workload generator. As a modern HTTP benchmarking tool, wrk2 is capable of generating different types and proportions of workload requests for performance testing and measuring how well the cloud applications can handle varied traffic. Its ability to maintain a constant request rate makes it particularly useful for understanding the end-to-end performance of a server under controlled workload conditions.

## C. QoS Targets and Compared Methods

1) *QoS Target Determination (Trigger-Oriented):* In our experiments, we use a fixed latency QoS target as the control-plane trigger for Trade: every  $\tau=30s$  the scheduler queries Prometheus over a sliding window  $W$  (default  $W=1min$ ) for Istio metrics in the target namespace with `response_code = 200`—specifically `istio_request_duration_milliseconds_sum` and `istio_request_duration_milliseconds_count` (via `rate` and `custom_query_range` functions)—and computes the windowed average response time  $\bar{L} = \frac{\sum \text{sum\_values}}{\sum \text{count\_values}}$  (ms); if  $\bar{L} > T$  with  $T=300$  ms (we also test  $T \in \{250, 300, 350\}$  ms in Fig. 15), Trade sets `Trigger = True` and executes the rescheduling pipeline implemented in Trade (traffic-graph construction, current placement extraction, and PGA-based remapping/migration); if the window has no data or the traffic count is zero, no trigger is raised.

2) *Compared Methods:* To evaluate our proposed Trade framework, we compared it with the default Kubernetes scheduling policy and the recent traffic-aware NetMARKS [11] which also targets network-aware scheduling for microservice applications.

*K8s default policy:* In Kubernetes, the default scheduling and Quality of Service (QoS) policies are designed to evenly distribute workloads across the cluster without further rescheduling policies even when microservice performance is violated. QoS policies in the k8s cluster classify pods into three classes: *Guaranteed*, *Burstable*, and *BestEffort*. *Guaranteed* provides the highest priority ensuring pods always get the requested resources, *Burstable* offers a flexible middle ground where pods have guaranteed minimum resources but can consume more if available, while *BestEffort* has the lowest priority, where pods

have no resource guarantees and can be preempted first during resource contention. In the evaluation experiments, we will use the default scheduling policy with *Burstable* QoS class.

*NetMARKS:* The recent work NetMARKS [11] introduces a microservice pod scheduling scheme that leverages dynamic network metrics collected from the Istio Service Mesh. The main idea of NetMARKS is the proposed node scoring algorithm, which calculates node scores for a target pod by iteratively analyzing all pods running on each node and identifying those with traffic connections to the target pod. Each node's score is calculated based on the sum of traffic flows between the target pod and the selected pods on that node. The node with the highest score is then chosen to host the target pod.

With the implemented modules of *Traffic Analyzer* in Section V, *Dynamics Manager* in Section VI and *PGA Mapper* in Section VII, we evaluated the proposed Trade with benchmark microservice application and compared the end-to-end performance with default k8s QoS policy [31] and NetMARKS [11]. In the evaluation experiments, we implemented the K8s QoS policy with *Burstable* and the pod scheduling policy without any pre-set rules like affinity and taints. For NetMARKS [11], we implemented the node scoring algorithm proposed by NetMARKS and adopted it for rescheduling the predefined target microservice pods experiencing QoS violations.

To ensure fair evaluations, three separate namespaces are created for three identical social network applications from [23], each managed by a different method (i.e., K8s default, NetMARKS, and Trade). Additionally, sustained workloads are sent concurrently to each of the three identical social network applications. This setup isolates the workloads for each method while maintaining the same cross-node delay settings for the three applications at the cluster level.

We generated multiple workloads with different requests and varying QPS to evaluate the end-to-end performance of our proposed Trade and existing methods.

*Response Time and Durations.* For the social network benchmark application, we used the wrk2 tool to generate three request types: `compose-post`, `read-user-timeline`, and `read-home-timeline`. Each type shows distinct call graphs and traffic patterns across the application's dependency graph. Fig. 12 compares the average response times under various workloads, including changes in QPS and user requests. For these request types, Trade consistently outperforms existing methods, meeting QoS targets across scenarios. Compared to NetMARKS [11], Trade achieves up to 12.3% lower response times for `compose-post`, 48.3% for `read-user-timeline`, and 25.8% for `read-home-timeline` requests. Notably, Trade adapts effectively under varied workloads, where K8s-burstable and NetMARKS do not always meet QoS.

Fig. 12(d) illustrates the CDF of execution times for all triggered microservices under a mixed workload (6:2:2 ratio of `compose-post`, `read-user-timeline`, and `read-home-timeline` requests). Here, Trade demonstrates reduced microservice execution time, leading to lower response times and higher throughput.

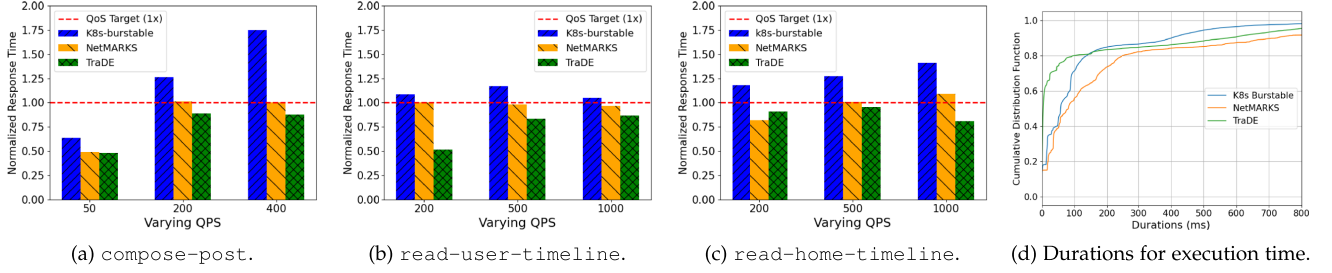


Fig. 12. In (a)–(c), different response time under varying QPS and request types. In (d), the cumulative distribution function (CDF) figures of all triggered microservices execution time by mixed workload requests (with 6:2:2 ratio of *compose-post*, *read-user-timeline*, and *read-home-timeline* requests), showing TraDE has an overall reduced execution time, thereby leading to better end-to-end performance.

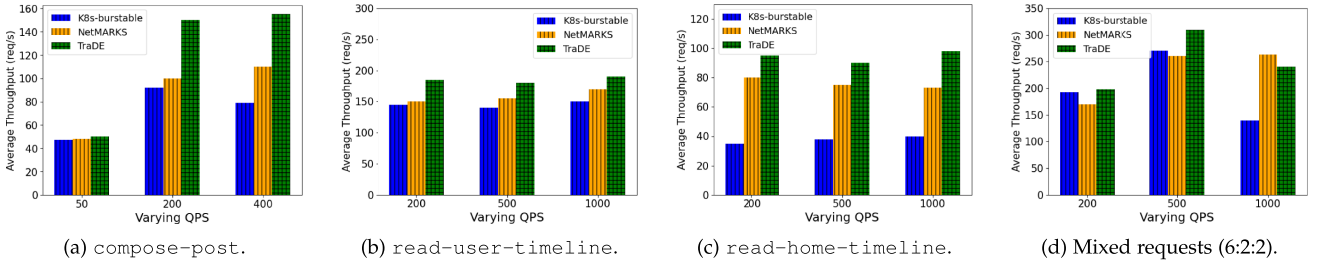


Fig. 13. In (a)–(c), the evaluation comparisons of average throughput are shown under varying QPS and different request call-graphs. In (d), the throughput under mixed workload requests is displayed with the proportion of 6:2:2 for *compose-post*, *read-user-timeline*, and *read-home-timeline*, respectively.

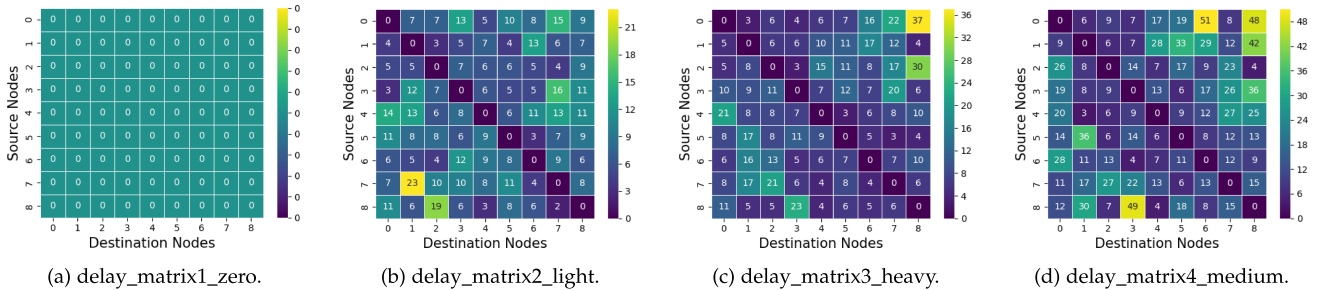


Fig. 14. In (a)–(d), dynamic delays are injected to the nine worker nodes.

## D. End-to-End Performance

**Throughput and Goodput.** Throughput and goodput are key metrics for assessing end-to-end performance in dynamic environments. Throughput represents total data transmitted, including overhead, while goodput measures only the useful data successfully received at the application layer.

In terms of *throughput*, we conducted experiments to measure the average throughput (requests/second) across different QPS and mixed workloads. As shown in Fig. 13, TraDE achieves higher throughput than NetMARKS [11]—up to 1.5x for *compose-post*, 1.2x for *read-user-timeline*, 1.4x for *read-home-timeline*, and 1.2x for mixed requests, indicating superior throughput in various scenarios.

For *goodput*, we analyzed response types using Istio Service Mesh across separate deployments (K8s-burstable, NetMARKS, and TraDE) for isolation. Fig. 16 shows TraDE

achieves a 95.36% success rate, outperforming NetMARKS (71.99%) and K8s-burstable (65.43%). These results show that TraDE surpasses existing methods in both throughput and goodput across dynamic workloads.

## E. Adaptive Performance Under Changing Delays

To assess the adaptive capability of the proposed TraDE framework, we evaluated its performance under fluctuating cross-node communication delays. Specifically, in Fig. 14, four different cross-node delays were injected to the cluster nodes every five minutes. As shown in Fig. 15, TraDE effectively responds to these changing delays by adaptively redeploying microservice instances to meet QoS targets once detecting QoS violations. Comparing with the existing two methods, it is clear to observe that TraDE can consistently maintain response times within the QoS targets (i.e., < 250 ms, < 300 ms, < 350 ms)

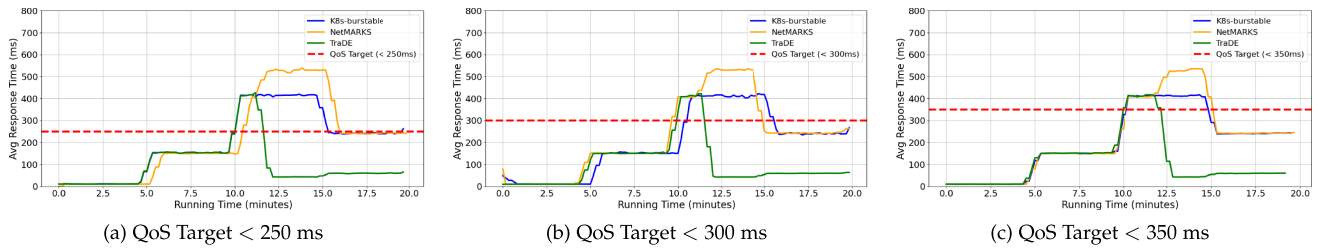


Fig. 15. Under three different QoS targets and varying cross-node delays, the adaptive response of k8s-burstable, NetMARKS, and TraDE. In 20 minutes of the test time, 0 ~ 5 mins, there are no injected cross-node delays; 5~10 mins, light cross-node delays are injected; 10~15 mins, heavy cross-node delays are injected; 15~20 mins, medium cross-node delays are injected.

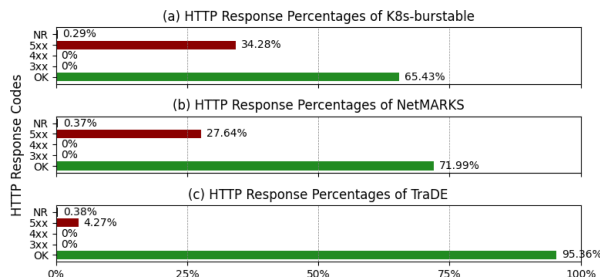


Fig. 16. In each service mesh, HTTP response percentage distributions show the overall microservice application performance under different deployment methods. The higher the percentages of 'OK' responses, the higher the goodput ratio.

throughout the remaining runtime, while the other two methods (K8s-burstable and NetMARKS) failed to meet these QoS targets.

## IX. CONCLUSIONS AND FUTURE WORK

In this work, we designed a traffic and network-aware framework, TraDE, to address the challenges of QoS violations in containerized microservices running in dynamic computing environments. Our framework primarily consists of three components: a traffic stress analyzer, a network dynamics manager, and an efficient service node mapper. We evaluated our proposed TraDE against existing solutions and demonstrated that our framework effectively meets the QoS targets under various dynamic conditions, outperforming the existing method NetMARKS by reducing response time by up to 48.3%, improving throughput by up to 1.4x and showing robust adaptiveness under sustained workloads.

As part of future work, we plan to enhance our framework to support resource auto-scaling for microservice-based GPT (Generative Pre-trained Transformer) applications at the cloud-edge continuum, which demands stricter end-to-end performance, and they are sensitive to dynamics like cross-node delays.

*Software:* TraDE software code as open source is available at <https://github.com/Cloudslab/TraDE>.

## ACKNOWLEDGMENT

The authors thank anonymous reviewers for their comments on improving the article.

## REFERENCES

- [1] S. Pallewatta, V. Kostakos, and R. Buyya, "Placement of microservices-based IoT applications in fog computing: A taxonomy and future directions," *ACM Comput. Surv.*, vol. 55, Jul. 2023, Art. no. 321.
- [2] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," *SIGAPP Appl. Comput. Rev.*, vol. 17, pp. 29–45, Jan. 2018.
- [3] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the, "micro" back in microservice," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 645–650.
- [4] S. Luo et al., "Optimizing resource management for shared microservices: A scalable system design," *ACM Trans. Comput. Syst.*, vol. 42, Feb. 2024, Art. no. 1.
- [5] L. Gu, D. Zeng, J. Hu, H. Jin, S. Guo, and A. Y. Zomaya, "Exploring layered container structure for cost efficient microservice deployment," in *Proc. IEEE Conf. Comput. Commun.*, 2021, pp. 1–9.
- [6] S. Pallewatta, V. Kostakos, and R. Buyya, "Reliability-aware proactive placement of microservices-based IoT applications in fog computing environments," *IEEE Trans. Mobile Comput.*, vol. 23, no. 12, pp. 11326–11341, Dec. 2024.
- [7] M. Xu et al., "CoScal: Multifaceted scaling of microservices with reinforcement learning," *IEEE Trans. Netw. Service Manag.*, vol. 19, no. 4, pp. 3995–4009, Dec. 2022.
- [8] S. Luo et al., "Erms: Efficient resource management for shared microservices with SLA guarantees," in *Proc. 28th ACM Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2022, pp. 62–77.
- [9] J. Liu, S. Zhang, and Q. Wang, "ConAdapter: Reinforcement learning-based fast concurrency adaptation for microservices in cloud," in *Proc. 2023 ACM Symp. Cloud Comput.*, 2023, pp. 427–442.
- [10] X. Zhu et al., "On optimizing traffic scheduling for multi-replica containerized microservices," in *Proc. 52nd Int. Conf. Parallel Process.*, 2023, pp. 358–368.
- [11] L. Wojciechowski et al., "NetMARKS: Network metrics-aware kubernetes scheduler powered by service mesh," in *Proc. IEEE Conf. Comput. Commun.*, 2021, pp. 1–9.
- [12] Istio: An open source service mesh, 2024. Accessed: May 25, 2024. [Online]. Available: <https://istio.io/>
- [13] C. Song et al., "ChainsFormer: A chain latency-aware resource provisioning approach for microservices cluster," in *Proc. 21st Int. Conf. Serv.-Oriented Comput.*, 2023, pp. 197–211.
- [14] L. Cao and P. Sharma, "Co-locating containerized workload using service mesh telemetry," in *Proc. 17th Int. Conf. Emerg. Netw. Experiments Technol.*, 2021, pp. 168–174.
- [15] A. Marchese and O. Tomarchio, "Network-aware container placement in cloud-edge kubernetes clusters," in *Proc. 22nd IEEE Int. Symp. Cluster Cloud Internet Comput.*, 2022, pp. 859–865.
- [16] A. Marchese and O. Tomarchio, "Extending the kubernetes platform with network-aware scheduling capabilities," in *Proc. 20th Int. Conf. Serv.-Oriented Comput.*, 2022, pp. 465–480.
- [17] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 805–825.
- [18] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "GrandSLAM: Guaranteeing SLAs for jobs in microservices execution frameworks," in *Proc. 14th EuroSys Conf.*, 2019, Art. no. 34.

- [19] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical and scalable ML-driven performance debugging in microservices," in *Proc. 26th ACM Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2021, pp. 135–151.
- [20] H. Tian, Y. Zheng, and W. Wang, "Characterizing and synthesizing task dependencies of data-parallel jobs in Alibaba cloud," in *Proc. ACM Symp. Cloud Comput.*, 2019, pp. 139–151.
- [21] S. Luo et al., "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 412–426.
- [22] A. Mirhosseini, S. Elnikety, and T. F. Wenisch, "Parslo: A gradient descent-based approach for near-optimal partial SLO allotment in microservices," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 442–457.
- [23] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proc. 24th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2019, pp. 3–18.
- [24] Envoy: An Open Source Edge and Service Proxy, Designed for Cloud Native Apps, 2024. Accessed: May 25, 2024. [Online]. Available: <https://www.envoyproxy.io/>
- [25] Istio: Performance and Scalability, 2024. Accessed: May 25, 2024. [Online]. Available: <https://istio.io/latest/docs/ops/deployment/performance-and-scalability/>
- [26] CNCF Community Infrastructure Lab (CIL): An on-demand infrastructure resource for CNCF developers, 2024. Accessed: May 25, 2024. [Online]. Available: <https://istio.io/latest/docs/ops/deployment/performance-and-scalability/>
- [27] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Trans. Mobile Comput.*, vol. 20, no. 3, pp. 939–951, Mar. 2021.
- [28] M. Xu, Q. Zhou, H. Wu, W. Lin, K. Ye, and C. Xu, "PDMA: Probabilistic service migration approach for delay-aware and mobility-aware mobile edge computing," *Softw.: Pract. Experience*, vol. 52, no. 2, pp. 394–414, 2022.
- [29] K. Fu et al., "QoS-aware and resource efficient microservice deployment in cloud-edge continuum," in *Proc. 2021 IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 932–941.
- [30] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 8, pp. 1825–1840, Aug. 2022.
- [31] K. A. open-source container orchestration system, 2024. Accessed: May 25, 2024. [Online]. Available: <https://kubernetes.io/>
- [32] wrk2: An HTTP benchmarking tool based on wrk, 2024. Accessed: May 25, 2024. [Online]. Available: <https://github.com/giltene/wrk2>



**Muhammed Tawfiqul Islam** received the PhD degree from the School of Computing and Information Systems (CIS), University of Melbourne, Australia, in 2021. He is a lecturer with the School of Computing and Information Systems (CIS), University of Melbourne, Australia. His research interests include resource management, cloud computing, and Big Data.



**Maria Rodriguez Read** is a lecturer with the School of Computing and Information Systems, University of Melbourne, Australia. Her research focuses on distributed and parallel systems, particularly optimizing support for containerized and cloud-native applications to improve scalability, robustness, and cost-efficiency in cloud deployments.



**Rajkumar Buyya** (Fellow, IEEE) is a Redmond Barry distinguished professor and director with the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne, Australia. He has authored more than 800 publications and seven textbooks. He is a fellow of the Academia Europea. He is one of the highly cited authors in computer science and software engineering worldwide (h-index=175, g-index=384, 164,100+ citations).



**Ming Chen** is currently working toward the PhD degree with the School of Computing and Information Systems (CIS), University of Melbourne, Australia. His main research interests focus on resource scheduling, networking, and cloud computing.