# Layer-aware Collaborative Microservice Deployment toward Maximal Edge Throughput

Lin Gu*, Zirui Chen*, Honghao Xu*, Deze Zeng†, Bo Li‡, Hai Jin*

*National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
†School of Computer Science, China University of Geosciences, Wuhan, Hubei, China
‡Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong

***Abstract*—Lightweight container-based microservice has been widely advocated to promote the elasticity of edge cloud. The inherent layered structure of containers offers a compelling way to cope with the resource scarcity of edge servers through layer sharing, which can significantly increase storage utilization and improve the edge throughput. Recent studies show that it is possible to share layers not only within the same server but also between servers, which microservice deployment can take full advantage of. In this paper, we investigate the problem of how to collaboratively deploy microservices by incorporating both intra-server and inter-server layer sharing to maximize the edge throughput. We formulate this problem into an integer linear programming form and prove it as NP-hard. We propose a randomized rounding based heuristic algorithm, and conduct formal analysis on the guaranteed approximation ratio. Through extensive experiments, we verify the efficiency of our proposed algorithm, and the results demonstrate that it can deploy $6\times$ and $12\times$ more microservice instances and improve the edge throughput by $27.74\%$ and $38.46\%$ in comparison with state-of-the-art strategies.**

## I. Introduction

Edge cloud has emerged as a promising platform complementary to cloud systems by provisioning computation resources at the network edge. Owing to the advantage on end user proximity, edge cloud is ideal for many delay-sensitive applications such as self-driving, healthcare, augmented reality, smart manufacturing and etc. Meanwhile, in comparison with central cloud empowered by large datacenters, edge cloud resources are relatively scarce. Therefore, how to mostly effectively utilize the edge resources has become one of the main research focuses recently. On the other hand, *container-based microservices* have been advocated as an edge service provision paradigm, thanks to its lightweight advantages, which offer flexibility and elasticity over conventional virtual machines (VMs) [1]–[3].

While containers are relatively lightweight, their footprints are still far from being negligible. For instance, the size of top 134 most downloaded images in Docker Hub varies from 109MB to 2045MB. This puts a heavy burden on the storage and severely limits the service capacity of an edge server. At the same time, it can also lead to massive image downloading traffic and slow microservice startup [4]. A recent study [5] shows that the average single microservice startup time over a 100Mbps network is as high as $20.7$ seconds and $72.4\%$ of the startup time is spent on image downloading. To overcome this problem, existing studies have tried to slim container image size [6], enable partial image downloading [7], pre-import required libraries [8] or redesign container images [9].

By closely examining the container architecture, it can be observed that container images like Docker have an intrinsic *layered structure*. Such a feature can be properly exploited to reduce the image storage requirement, downloading traffic, as well as microservice startup time without refactoring container images. Container-based microservice packages everything required in its container image and stores them in multiple layers, including runtime tools, system tools and system dependencies. Thus, deploying a microservice on a server is essentially equivalent to load all the required layers into the server [10]. Fortunately, it has been pointed out that some common layers (especially those read-only layers) can be shared between different microservices on the same server (i.e., intra-server sharing) [11] or across different servers via distributed file system (i.e., inter-server sharing) [12]. A recent study [7] shows that $57$ representative microservice images have $19$ common base layers. Taking microservices of Cassandra, JAVA, Python and gcc as examples, their images all require one non-latest Linux distribution layer of Debian. By sharing the common layers, the storage requirement of a microservice can be significantly reduced. This not only reduces the image download traffic and the startup time, but also potentially expands the service capacity of a resource-constrained edge server.

In an edge-cloud system, given the resource scarcity of edge servers, it is natural to consider layer sharing during microservice deployment. Existing studies [9], [13] have largely treated containers as lightweight VMs. Gu et al. [11] recently propose a layer sharing microservice deployment solution to improve the *edge throughput* (defined as the number of requests satisfied by an edge cloud). However, this study only focuses on intra-server layer sharing. When inter-server layer sharing is considered, a layer not stored locally can be loaded from another server as long as the startup time is acceptable [12]. This essentially expands the layer sharing scope and inflates the resource utilization to achieve higher

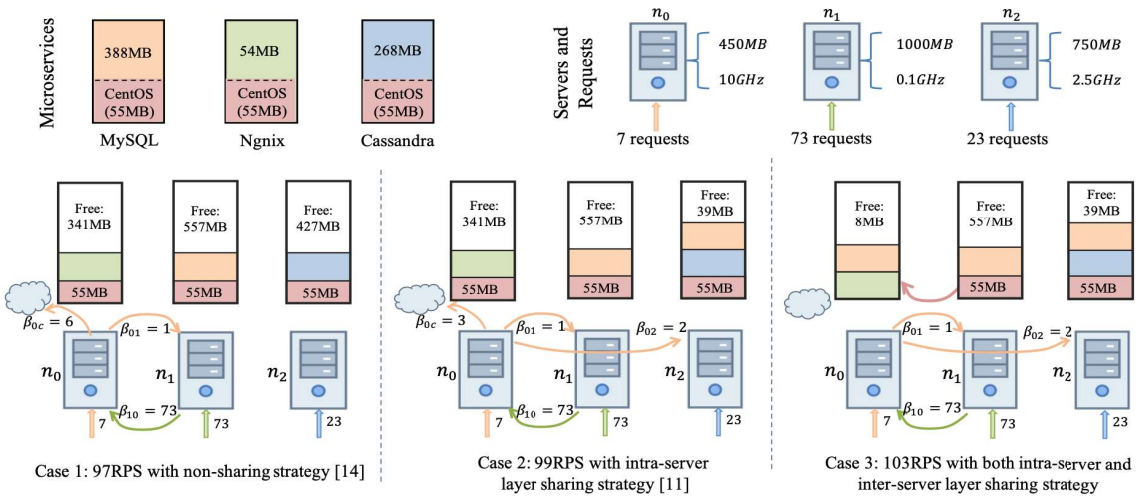



Fig. 1. An example of three microservice deployment and layer storage strategies

edge throughput. So far there has been no systematic study that quantitatively characterizes the throughput improvement, and this work attempts to fill the gap.

It is worth noticing that by enabling inter-server layer sharing, the microservice deployment and layer storage are no longer tightly coupled as a required layer of a microservice can be stored independently apart from the microservice. Consequently, the microservice deployment inevitably becomes more complicated in that edge servers can now collaborate with each other to balance the storage and computation resource usages, in order to handle more requests on edge cloud. This work focuses on the layer-aware collaborative microservice deployment with the objective of maximizing edge throughput, which is proved to be NP-hard. To tackle the computation complexity, a heuristic algorithm is proposed based on randomized rounding. Our major contributions are summarized below.

- To our best knowledge, this is the first work to consider both intra-server and inter-server layer sharing and investigate the joint microservice deployment and layer storage (JMDLS) problem. We formulate the JMDLS problem toward maximal edge throughput into an integer linear programming (ILP) formulation.
- The JMDLS problem is proved to be NP-hard through reducing from set-union knapsack problem. By incorporating the ILP formulation, we further design a randomized rounding based algorithm with a guarantee approximation ratio of $1 - \sqrt{\frac{2M}{\mathcal{OL}}}$.
- We conduct extensive trace driven experiments to analyze the performance of our proposed algorithm by comparing with state-of-the-art microservice deployment strategies. The results show that our algorithm can increase the edge throughput by $27.74\%$ and $38.46\%$ over LA-MPRS [11] and JSPRR [14], respectively.

The remainder of this paper is organized as follows. Section II presents the motivations and the JMDLS problem is formulated in Section III. Then, we propose our randomized rounding based algorithm and analysis in Section IV. The trace driven performance evaluation results are reported in Section V. Section VI discusses some related work. Finally, Section VII concludes our work.

## II. Background and Motivation

Let us first consider an edge cloud as shown in Fig. 1, where there are three edge servers and three microservices to be deployed, i.e., MySQL, Nginx and Cassandra. According to the Docker Hub library, the latest versions of MySQL, Nginx and Cassandra are 443MB, 109MB and 323MB, respectively. All three images contain two common base layers of 55MB as CentOS, as shown in Fig. I. These three edge servers are with available storage capacity of 450MB, 1000MB and 750MB, and computation capacity of 10GHz (e.g., 2.0GHz×5cores), 0.1GHz and 2.5GHz, respectively. Following a real trace from IBM Docker Registry Trace Player, the request rates of MySQL, Nginx and Cassandra are set as 7, 73 and 23 requests per second (RPS), and computation resource requirements are 0.1GHz, 0.1GHz and 0.1GHz, respectively. In this toy example, we only concentrate on the storage and computation resource constraints and let the communication capacity be infinite. Let us check how to deploy the three microservices to maximize the edge throughput.

In case 1, following the strategy in [14] that treats a microservice image as a whole, deploying MySQL, Nginx and Cassandra on servers $n_2$, $n_1$ and $n_3$, respectively, will achieve the highest edge throughput as 97. When intra-server sharing aware strategy [11] is applied, as shown in case 2, both MySQL and Cassandra can be deployed on server $n_3$ to share the common base layers of Debian. The throughput is increased to 99 thanks to the 2 more requests for MySQL handled by $n_3$. We further take inter-server sharing into consideration, as shown in case 3. By such means, MySQL

could be deployed on all three servers $n_1$, $n_2$ and $n_3$ by layer sharing to fully exploit the computation resources. In this case, the edge throughput is further increased to 103.

It can be seen from case 3 that inter-server layer sharing decouples microservice deployment and layer storage, and enables server collaboration to balance their storage and computation resource usage. This raises a new question on how to make microservice deployment and layer storage decisions to better utilize edge resources for maximal edge throughput. Motivated by such issue, we investigate the JMDLS problem toward edge throughput maximization and propose a randomized rounding based algorithm in this paper.

## III. System Model and Problem Formulation

### A. System Model and Problem Statement

In this paper, we consider an edge cloud consisting of a set $\mathbf{N}$ of resource capacitated servers. With respect to the microservice provision process, we mainly concern on storage capacity, computation capacity, uplink capacity and downlink capacity, which are denoted as $\Omega_n^S$, $\Omega_n^C$, $\Omega_n^U$ and $\Omega_n^D, \forall n \in \mathbf{N}$, respectively. To enable inter-server layer sharing as proposed in [12], we assume that distributed file system is implemented and the data access rate between server $n^\dagger$ and $n$ is $R_{n^\dagger n}$.

There is a set $\mathbf{M} = \{1, 2, \cdots, M\}$ of microservices to be deployed. We assume that each microservice corresponds to one container image. A container image is made up of several layers, including common sharable layers and unique non-sharable layers (usually the functionality codes and tools of microservices). The layers of all the images are denoted as set $\mathbf{L}$. A layer $l \in \mathbf{L}$ is with size (i.e., storage requirement) $s_l$. We introduce $\theta_m^l$ to indicate whether the container image of microservice $m \in \mathbf{M}$ requires $l \in \mathbf{L}$ ($\theta_m^l = 1$) or not ($\theta_m^l = 0$). Starting up a microservice $m$ on a server $n$ is equivalent to loading all the required layers of $m$ to server $n$. To guarantee the QoS, we assume that microservice $m$ must be started up within time threshold $\mathcal{D}_m$.

Once a microservice is successfully started up, it can handle the corresponding requests, which may first arrive at any edge server due to geographic distribution of end users. Let $\mathbf{U}$ be the set of users, and the request arrival rate of user $u$ for microservice $m$ at server $n$ is denoted as $r_{nm}^u$. The requests of a user can be handled integrally on any server with the intended microservice, provided that there are sufficient resources. Different microservices have different computation resource requirements. Let $\omega_m^C$ be the unit computation resource for handling a request of microservice $m \in \mathbf{M}$. A request that cannot be handled locally has to be transferred to another server with the intended microservice with transfer overhead of $\omega_m^S$.

Considering the resource capacity constraints of edge servers, the requests that cannot be satisfied by edge cloud have to resort to central cloud. Therefore, being aware of layer sharing, we are particularly interested in how to jointly make microservice deployment and layer storage decisions to maximize the edge throughput, subject to the edge resource capacity constraints.

### B. Problem Formulation

Based on the above system model, we next provide a formal description of the JMDLS problem.

*1) Microservice Deployment and Request Distribution:* A microservice can be deployed on any server. We introduce binary variables $\alpha_n^m$ to represent whether microservice $m \in \mathbf{M}$ is deployed on edge server $n \in \mathbf{N}$ ($\alpha_n^m = 1$) or not ($\alpha_n^m = 0$). The requests of user $u$ for microservice $m$ arriving at server $n'$ may be handled by any server deployed with microservice $m$ and with enough computation resources. We introduce binary variables $\beta_{n'n}^{mu}$ to denote whether the requests are distributed to server $n$ ($\beta_{n'n}^{mu} = 1$) or not ($\beta_{n'n}^{mu} = 0$). Obviously, a user's requests can only be distributed to an edge server with the intended microservice. Therefore, we have

$$0 \leq \beta_{n'n}^{mu} \leq \alpha_n^m, \quad \forall n, n' \in \mathbf{N}, m \in \mathbf{M}, \forall u \in \mathbf{U}. \tag{1}$$

Due to resource capacity limitations, the requests that cannot be handled by the edge cloud have to resort to the central cloud. We assume that the central cloud has sufficient resources to deploy all the microservices and handle any request. Anyhow, all the requests must be completely handled to guarantee the QoS. By further introducing $\beta_{n'c}^{mu}$ to denote whether the requests of user $u$ for microservice $m$ are dispatched from server $n'$ to the central cloud or not, we have

$$\sum_{n \in \mathbf{N}} \beta_{n'n}^{mu} + \beta_{n'c}^{mu} = 1, \quad \forall u \in \mathbf{U}, \forall m \in \mathbf{M}, n' \in \mathbf{N}. \tag{2}$$

The total computation resource requirement of all requests distributed to server $n$ cannot exceed its computation capacity $\Omega_n^C$. Therefore, we have

$$\sum_{m \in \mathbf{M}} \sum_{n' \in \mathbf{N}} \sum_{u \in \mathbf{U}} \beta_{n'n}^{mu} r_{n'm}^u \omega_m^C \leq \Omega_n^C, \quad \forall n \in \mathbf{N}. \tag{3}$$

Besides, the transfer of requests between servers is limited by the uplink capacity $\Omega_{n'}^U$ of the sending server $n'$ and the downlink capacity $\Omega_n^D$ of the receiving server $n$. That is,

$$\sum_{m \in \mathbf{M}} \sum_{n \in \mathbf{N}/\{n'\}} \sum_{u \in \mathbf{U}} \beta_{n'n}^{mu} r_{n'm}^u \omega_m^S \leq \Omega_{n'}^U, \quad \forall n' \in \mathbf{N}, \tag{4}$$

and

$$\sum_{m \in \mathbf{M}} \sum_{n' \in \mathbf{N}/\{n\}} \sum_{u \in \mathbf{U}} \beta_{n'n}^{mu} r_{n'm}^u \omega_m^S \leq \Omega_n^D, \quad \forall n \in \mathbf{N}. \tag{5}$$

Note that no transfer overhead will be incurred if a request is handled locally. Therefore, we specially exclude the case $n = n'$ in (4) and (5).

*2) Layer Storage and Sharing:* By enabling inter-server layer sharing, the layers of a microservice can be stored on any server $n$ and shared to other servers. We introduce binary variables $\gamma_n^l$ to denote whether layer $l \in \mathbf{L}$ is stored on server $n \in \mathbf{N}$ ($\gamma_n^l = 1$) or not ($\gamma_n^l = 0$). The layers that can be stored on a server are limited by server storage capacity. Therefore, we have

$$\sum_{l \in \mathbf{L}} \gamma_n^l s_l \leq \Omega_n^S, \quad \forall n \in \mathbf{N}. \tag{6}$$

If a required layer $l$ is not stored locally, it needs to be loaded from another server. Let binary variables $\varepsilon^l_{n^\dagger n}$ represent whether a layer $l$ is loaded from server $n^\dagger$ to $n$ ($\varepsilon^l_{n^\dagger n}=1$) or not ($\varepsilon^l_{n^\dagger n}=0$). Specially, $\varepsilon^l_{nn}=1$ indicates that layer $l$ is loaded from local server $n$. A layer $l$ can be loaded from server $n^\dagger$ only if $n^\dagger$ stores layer $l$, i.e.,

$$\varepsilon^l_{n^\dagger n} \le \gamma^l_{n^\dagger}, \quad \forall n, n^\dagger \in \mathbf{N}, l \in \mathbf{L}. \tag{7}$$

A microservice can be successfully deployed at an edge server if and only if all required layers are completely loaded. That is,

$$\alpha^m_n \theta^l_m \le \sum_{n^\dagger \in \mathbf{N}} \varepsilon^l_{n^\dagger n}, \quad \forall n \in \mathbf{N}, m \in \mathbf{M}, l \in \mathbf{L}. \tag{8}$$

It indicates that a microservice $m$ cannot be provisioned on server $n$ if either of its required layer $l$ is not stored in the edge cloud, i.e., $\alpha^m_n \equiv 0$ if $\exists l \in \mathbf{L}, \theta^l_m = 1 \ \& \ \sum_{n^\dagger \in \mathbf{N}} \varepsilon^l_{n^\dagger n} = 0$.

Note that there might coexist multiple replicas stored on different servers for the same layer. A server should select at most one server (including itself) to load a layer. Hence, we have

$$\sum_{n^\dagger \in \mathbf{N}} \varepsilon^l_{n^\dagger n} \le 1, \quad \forall n \in \mathbf{N}, l \in \mathbf{L}. \tag{9}$$

To ensure the QoS, the total layer loading time for a microservice must satisfy

$$\sum_{l \in \mathbf{L}} \sum_{n^\dagger \in \mathbf{N}} \frac{\alpha^m_n \theta^l_m \varepsilon^l_{n^\dagger n} s_l}{R_{n^\dagger n}} \le \mathcal{D}_m, \quad \forall n \in \mathbf{N}, \forall m \in \mathbf{M}. \tag{10}$$

Obviously, the threshold $\mathcal{D}_m$ restricts the layer sharing scope and therefore a layer cannot be shared among arbitrary servers. One may notice that (10) contains non-linear terms. Fortunately, it can be equivalently transformed to linear form as

$$\sum_{l \in \mathbf{L}} \sum_{n^\dagger \in \mathbf{N}} \frac{\theta^l_m \varepsilon^l_{n^\dagger n} s_l}{R_{n^\dagger n}} \le \alpha^m_n \mathcal{D}_m + (1-\alpha^m_n) A, \forall n \in \mathbf{N}, \forall m \in \mathbf{M}, \tag{11}$$

where $A$ is an arbitrarily large number. The equivalence between (10) and (11) can be easily verified by enumerating all possible combinations of $\varepsilon^l_{n^\dagger n}$ and $\alpha^m_n$. We omit here for brevity.

*3) JMDLS Problem Formulation:* By summing up the above, we can formulate our JMDLS problem, with the goal of edge throughput maximization, into an ILP as

**JMDLS**:

$$\max_{\alpha,\beta}: \sum_{m \in \mathbf{M}} \sum_{n \in \mathbf{N}} \sum_{n' \in \mathbf{N}} \sum_{u \in \mathbf{U}} \beta^{mu}_{n'n} r^u_{n'm}$$
$$\text{s.t.}: \quad (1)-(9), (11).$$

### C. Complexity Analysis

In this section, we prove the NP-hardness of our JMDLS through reducing from set-union knapsack problem (SKP) [15]. As a generalized 0-1 knapsack problem, SKP consists of a set $\mathbf{I}$ of $I$ elements with weight $w_i$ for $i \in \mathbf{I}$, a set $\mathbf{J}$ of $J$ items where each consisting of a subset of elements, i.e., $j \subseteq \mathbf{I}$, and a knapsack with capacity $b$. Note that only an item is associated with a nonnegative profit $v_j, j \in \mathbf{J}$. The elements required by all items equals to the element set, i.e., $\bigcup_{j \in \mathbf{J}} j = \mathbf{I}$. The goal of SKP is to find a subset of items $\mathbf{K} \subseteq \mathbf{J}$ with the maximum profit, subject to the knapsack capacity constraint, i.e., $\sum_{i \in \bigcup_{k \in \mathbf{K}} k} w_i \le b$.

In the JMDLS problem, we can view a layer $l \in \mathbf{L}$ as an element with weight as its storage resource requirement $s_l$ and a microservice $m \in \mathbf{M}$ requiring a subset of layers as an item. Let us consider a special case where all the servers are with sufficient computation and communication resources. In this case, one instance of each microservice is enough to handle all requests and the profit of microservice $m$ can be calculated as $v_m = \sum_{n \in \mathbf{N}, u \in \mathbf{U}} r^u_{nm}$. Hence, the JMDLS problem can be solved by deploying a subset of microservices $\mathbf{M}^* \subseteq \mathbf{M}$ into server knapsacks, aiming to achieve maximum throughput $\sum_{m^* \in \mathbf{M}^*} v_m$ without violating the knapsack capacity $\Omega^S_n$. This is a typical SKP, which has been proved as NP-hard [16]. Hence, the JMDLS problem, as a general case, is NP-hard.

**Algorithm 1** Randomized Rounding based Algorithm

1: Solve the linear relaxation of JMDLS to obtain real number solutions $\widehat{\alpha}^m_n$, $\widehat{\beta}^{mu}_{n'n}$, $\widehat{\gamma}^l_n$ and $\widehat{\varepsilon}^l_{n^\dagger n}$.
2: **repeat**
3:  **for** $m \in \mathbf{M}$ and $n \in \mathbf{N}$ **do**
4:  Set $\widetilde{\alpha}^m_n \leftarrow 1$ with probability $\widehat{\alpha}^m_n$
5:  **end for**
6:  **for** $l \in \mathbf{L}$ and $n \in \mathbf{N}$ **do**
7:  Set $\widetilde{\gamma}^l_n \leftarrow 1$ with probability $\widehat{\gamma}^l_n$
8:  **end for**
9:  **for** $m \in \mathbf{M}$, $u \in \mathbf{U}$ and $n, n' \in \mathbf{N}$ **do**
10:  **if** $\widetilde{\alpha}^m_n = 1$ **then**
11:  Set $\widetilde{\beta}^{mu}_{n'n} \leftarrow 1$ with probability $\frac{\widehat{\beta}^{mu}_{n'n}}{\widehat{\alpha}^m_n}$
12:  **else**
13:  Set $\widetilde{\beta}^{mu}_{n'n} \leftarrow 0$
14:  **end if**
15:  **end for**
16:  **for** $l \in \mathbf{L}$ and $n, n^\dagger \in \mathbf{N}$ **do**
17:  **if** $\widetilde{\gamma}^l_{n^\dagger} = 1$ **then**
18:  Set $\widetilde{\varepsilon}^l_{n^\dagger n} \leftarrow 1$ with probability $\frac{\widehat{\varepsilon}^l_{n^\dagger n}}{\widehat{\gamma}^l_{n^\dagger}}$
19:  **else**
20:  Set $\widetilde{\varepsilon}^l_{n^\dagger n} \leftarrow 0$
21:  **end if**
22:  **end for**
23: **until** $(\widetilde{\alpha}^m_n, \widetilde{\beta}^{mu}_{n'n}, \widetilde{\gamma}^l_n, \widetilde{\varepsilon}^l_{n^\dagger n})$ define a feasible solution.
24: Set $\widetilde{\beta}^{mu}_{n'c} \leftarrow 1$ to schedule remaining requests to the cloud.

## IV. Randomized Rounding based Algorithm

### A. Algorithm Design

To tackle the computation complexity, we incorporate the JMDLS formulation and propose a randomized rounding (RR) based solution to pursue sub-optimal solutions. The algorithm is presented in Algorithm 1.

By relaxing the binary variables (i.e., $\alpha_n^m, \beta_{n'n}^{mu}, \gamma_n^l, \varepsilon_{n^\dagger n}^l$) to real ones in the range of $[0,1]$ in line 1, the original JMDLS formulation in ILP form is first relaxed into an LP that can be solved in polynomial time. The real number solutions $(\widehat{\alpha}_n^m, \widehat{\beta}_{n'n}^{mu}, \widehat{\gamma}_n^l, \widehat{\varepsilon}_{n^\dagger n}^l)$ obtained by solving the relaxed JMDLS can be considered as the probability of making a decision. Hence, the deployment and storage decisions, i.e., $\widetilde{\alpha}_n^m$ and $\widetilde{\gamma}_n^l$, are set to 1 with probability $\widehat{\alpha}_n^m$ and $\widehat{\gamma}_n^l$ in line 4 and line 7, respectively. The user requests should be processed locally with the highest priority. The remaining requests from user $u$ arriving at server $n'$ can be distributed to server $n$ deployed with $m$ (i.e., $\widetilde{\alpha}_n^m = 1$). Note that only when a microservice $m$ is deployed on $n$, corresponding requests can be processed. Hence, if $\widetilde{\alpha}_n^m = 1$, we set $\widetilde{\beta}_{n'n}^{mu}$ to 1 with probability $\frac{\widehat{\beta}_{n'n}^{mu}}{\widehat{\alpha}_n^m}$, as shown in line 11. Otherwise, $\widetilde{\beta}_{n'n}^{mu}$ is set to 0 in line 13. Similarly, the layer loading decisions of $\widetilde{\varepsilon}_{n^\dagger n}^l$ are made from line 16 to line 22. Note that due to the relaxation, the randomized rounded solutions may violate the resource capacity constraints. Hence, we check if any storage, computation, or link capacity constraint is violated. If so, we repeat the randomized rounding procedure from line 3 to line 22 until a feasible solution is found. Finally, all the requests that cannot be processed on edge servers will be distributed to the cloud.

### *B. Algorithm Analysis*

Let $Pr(\cdot)$ represent the probability of an event. The probabilities of the decision variables being set to 1 are $Pr[\widetilde{\alpha}_n^m = 1] = \widehat{\alpha}_n^m$ and $Pr[\widetilde{\gamma}_n^l = 1] = \widehat{\gamma}_n^l$. Note that, $\widetilde{\beta}_{n'n}^{mu}$ and $\widetilde{\varepsilon}_{n^\dagger n}^l$ are dependent on $\widetilde{\alpha}_n^m$ and $\widetilde{\gamma}_n^m$, respectively. The value of $\widetilde{\beta}_{n'n}^{mu}$ can be 1 only when $\widetilde{\alpha}_n^m = 1$ and $\widetilde{\varepsilon}_{n^\dagger n}^l$ can be 1 only when $\widetilde{\gamma}_n^m = 1$. That is,

$$\begin{aligned} Pr[\widetilde{\beta}_{n'n}^{mu} = 1] &= Pr[\widetilde{\beta}_{n'n}^{mu} = 1 | \widetilde{\alpha}_n^m = 1] Pr[\widetilde{\alpha}_n^m = 1] \\ &= \frac{\widehat{\beta}_{n'n}^{mu}}{\widehat{\alpha}_n^m} \widehat{\alpha}_n^m = \widehat{\beta}_{n'n}^{mu}, \end{aligned} \tag{12}$$

and

$$\begin{aligned} Pr[\widetilde{\varepsilon}_{n^\dagger n}^l = 1] &= Pr[\widetilde{\varepsilon}_{n^\dagger n}^l = 1 | \widetilde{\gamma}_{n^\dagger}^l = 1] Pr[\widetilde{\gamma}_{n^\dagger}^l = 1] \\ &= \frac{\widehat{\varepsilon}_{n^\dagger n}^l}{\widehat{\gamma}_{n^\dagger}^l} \widehat{\gamma}_{n^\dagger}^l = \widehat{\varepsilon}_{n^\dagger n}^l. \end{aligned} \tag{13}$$

Based on the above probabilities, we provide theoretical analysis on the solution quality of our RR algorithm.

**Lemma 1.** *The RR algorithm guarantees all the resource capacity constraints in expectation.*

*Proof.* Let us first check the storage capacity constraint. The expectation of total storage resource consumption on server $n$ can be calculated as

$$E(\sum_{l \in \mathbf{L}} \widetilde{\gamma}_n^l s_l) = \sum_{l \in \mathbf{L}} Pr[\widetilde{\gamma}_n^l = 1] s_l = \sum_{l \in \mathbf{L}} \widehat{\gamma}_n^l s_l \leq \Omega_n^S, \tag{14}$$

where the second equation holds because $Pr[\widetilde{\gamma}_n^l = 1] = \widehat{\gamma}_n^l$ and the last inequality follows constraint (6) in the relaxed JMDLS.

Similarly, we can get the expected computation, uplink, downlink resource requirements and microservice startup latency as

$$\begin{aligned} &E(\sum_{m \in \mathbf{M}} \sum_{n' \in \mathbf{N}} \sum_{u \in \mathbf{U}} \widetilde{\beta}_{n'n}^{mu} r_{n'm}^u \omega_m^C) \\ &= \sum_{m \in \mathbf{M}} \sum_{n' \in \mathbf{N}} \sum_{u \in \mathbf{U}} Pr[\widetilde{\beta}_{n'n}^{mu} = 1] r_{n'm}^u \omega_m^C \\ &= \sum_{m \in \mathbf{M}} \sum_{n' \in \mathbf{N}} \sum_{u \in \mathbf{U}} \widehat{\beta}_{n'n}^{mu} r_{n'm}^u \omega_m^C \leq \Omega_n^C, \end{aligned} \tag{15}$$

$$\begin{aligned} &E(\sum_{m \in \mathbf{M}} \sum_{n \in \mathbf{N}/\{n'\}} \sum_{u \in \mathbf{U}} \widetilde{\beta}_{n'n}^{mu} r_{n'm}^u \omega_m^S) \\ &= \sum_{m \in \mathbf{M}} \sum_{n \in \mathbf{N}/\{n'\}} \sum_{u \in \mathbf{U}} Pr[\widetilde{\beta}_{n'n}^{mu} = 1] r_{n'm}^u \omega_m^S \\ &= \sum_{m \in \mathbf{M}} \sum_{n \in \mathbf{N}/\{n'\}} \sum_{u \in \mathbf{U}} \widehat{\beta}_{n'n}^{mu} r_{n'm}^u \omega_m^S \leq \Omega_{n'}^U, \end{aligned} \tag{16}$$

$$\begin{aligned} &E(\sum_{m \in \mathbf{M}} \sum_{n' \in \mathbf{N}/\{n\}} \sum_{u \in \mathbf{U}} \widetilde{\beta}_{n'n}^{mu} r_{n'm}^u \omega_m^S) \\ &= \sum_{m \in \mathbf{M}} \sum_{n' \in \mathbf{N}/\{n\}} \sum_{u \in \mathbf{U}} Pr[\widetilde{\beta}_{n'n}^{mu} = 1] r_{n'm}^u \omega_m^S \\ &= \sum_{m \in \mathbf{M}} \sum_{n' \in \mathbf{N}/\{n\}} \sum_{u \in \mathbf{U}} \widehat{\beta}_{n'n}^{mu} r_{n'm}^u \omega_m^S \leq \Omega_n^D, \end{aligned} \tag{17}$$

and

$$\begin{aligned} &E(\sum_{l \in \mathbf{L}} \sum_{n^\dagger \in \mathbf{N}} \frac{\widetilde{\alpha}_n^m \theta_m^l \widetilde{\varepsilon}_{n^\dagger n}^l s_l}{R_{n^\dagger n}}) \\ &= \sum_{l \in \mathbf{L}} \sum_{n^\dagger \in \mathbf{N}} \frac{Pr[\widetilde{\alpha}_n^m = 1] Pr[\widetilde{\varepsilon}_{n^\dagger n}^l = 1] \theta_m^l s_l}{R_{n^\dagger n}} \\ &= \sum_{l \in \mathbf{L}} \sum_{n^\dagger \in \mathbf{N}} \frac{\widehat{\alpha}_n^m \theta_m^l \widehat{\varepsilon}_{n^\dagger n}^l s_l}{R_{n^\dagger n}} \leq \mathcal{D}_m, \end{aligned} \tag{18}$$

which follow constraints (3), (4), (5) and (10), respectively. Thus we prove that Algorithm 1 guarantees all the resource capacity constraints in expectation. □

**Lemma 2.** *The RR algorithm guarantees that deployed microservices are stored intact on the edge with high probability.*

*Proof.* The relationship between the probability of layer loading and that of microservice deployment is $\sum_{n^\dagger \in \mathbf{N}} Pr[\widetilde{\varepsilon}_{n^\dagger n}^l = 1] = \sum_{n^\dagger \in \mathbf{N}} \widehat{\varepsilon}_{n^\dagger n}^l \geq \widehat{\alpha}_n^m \theta_m^l = Pr[\widetilde{\alpha}_n^m = 1] \theta_m^l$.

Both the first and the third equations hold, due to (13) and $Pr[\widetilde{\alpha}_n^m = 1] = \widehat{\alpha}_n^m$ respectively, and the second inequality follows constraint (8). As such, we can see that the probability of a microservice being deployed will not be larger than the probability of its required layers being loaded. It implies that any required layer of a microservice to be deployed can be successfully loaded with high probability. As the layer loading is constrained by the layer storage, the deployed microservices are stored intact on the edge with high probability. □

**Lemma 3.** *The RR algorithm guarantees that the layers required by a server will not be loaded repeatedly with high probability.*

*Proof.* The sum of probability of loading layer $l$ to server $n$ is $\sum_{n^\dagger \in \mathbf{N}} Pr[\widetilde{\varepsilon}^l_{n^\dagger n} = 1] = \sum_{n^\dagger \in \mathbf{N}} \widetilde{\varepsilon}^l_{n^\dagger n} \leq 1$, where the last inequality holds due to constraint (9). Hence, a layer will not be loaded repeatedly with high probability. □

We can easily derive that the other constraints also always hold. As $\widetilde{\beta}^{mu}_{n'n}$ and $\widetilde{\varepsilon}^l_{n^\dagger n}$ are dependent on $\widetilde{\alpha}^m_n$ and $\widetilde{\gamma}^l_{n^\dagger}$, respectively, the constraints (1) and (7) hold. Constraint (2) always holds due to request distribution in line 24.

The above lemmas prove that the RR algorithm fits all the constraints in expectation. Hence, the achievable performance in terms of the edge throughput of our RR algorithm can be analyzed as follows.

**Theorem 1.** *The JMDLS problem can be solved by the RR algorithm with an approximation ratio of* $1 - \sqrt{\frac{2M}{\mathcal{OL}}}$*, where* $\mathcal{OL}$ *is the optimal value of the relaxed JMDLS in LP form.*

*Proof.* The RR algorithm returns the expectation of objective value as

$$\begin{aligned} &E(\sum_{m\in\mathbf{M}}\sum_{n\in\mathbf{N}}\sum_{n'\in\mathbf{N}}\sum_{u\in\mathbf{U}}\widetilde{\beta}^{mu}_{n'n}r^u_{n'm}) \\ &= \sum_{m\in\mathbf{M}}\sum_{n\in\mathbf{N}}\sum_{n'\in\mathbf{N}}\sum_{u\in\mathbf{U}}Pr[\widetilde{\beta}^{mu}_{n'n}=1]r^u_{n'm} \\ &= \sum_{m\in\mathbf{M}}\sum_{n\in\mathbf{N}}\sum_{n'\in\mathbf{N}}\sum_{u\in\mathbf{U}}\widehat{\beta}^{mu}_{n'n}r^u_{n'm}. \end{aligned} \tag{19}$$

Note that, each term $\widetilde{\beta}^{mu}_{n'n}r^u_{n'm}$ in (19) is an independent random variable, and can be set to 0 or 1 by appropriate normalization. Let $\mathcal{OL}$ represent the optimal value of relaxed JMDLS and we can apply the Chernoff Bound theorem [17] to prove that

$$Pr[\sum_{m\in\mathbf{M}}\sum_{n\in\mathbf{N}}\sum_{n'\in\mathbf{N}}\sum_{u\in\mathbf{U}}\widetilde{\beta}^{mu}_{n'n}r^u_{n'm} \leq (1-\delta)\mathcal{OL}] \leq e^{-\frac{\delta^2}{2}\mathcal{OL}}, \tag{20}$$

where $0 < \delta < 1$. Let $\mathcal{OP}$ represent the optimal solution of JMDLS problem. Obviously, $\mathcal{OP} \leq \mathcal{OL}$, based on which we can further derive that

$$\begin{aligned} &Pr[\sum_{m\in\mathbf{M}}\sum_{n\in\mathbf{N}}\sum_{n'\in\mathbf{N}}\sum_{u\in\mathbf{U}}\widetilde{\beta}^{mu}_{n'n}r^u_{n'm} \leq (1-\delta)\mathcal{OP}] \\ &\leq Pr[\sum_{m\in\mathbf{M}}\sum_{n\in\mathbf{N}}\sum_{n'\in\mathbf{N}}\sum_{u\in\mathbf{U}}\widetilde{\beta}^{mu}_{n'n}r^u_{n'm} \leq (1-\delta)\mathcal{OL}] \\ &\leq e^{-\frac{\delta^2}{2}\mathcal{OL}}. \end{aligned} \tag{21}$$

In order to make the upper bound of the probability as small as possible, we take

$$e^{-\frac{\delta^2}{2}\mathcal{OL}} \leq \frac{1}{e^M}, \tag{22}$$

which implies that the upper bound quickly converges to 0 as the number of microservices grows. Accordingly, $\delta$ should be

$$\delta \geq \sqrt{\frac{2M}{\mathcal{OL}}}. \tag{23}$$

In practice, $\mathcal{OL}$ tends to be much bigger than $M$, and therefore $0 < \delta < 1$ always holds when $\delta = \sqrt{\frac{2M}{\mathcal{OL}}}$. Consequently, the approximation ratio of the RR algorithm is $1 - \sqrt{\frac{2M}{\mathcal{OL}}}$. □

With regard to the randomness of the RR algorithm, we iteratively apply randomized rounding to find feasible solutions. One may concern whether the loop can stop and how many iterations are needed till termination. This is related to the probability of resource capacity constraint violation. Let us begin with the analysis on the storage capacity constraint.

**Lemma 4.** *The RR algorithm ensures that the layers' storage footprint in either server* $n \in \mathbf{N}$ *does not exceed* $\frac{1}{2}(\sqrt{\frac{s^{max}_l}{\Omega^S_n}}+1)(\sqrt{\frac{s^{max}_l}{\Omega^S_n}}+2)$ *times than its storage capacity with high probability, where* $s^{max}_l$ *represents the largest layer size.*

*Proof.* In the RR algorithm, we try to deploy as many microservices as possible in edge servers to improve the edge throughput. The layers are greedily stored in the edge cloud, expecting that $E(\sum_{l\in\mathbf{L}}\widetilde{\gamma}^l_n s_l) = \Omega^S_n$. Each term $\widetilde{\gamma}^l_n s_l$ is an independent variable that can be set to value 0 or 1 by appropriately normalizing. Therefore, also by applying the Chernoff Bound theorem, we can prove that

$$Pr[\sum_{l\in\mathbf{L}}\widetilde{\gamma}^l_n s_l \geq (1+\sigma)\Omega^S_n] \leq e^{-\frac{\sigma^2}{2+\sigma}\Omega^S_n}, \tag{24}$$

where $\sigma > 0$. Similarly, to make the upper bound small enough and follow the fact that the microservices are stored in layers, we take

$$e^{-\frac{\sigma^2}{2+\sigma}\Omega^S_n} \leq \frac{1}{e^{s^{max}_l}}, \tag{25}$$

where $s^{max}_l = \max\{s_l, \forall l \in \mathbf{L}\}$ represents the largest layer size. Now, $s^{max}_l$ makes the upper bound close to 0. Accordingly, $\sigma$ should satisfy

$$\sigma \geq \frac{s^{max}_l + \sqrt{(s^{max}_l)^2 + 8s^{max}_l\Omega^S_n}}{2\Omega^S_n} \tag{26}$$

In practice, $\Omega^S_n$ should be much larger than $s^{max}_l$. Without loss of generality, we set

$$\sigma = \frac{s^{max}_l + 3\sqrt{s^{max}_l\Omega^S_n}}{2\Omega^S_n}. \tag{27}$$

The threshold of storage capacity then can be derived as

$$1+\sigma = \frac{1}{2}(\sqrt{\frac{s^{max}_l}{\Omega^S_n}}+1)(\sqrt{\frac{s^{max}_l}{\Omega^S_n}}+2). \tag{28}$$

That is, the storage consumption of server $n$ does not exceed $\frac{1}{2}(\sqrt{\frac{s^{max}_l}{\Omega^S_n}}+1)(\sqrt{\frac{s^{max}_l}{\Omega^S_n}}+2)$ times than its storage capacity with high probability. □

**Lemma 5.** *By the RR algorithm, the computation resource consumption on a server will not exceed* $\frac{1}{2}(\sqrt{\frac{p^{max}}{C_{min}}}+1)(\sqrt{\frac{p^{max}}{C_{min}}}+2)$ *times than its computation capacity with high probability, where* $p^{max}$ *represents the maximum computation resource requirement of one user request and* $C_{min}$ *represents*

*the minimum resource consumption of one server in the relaxed JMDLS solution.*

*Proof.* Similar to the proof of Lemma 4, let $E(\sum_{m\in\mathbf{M}}\sum_{n'\in\mathbf{N}}\sum_{u\in\mathbf{U}}\widetilde{\beta}^{mu}_{n'n}r^u_{n'm}\omega^C_m) = C$. Through the Chernoff Bound theorem, we have

$$Pr[\sum_{m\in\mathbf{M}}\sum_{n'\in\mathbf{N}}\sum_{u\in\mathbf{U}}\widetilde{\beta}^{mu}_{n'n}r^u_{n'm}\omega^C_m \geq (1+\sigma)C] \leq e^{-\frac{\sigma^2}{2+\sigma}C}. \tag{29}$$

Note that the expectation of computation resource consumption may not be exactly equal to its capacity. According to (15), we have $C = \sum_{m\in\mathbf{M}}\sum_{n'\in\mathbf{N}}\sum_{u\in\mathbf{U}}\widehat{\beta}^{mu}_{n'n}r^u_{n'm}\omega^C_m \leq \Omega^C_n$. Let $C_{min}$ be the minimum resource consumption of one server in the relaxed JMDLS solution. Obviously, $C_{min} \leq C$, based on which we can derive that

$$\begin{aligned} &Pr[\sum_{m\in\mathbf{M}}\sum_{n'\in\mathbf{N}}\sum_{u\in\mathbf{U}}\widetilde{\beta}^{mu}_{n'n}r^u_{n'm}\omega^C_m \geq (1+\sigma)\Omega^C_n] \\ &\leq Pr[\sum_{m\in\mathbf{M}}\sum_{n'\in\mathbf{N}}\sum_{u\in\mathbf{U}}\widetilde{\beta}^{mu}_{n'n}r^u_{n'm}\omega^C_m \geq (1+\sigma)C] \\ &\leq e^{-\frac{\sigma^2}{2+\sigma}C} \leq e^{-\frac{\sigma^2}{2+\sigma}C_{min}}. \end{aligned} \tag{30}$$

Let $p^{max} = \max\{r^u_{nm}\omega^C_m, \forall n \in \mathbf{N}, \forall m \in \mathbf{M}, \forall u \in \mathbf{U}\}$ be the maximum computation resource requirement of one user request. We set $e^{-\frac{\sigma^2}{2+\sigma}C_{min}} \leq \frac{1}{e^{p^{max}}}$ and the value of $(1+\sigma)$ can be presented as $\frac{1}{2}(\sqrt{\frac{p^{max}}{C_{min}}}+1)(\sqrt{\frac{p^{max}}{C_{min}}}+2)$. □

Similarly, we can prove the violation of the uplink capacity, downlink capacity and microservice startup latency constraints as follows.

**Lemma 6.** *With the RR algorithm, the request traffic sent from a server will not exceed* $\frac{1}{2}(\sqrt{\frac{b^{max}}{B^U_{min}}}+1)(\sqrt{\frac{b^{max}}{B^U_{min}}}+2)$ *times of its uplink capacity with high probability, where* $b^{max} = \max\{r^u_{nm}\omega^S_m, \forall n \in \mathbf{N}, \forall m \in \mathbf{M}, \forall u \in \mathbf{U}\}$ *represents the maximum request transferring traffic and* $B^U_{min}$ *represents the minimum uplink traffic in the relaxed JMDLS solution.*

**Lemma 7.** *With the RR algorithm, the request traffic received by a server will not exceed* $\frac{1}{2}(\sqrt{\frac{b^{max}}{B^D_{min}}}+1)(\sqrt{\frac{b^{max}}{B^D_{min}}}+2)$ *times of its downlink capacity with high probability, where* $B^D_{min}$ *represents the minimum downlink traffic in the relaxed JMDLS solution.*

**Lemma 8.** *With the RR algorithm, the startup time of a microservice* $m \in \mathbf{M}$ *on any server will not exceed* $\frac{1}{2}(\sqrt{\frac{d^{max}}{D^{min}_m}}+1)(\sqrt{\frac{d^{max}}{D^{min}_m}}+2)$ *times than its threshold* $D_m$ *with high probability, where* $d^{max}$ *represents the maximum layer loading time and* $D^{min}_m$ *is the minimum microservice startup time in the relaxed JMDLS solution.*

In practice, RR can satisfy all resource constraints with extremely high probability, and usually only few iterations are needed in the execution of Algorithm 1. Taking storage resource as an example, with available storage capacity of 10TB ($\Omega^S_n = 1e7$) and the largest layer size as 200MB ($s^{max}_l = 200$), the storage consumption of the server will not likely to exceed its storage capacity, i.e., $\frac{1}{2}(\sqrt{\frac{200}{1e7}}+1)(\sqrt{\frac{200}{1e7}}+2) \approx 1.02$.

## V. Performance Evaluation

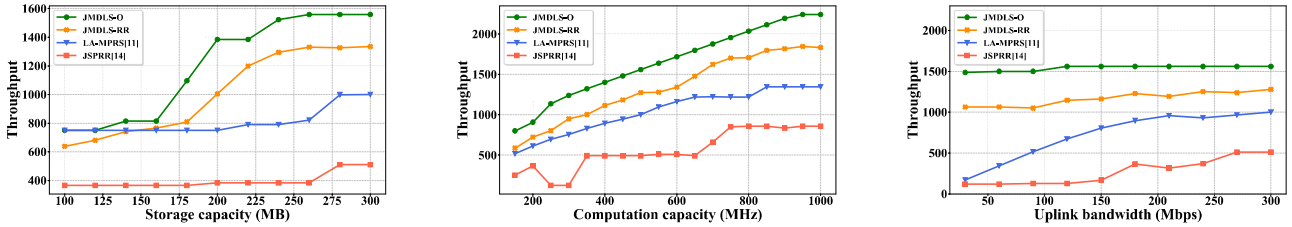
### A. In Small-scale Cases

To evaluate the optimality of our randomized rounding based algorithm, we first compare the RR algorithm (JMDLS-RR) to the optimal solution (JMDLS-O) in small scale such that the optimal solution via solving JMDLS in ILP form can be obtained as a baseline. Both intra-server layer-aware algorithm LA-MPRS [11] and JSPRR [14] are also implemented as competitors. In this case, we consider 4 distributed edge servers. In default, the available storage capacity, computation capacity, uplink capacity and downlink capacity are randomly set in the ranges of $109 \sim 384$MB, $0.8 \sim 1.2$GHz, $80 \sim 120$Mbps and $320 \sim 480$Mbps, respectively. 10 different microservices are selected from Docker Hub with the size of $67 \sim 740$MB. The microservice request arriving rates are set in the range of $1 \sim 132$RPS following a real world data set from IBM Docker Registry Trace Player. We vary the capacity of one resource in each experiment group to assess the performance of these algorithms and the influence of the resource capacity to the edge throughput. The results are reported in Fig. 2.

We first check how the storage capacity affects the achievable edge throughput by increasing it from 100MB to 300MB, as shown in Fig. 2(a). As expected, higher storage capacity implies more microservices on the edge and hence higher throughput. JMDLS-RR always performs close to JMDLS-O and outperforms both LA-MPRS and JSPRR, thanks to the exploration of both intra-server and inter-server layer sharing. Especially, the gap between JMDLS-RR and JMDLS-O is nearly zero when the storage capacity is small, i.e., 100MB to 160MB. This is because there is no much optimization space when the storage resource is too small, and the throughput of either algorithm is low. While, when the capacity increases from 260MB to 300MB, the throughputs of JMDLS-RR and JMDLS-O both converge because the computation and link capacity constraints become dominant in this case.

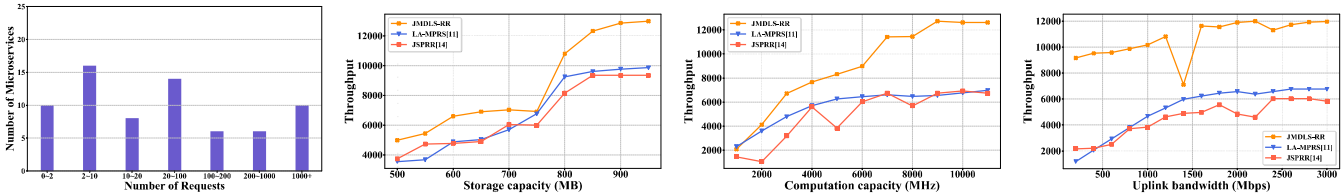
Then, Fig. 2(b) shows the edge throughput when the average computation capacity increases from 100MHz to 1000MHz. Without doubt that more requests can be handled at the edge with higher computation capacity under the same microservice deployment. This explains why the edge throughput by any algorithm almost linearly increases with the computation capacity. While, compared to JMDLS-O and LA-MPRS, JMDLS-RR and JSPRR are not that stable. This can be attributed to their basic algorithm, i.e., randomized rounding, which randomly rounds the real solutions to binary ones. Nonetheless, the sub-optimality of JMDLS-RR, as well as its advantage over LA-MPRS and JSPRR, can still always be observed.

Next, we present the throughput under different link capacities. Since the downlink capacity is usually relatively



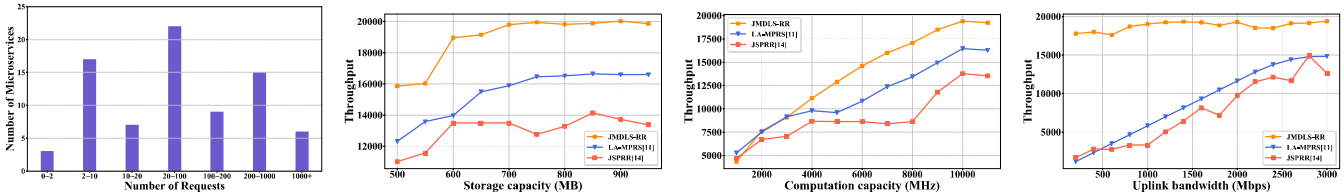

(a) Storage capacity (b) Computation capacity (c) Uplink capacity

Fig. 2. Impact of (a) storage capacity, (b) computation capacity and (c) uplink capacity on throughput in small-scale cases




(a) Request distribution (b) Storage capacity (c) Computation capacity (d) Uplink capacity

Fig. 3. Impact of (b) storage capacity, (c) computation capacity and (d) uplink capacity with IBM Docker Registry Trace #1




(a) Request distribution (b) Storage capacity (c) Computation capacity (d) Uplink capacity

Fig. 4. Impact of (b) storage capacity, (c) computation capacity and (d) uplink capacity with IBM Docker Registry Trace #2

larger, we take the uplink capacity as representative and vary it from 30Mbps to 300Mbps. The results are reported in Fig. 2(c). Besides the sub-optimality, we notice that the edge throughput of either JMDLS-O or JMDLS-RR is not affected too much by the uplink capacity but it grows along with the uplink capacity in the case of LA-MPRS or JSPRR. By carefully analyzing the results, we notice that most requests are handled locally in the cases of JMDLS-O and JMDLS-RR. With the help of inter-server layer sharing, the microservices' footprint is significantly reduced. Hence, many microservices can be successfully deployed and corresponding user requests can be handled locally as long as the computation resource is sufficient. Under such circumstance, enlarging the uplink capacity does not take much benefit to edge throughput. While, in LA-MPRS or JSPRR, due to comparatively larger storage requirement, less microservice instances can be deployed. Hence the requests that can not be handled locally have to be transferred to another server with enough computation resources, otherwise to the central cloud. Consequently, their performance is highly affected by the uplink capacity as higher uplink capacity implies that more locally unsatisfied requests can be transferred to other edge servers toward higher edge throughput.

### *B. In Large-scale Cases*

Then, to evaluate the performance of our proposed RR algorithm in more practical scenarios in larger scale, we consider an edge cloud consisting of 14 distributed edge servers with storage capacity, computation capacity, uplink capacity, downlink capacity and data access rate in the range of $109 \sim 1045$MB, $8 \sim 12$GHz, $320 \sim 2400$Mbps, $1280 \sim 1920$Mbps and $100 \sim 400$Mbps, respectively. 80 microservices from Docker Hub are considered. The number of layers of each microservice is in the range of $6 \sim 13$ and the image size is in the range of $109 \sim 2045$MB, with computation resource requirement ranging in $1 \sim 5$MHz. In this case, it is computationally prohibited to obtain the optimal solution anymore, and hence we only evaluate JMDLS-RR, LA-MPRS and JSPRR. Different microservice request traces from IBM Docker Registry Trace Player with rates varying from $1 \sim 1721$RPS are incorporated. The results of two traces under different storage, computation and uplink capacities are reported in Fig. 3 and Fig. 4, respectively. Fig. 3(a) and Fig. 4(a) first present the traces' request distribution. Taking Trace #2 as an example, most microservices (22 of them) are with request rates in the range of $[20, 100]$RPS and only a few (6 microservices) have rates more than 1000RPS.

First of all, we can see a similar trend from Fig. 3 and

TABLE I
A COMPARISON OF THREE METRICS ON LAYER AWARE AND UNAWARE STRATEGIES

| Traces | Storage | No. of Microservice Instances | | | Average No. of Layer Replicas | | | Throughput Improvement Ratio | |
|---|---|---|---|---|---|---|---|---|---|
| | | JMDLS | LA-MPRS | JSPRR | JMDLS | LA-MPRS | JSPRR | JMDLS vs LA-MPRS | JMDLS vs JSPRR |
| IBM Trace #1 | 500MB | 51 | 8 | 4 | 2.32 | 2.59 | 2.30 | 40.31% | 33.29% |
| | 600MB | 55 | 12 | 3 | 2.48 | 2.86 | 2.83 | 35.10% | 38.27% |
| | 700MB | 61 | 11 | 5 | 2.31 | 2.83 | 2.47 | 23.57% | 16.58% |
| | 800MB | 79 | 24 | 6 | 2.30 | 3.27 | 2.67 | 16.79% | 32.43% |
| | 900MB | 89 | 21 | 11 | 2.56 | 3.38 | 3.39 | 31.74% | 37.75% |
| IBM Trace # 2 | 500MB | 83 | 14 | 6 | 2.37 | 3.19 | 2.50 | 28.79% | 43.95% |
| | 600MB | 137 | 13 | 9 | 2.28 | 3.00 | 2.74 | 35.75% | 40.51% |
| | 700MB | 140 | 12 | 9 | 2.67 | 3.04 | 2.91 | 24.61% | 46.73% |
| | 800MB | 137 | 14 | 11 | 2.47 | 3.09 | 2.79 | 20.09% | 49.35% |
| | 900MB | 178 | 13 | 12 | 2.20 | 3.19 | 2.45 | 20.68% | 45.77% |

Fig. 4 as in small cases. It can be also observed that JMDLS-RR always achieves higher throughput than LA-MPRS and JSPRR, in any request pattern, for the same reason as just explained. One may notice that LA-MPRS is better than JMDLS-RR when the average computation capacity is 1000MHz (in Fig. 4(c)). This is because JMDLS-RR is based on randomized rounding, which may generate unsatisfactory decisions occasionally, especially when the resource capacity is relatively small. While, in practice, a typical 16-core edge server should have computation capacity higher than 32.00GHz. Such case rarely happens. In most cases, we can observe that JMDLS-RR always achieves the best performance.

To get an insight on the performance advantage of JMDLS-RR, we further report the average performance improvement of JMDLS-RR (written as JMDLS in the table) over LA-MPRS and JSPRR for the two traces under different storage capacities in Table I. It can be seen from the last column that JMDLS-RR always significantly outperforms LA-MPRS and JSPRR. As we have known, it is because, with inter-layer sharing, less layers need to be stored and more microservice instances can be deployed at the edge. One interesting finding observed from Table I is that the average number of common layer replicas by LA-MPRS sometimes surpasses JSPRR. By analyzing the deployment solutions, we find that LA-MPRS enables intra-server layer sharing and can save more space for small size microservice images. For example, for Trace #2 with 600MB storage, the smallest microservice $m_2$ is deployed on servers $s_2$, $s_5$, $s_7$ and $s_8$ to fully explore their storage and computation resources toward throughput maximization. Without inter-server layer sharing, this common layer has to be stored 4 times. Hence, the average number of layers stored in LA-MPRS is higher than JSPRR occasionally.

In summary, our JMDLS-RR can deploy $6\times$ and $12\times$ more microservice instances than LA-MPRS and JSPRR, respectively, subject to the resource capacity constraints, hence producing $27.74\%$ and $38.46\%$ throughput improvement over LA-MPRS and JSPRR, respectively.

## VI. RELATED WORK

Container-based microservice provision has emerged as a compelling solution to provide fast, flexible and scalable services [18]. Microservice deployment therefore has been widely investigated toward different optimization goals [19], [20]. To support the growing user demands, some studies try to maximize the throughput [21]. For example, Samanta et al. [22] study the microservice deployment problem in mobile edge cloud and design a microservice prioritization algorithm. Poularakis et al. [14] investigate the joint optimization of service deployment and request routing problem at the edge and also propose a randomized rounding based algorithm JSPRR to maximize edge throughput. To improve the user experience, the service latency should be considered [23]–[25]. Guo et al. [26] propose a microservice scheduling strategy based on particle swarm algorithm with the consideration of service neighborhood division to provide faster service response. Yu et al. [27] focus on the interdependency to minimize inter-microservice traffic as well as the service latency. Cost minimization is also another widely concerned objective [28]–[30]. Wang et al. [31] propose a reinforcement learning based algorithm to minimize service migration cost. Wang et al. [32] design a latency-aware task scheduling, container deployment and resource auto-scaling algorithm to minimize the service cost.

By literature survey, most existing microservice deployment related studies usually treat a microservice as a whole [33]–[35], and the layered structure of microservice images is rarely considered. Some recent studies point out that the service performance can be significantly improved by taking advantage of the layered structure. For example, Nathan et al. [36] design layer-level image registries to speedup microservice startup and Gu et al. [11] highlight intra-server layer sharing to improve the edge throughput. Zheng et al. [12] also show that image layer can be shared between microservices deployed on different servers, i.e., inter-server layer sharing. This provides us an opportunity to share common layers across nearby servers, further lowering the image downloading overhead and storage footprint. However, how to exploit such opportunity has not been discussed yet and our work is the first to investigate this issue.

## VII. CONCLUSION

In this paper, we consider the collaborative layer-aware microservice deployment and layer storage problem with the objective of maximizing edge throughput. By leveraging both intra-server and inter-server layer sharing, we jointly balance storage and computation resource usages in order to maximize

the service capacity of edge cloud. We formulate the problem into an ILP form and prove it as NP-hard by reducing from set-union knapsack problem. We propose a randomized rounding based heuristic algorithm with guaranteed approximation ratio. The efficiency of our proposed algorithm is verified through extensive experiments and the results demonstrate that it can deploy $6\times$ and $12\times$ more microservice instances and increase the edge throughput by $27.74\%$ and $38.46\%$, over state-of-the-art strategies LA-MPRS [11] and JSPRR [14], respectively.

## REFERENCES


[1] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in *Proceedings of IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 202–211.

[2] F. Ramalho and A. Neto, "Virtualization at the network edge: A performance comparison," in *Proceedings of IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2016, pp. 1–6.

[3] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171–172.

[4] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *Proceedings of Usenix Annual Technical Conference (ATC)*, 2018, pp. 923–935.

[5] L. Gu, Q. Tang, S. Wu, H. Jin, Y. Zhang, G. Shi, T. Lin, and J. Rao, "N-docker: A nvm-hdd hybrid docker storage framework to improve docker performance," in *Proceedings of IFIP International Conference on Network and Parallel Computing*, 2019, pp. 182–194.

[6] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight OS containers," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2018, pp. 199–212.

[7] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2016, pp. 181–195.

[8] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with serverless-optimized containers," in *Proceedings of USENIX Annual Technical Conference (ATC)*, Jul. 2018, pp. 57–70.

[9] Y. Li, B. An, J. Ma, and D. Cao, "Comparison between chunk-based and layer-based container image storage approaches: an empirical study," in *Proceedings of IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 197–202.

[10] L. Du, T. Wo, R. Yang, and C. Hu, "Cider: A rapid docker container deployment system through sharing network storage," in *Proceedings of IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2017, pp. 332–339.

[11] L. Gu, D. Zeng, J. Hu, B. Li, and H. Jin, "Layer aware microservice placement and request scheduling at the edge," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, 2021, online.

[12] C. Zheng, L. Rupprecht, V. Tarasov, D. Thain, M. Mohamed, D. Skourtis, A. S. Warke, and D. Hildebrand, "Wharf: Sharing docker images in a distributed file system," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 174–185.

[13] A. Samanta, L. Jiao, M. Mhlhuser, and L. Wang, "Incentivizing microservices for online resource sharing in edge clouds," in *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 420–430.

[14] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas, "Joint service placement and request routing in multi-cell mobile edge computing networks," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, 2019, pp. 10–18.

[15] G. Olivier, N. David, and Y. Gang, "Note: On the set-union knapsack problem," *Naval Research Logistics (NRL)*, pp. 833–842, 1994.

[16] A. David and Nehme-Haily, *The set-union knapsack problem*. ProQuest Dissertations Publishing, 1995.

[17] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[18] A. Celesti, L. Carnevale, A. Galletta, M. Fazio, and M. Villari, "A watchdog service making container-based micro-services reliable in iot clouds," in *Proceedings of IEEE International Conference on Future Internet of Things and Cloud (FiCloud)*, 2017, pp. 372–378.

[19] M. Karimzadeh-Farshbafan, V. Shah-Mansouri, and D. Niyato, "A dynamic reliability-aware service placement for network function virtualization," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 2, pp. 318–333, 2020.

[20] B. Dab, I. Fajjari, M. Rohon, C. Auboin, and A. Diquélou, "An efficient traffic steering for cloud-native service function chaining," in *Proceedings of Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020, pp. 71–78.

[21] H. M. Fard, R. Prodan, and F. Wolf, "Dynamic multi-objective scheduling of microservices in the cloud," in *Proceedings of IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, 2020, pp. 386–393.

[22] A. Samanta and J. Tang, "Dyme: Dynamic microservice scheduling in edge computing enabled iot," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6164–6174, 2020.

[23] S. Pallewatta, V. Kostakos, and R. Buyya, "Microservices-based iot application placement within heterogeneous and resource constrained fog computing environments," in *Proceedings of IEEE/ACM International Conference on Utility and Cloud Computing*, 2019, pp. 71–81.

[24] Y. Niu, F. Liu, and Z. Li, "Load balancing across microservices," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, 2018, pp. 198–206.

[25] Y. Wang, C. Zhao, S. Yang, X. Ren, L. Wang, P. Zhao, and X. Yang, "Mpcsm: Microservice placement for edge-cloud collaborative smart manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 9, pp. 5898–5908, 2020.

[26] Y. Guo and W. Yao, "A container scheduling strategy based on neighborhood division in micro service," in *Proceedings of NOMS IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1–6.

[27] R. Yu, V. T. Kilari, G. Xue, and D. Yang, "Load balancing for interdependent iot microservices," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, 2019, pp. 298–306.

[28] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, "Application deployment using microservice and docker containers: Framework and optimization," *Journal of Network and Computer Applications*, vol. 119, pp. 97–109, 2018.

[29] T. Zheng, X. Zheng, Y. Zhang, Y. Deng, E. Dong, R. Zhang, and X. Liu, "Smartvm: a sla-aware microservice deployment framework," *World Wide Web*, vol. 22, no. 1, pp. 275–293, 2019.

[30] J. Martín-Pérez, F. Malandrino, C.-F. Chiasserini, and C. J. Bernardos, "Okpi: All-kpi network slicing through efficient resource allocation," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, 2020, pp. 804–813.

[31] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE Transactions on Mobile Computing*, vol. 20, no. 3, pp. 939–951, 2021.

[32] S. Wang, Z. Ding, and C. Jiang, "Elastic scheduling for microservice applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 98–115, 2021.

[33] Z. Wen, T. Lin, R. Yang, S. Ji, R. Ranjan, A. Romanovsky, C. Lin, and J. Xu, "Ga-par: Dependable microservice orchestration framework for geo-distributed clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 129–143, 2019.

[34] J. Darrous, T. Lambert, and S. Ibrahim, "On the importance of container image placement for service provisioning in the edge," in *Proceedings of International Conference on Computer Communication and Networks (ICCCN)*, 2019, pp. 1–9.

[35] X. Hou, C. Li, J. Liu, L. Zhang, S. Ren, J. Leng, Q. Chen, and M. Guo, "Alphar: Learning-powered resource management for irregular, dynamic microservice graph," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 797–806.

[36] S. Nathan, R. Ghosh, T. Mukherjee, and K. Narayanan, "CoMICon: A co-operative management system for docker container images," in *Proceedings of International Conference on Cloud Engineering (IC2E)*, 2017, pp. 116–126.