SoCL: Scalable and Latency-Optimized Microservices in Serverless Edge Computing

Shuaibing Lu*¶, Bojin Xiang*, Jie Wu†‡, Ziyu You§, and Wentong Cai¶
*College of Computer Science, Beijing University of Technology, China
†China Telecom Cloud Computing Research Institute, China
‡Temple University, Philadelphia, USA
§Beijing Jiaotong University, China
¶Nanyang Technological University, Singapore

***Abstract*—Microservices have become an important design paradigm for large-scale distributed systems, offering flexible provisioning options. A fundamental challenge is the exponential growth of the solution space with the number of user requests, posing challenges to efficient provisioning and scheduling when aiming to balance cost and latency under resource constraints in large-scale dynamic edge environments. To tackle this problem, we formulate a joint optimization model for microservice provisioning and routing that integrates cost efficiency and latency reduction while accounting for uncertainties in the origin location of requests. To establish a unified framework that facilitates decision-making, we propose an integer linear programming (ILP) model that captures the dependencies between microservices in the service chain. Our Scalable optimization framework with Cost-efficiency and Latency reduction (SoCL) comprises three stages: an initial partitioning guarantees latency bounds, a pre-provisioning stage considers provisioning cost, and a multi-scale combination stage balances cost and latency through parallel and serial local search. Extensive experiments conducted across diverse scenarios based on a commonly used dataset demonstrate that the proposed SoCL framework significantly increases cost efficiency and decreases latency compared to established baselines, while reducing execution time up to one order of magnitude compared to obtaining the optimal solution by optimizer.**

***Index Terms*—cost-efficiency, latency reduction, microservice provisioning, scalable optimization, serverless edge computing**

I. Introduction

The rapid development of technologies such as the Internet of Things (IoT) and 5G has led to the emergence of serverless edge computing, a transformative paradigm that reduces reliance on centralized cloud services by deploying lightweight functions to the network edge. Microservices, as a dominant design paradigm for distributed systems, further amplify these benefits by decomposing monolithic applications into smaller, modular, and independently deployable services. Combined with containerization technologies (e.g., Docker) and orchestration tools (e.g., Kubernetes), microservices enable efficient resource provisioning, management, and scaling. Despite these advantages, efficiently managing microservices in large-scale dynamic environments remains a fundamental challenge.

Email: {lushuabing, sean-bonjean}@emails.bjut.edu.cn
Email: jiewu@temple.edu
Email: aswtcai@ntu.edu.sg

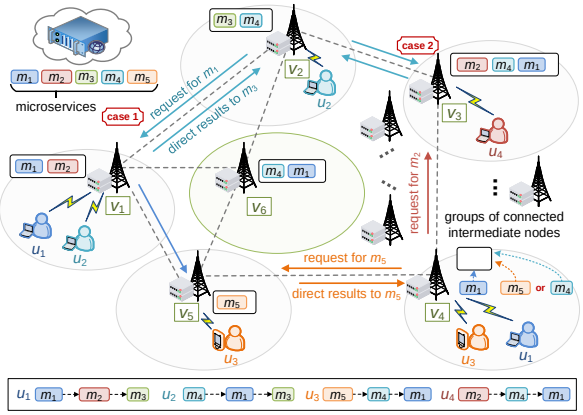
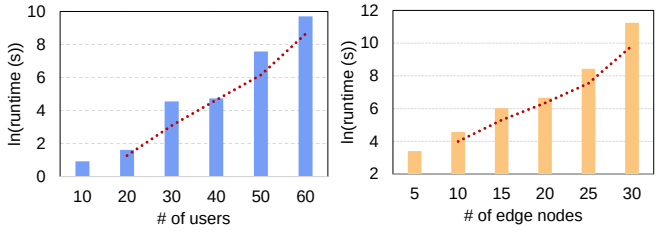


Fig. 1: An example illustrating challenges from user-side.

From the user's perspective, as illustrated in Figure 1, the joint optimization of provisioning and routing faces two key difficulties arising from user mobility and dynamic demand. ① **provisioning-adaption**: User mobility causes frequent and unpredictable shifts in request patterns, leading to dynamic trigger locations. Although overlapping requests offer opportunities to reduce redundancy, traditional static deployments lack adaptability, resulting in inefficient resource use. Dynamically selecting optimal instance locations is thus crucial to balance cost and latency in edge networks. ② **latency-optimized:** Once microservices (e.g.,m_1-m_5 in Figure 1) are deployed, efficient routing is critical to minimize latency and maintain user experience. Due to microservice interdependencies, requests require coordinated routing, often leading to path conflicts and network contention. Conventional strategies ignore these dependencies, increasing delay. Therefore, latency-aware routing that adapts to workload variations while minimizing end-to-end delay is essential.

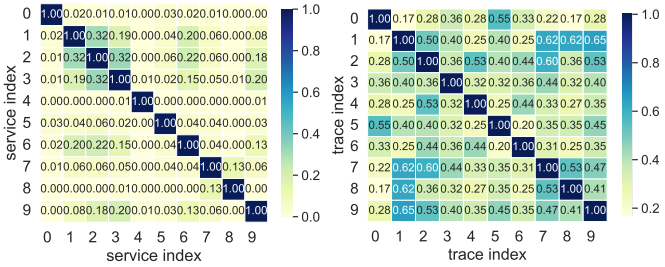
Meanwhile, from the operator's perspective, we identify several challenges on designing an efficient and robust microservice provisioning framework through detailed analysis of Alibaba Cluster traces and optimization complexity.

*1) **High complexity**:* Microservice provisioning and routing are more complex than traditional scheduling due to factors like network topology, resource constraints, and service dependencies. These interrelated variables create a high-dimensional optimization problem. Moreover, intricate de-



(a) users requests. (b) edge nodes.

Fig. 2: Runtime of optimal solutions using Gurobi.



(a) Similarity between services. (b) Similarity between different traces.

Fig. 3: Similarity comparison between services and traces.

pendencies among user requests require coordinated routing strategies to minimize latency and resource contention, further increasing the optimization difficulty. More importantly, the microservice provisioning and routing problem is NP-hard [3], and solving it directly as an integer linear programming (ILP) formulation becomes prohibitively expensive as the runtime scales exponentially with the number of edge servers and user requests. As shown in Figure 2, where the y-axis is plotted on a logarithmic scale, experiments using Gurobi with 10 to 30 edge servers show that runtime increased exponentially—over tenfold—as the number of users grew from 40 to 60, which highlights the need for scalable approaches.

*2) **Dynamic environment**:* The environment is inherently dynamic due to the diverse and time-varying access patterns of service demands from users. We analyzed microservice traces from the *Alibaba Cluster Trace Program* [25], [26] by selecting the 10 most frequently recorded services on a one-hour trace, to analyze similarity distributions across files. As shown in Figure 3 (a), the similarity values vary significantly across files, indicating a highly dynamic and heterogeneous service landscape. Further analyze each service that contains over 12 microservices in its dependency chain. As shown in Figure 3 (b), the maximum similarity of the traces is only 0.65, indicating diverse trigger points and dependency structures.

*3) **Unpredictable workload**:* User-generated requests exhibit high variability and randomness. During peak hours, intensive user requests can overwhelm the system, while off-peak periods lead to underutilized resources. To illustrate this, we collected 10-hour trace data from the *Alibaba Cluster Trace Program* and analyzed request volumes across time intervals. As shown in Figure 4, the request rates exhibit significant temporal fluctuations and recurring peaks, reflecting the unpredictable nature of demands. This time-varying workload complicates resource provisioning and calls for adaptive, load-aware strategies to maintain performance and cost-efficiency.

*4) **Limited resource**:* Edge servers are typically operated under stringent resource constraints, particularly in memory, e.g., edge servers from Alibaba have 8GB to 128GB memory size (Alibaba ENS general instances), which is way more scarce than the cloud servers. Limited memory capacity on edge nodes not only causes difficulty in arranging microservice instances, but also demand a lightweight algorithm to perform in a memory limited control node.

Motivated by these challenges, our proposed SoCL is designed for online microservice provisioning and routing. Rather than assuming stationary request patterns, SoCL processes decisions in a time-slotted manner, where at each time slot, it adapts to the observed system state and current user demand. The SoCL aims to address the above challenges through the following key features: ① **One-shot decision-making:** Our approach reacts promptly to mobility-induced changes, continuously responds to real-time user distributions and requests. ② **Dependency-based filtering:** The filtering mechanism can figure out the intrinsic dependencies inside the user requests and avoid detoured routing. ③ **Optimized for routing:** The partitioning method ensures each connectivity-based group has at least one instance, increasing the likelihood of finding a nearby edge server. ④ **Flexible storage planning:** The storage planning mechanism performs a deployment trade-off by instance priority, allowing more warm instances in the nearby area. This design enables SoCL to handle real-time fluctuations in service requests without relying on prior knowledge of future arrivals. The major contributions of this work include the following:

- We formulate an optimization model for microservices in serverless edge computing that integrates cost efficiency and latency reduction while accounting for user location uncertainty. To enable a systematic solution, we transform the model by introducing structured decision variables that capture dependencies among microservices in the service chain.
- We propose a novel framework, SoCL, consisting of three modules: initial partition, pre-provisioning, and multi-scale combination. The partition module clusters edge servers by electing candidate nodes based on origin locations, while the pre-provisioning module applies budget-based bounds to ensure practical feasibility under resource constraints. To further balance cost and latency, the multi-scale combination module integrates parallel and serial local search with a storage-aware planning mechanism, enabling fast, accurate decision-making and adaptive resource utilization.
- Extensive experiments were conducted to evaluate the performance of our SoCL against various benchmarks using a shared microservice dataset from the `eShopOnContainers` project. The results on the simulation platform and in Kubernetes indicate that the proposed SoCL significantly increases cost efficiency and decreases latency compared to established baselines, while reducing execution time in various scenarios.

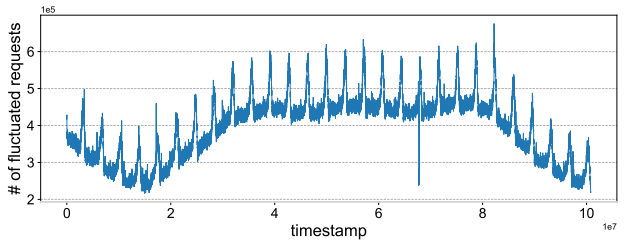


Fig. 4: Temporal distribution of user requests.

The remainder of this paper is organized as follows. Section II introduces related work. Section III describes the model and problem formulation. Section IV details the SoCL. Sections V and VI provide the evaluation and conclusion, respectively.

II. Related Work

In recent years, microservice provisioning based on dependency relationships has garnered significant attention due to its critical impact on system QoS. Existing studies primarily focus on two objectives: cost optimization and latency minimization. Some works emphasize the reduction of provisioning costs [1], [2], [4]–[6]. Deng et al. [1] optimized deployment costs under resource and performance constraints, while Fu et al. [2] proposed a runtime system to maintain QoS with minimal resource use. Yu et al. [4] introduced an auto-scaling framework for SLA compliance at minimal cost, and Tang et al. [5] employed Adam and weighted round-robin scheduling for dynamic optimization. He et al. [6] addressed service placement using greedy algorithms for fractional polynomial optimization. Although cost-effective, these methods often do not adequately account for the impact on latency.

Conversely, latency-aware studies focus on minimizing delays, leading to redundant microservice instances [7]–[11], [13]–[16]. He et al. [7] tackled deployment as an NP-hard problem with a two-stage greedy optimization method. Tian et al. [8] applied distributed reinforcement learning for placement in IoT environments. Zhao et al. [9] utilized a GA-based strategy for stochastic optimization under uncertain requests. Lv et al. [10] introduced a graph reinforcement learning framework to optimize dependencies via deep learning. Peng et al. [11] proposed a 2-approximation algorithm for joint deployment and routing. Liu et al. [13] employed neural networks for dynamic workload partitioning to reduce latency. Wang et al. [14] optimized vehicular networks with a placement mechanism to lower resource consumption. Ray et al. [15] have developed a method based on reinforcement learning for the placement and migration of microservices. Jia et al. [16] proposed a reinforcement learning based method for rearranging cache contents in CDN services. While these approaches drive cost or latency optimization, few pursue both objectives jointly.

Many request routing methods overlook the dependencies of microservices and underestimate their impact on data flow and latency [17], [18]. Yang et al. [17] have developed a distributed scheduling framework to optimize task acceptance under QoS constraints, while Zeng et al. [18] have proposed a parallel caching and routing algorithm extended to decentralized environments. With respect to microservice dependencies, probability-based strategies have been explored [1], [6], [7], [11], [19], [20] that utilize path probabilities for routing decisions. However, these methods often assume full connectivity of edge servers, which makes them sensitive to the settings of the probability model and less robust in practice. He and Deng [1], [6], [7] evaluated routing probabilities by analyzing node-level function satisfaction, while Chen et al. [19] applied Lyapunov optimization for service migration and dynamic routing. In this work, we optimize routing schedules while calculating latency, addressing both microservice dependencies and dynamic edge network conditions.

III. Model and Problem Formulation

A. System Model

Consider a substrate topology of an edge network, modeled as a weighted undirected graph $G(V, L)$. We use $V = \{v_k\}$ and $L = \{l_{k,k'}\}$ to represent the sets of edge servers and links, respectively. Here, v_k denotes the k-th edge server, and the computing capability of edge server v_k is represented as $c(v_k)$, measured in gflop/s. The edge link between servers v_k and $v_{k'}$ is represented by $l_{k,k'}$. Let $M = \{m_i\}$ denote the set of microservices, which are initially provisioned in the cloud data center upon user requests. The set of user requests is represented by $U = \{u_h\}$, where u_h corresponds to the h-th user request. Each user request is modeled as a directed chain of microservices, reflecting typical processing workflows: $u_h = \{M_h, E_h\}$. Here, $M_h = \{m_i\}$ represents the set of microservices involved in request u_h, and $E_h = \{e_{m_i \to m_j}\}$ denotes the directed communication links between them. Specifically, $e^h_{m_i \to m_j}$ captures the dependency between microservices m_i and m_j within the processing flow of u_h. The connection between users and edge servers is determined by the spatial distribution of users. Let U_k represent all user requests located within the coverage area of edge server v_k.

B. Cost Model

We define the deployment cost of a microservice instance m_i as $\kappa(m_i)$. Let $x(i,k)$ be a binary indicator, where $x(i,k) = 1$ if an instance of microservice m_i is deployed on edge server v_k, and $x(i,k) = 0$ otherwise. The total deployment cost on edge server v_k is then expressed as:

$$\mathcal{K}_k = \textstyle\sum_{m_i \in M} \kappa(m_i) \cdot x(i,k). \tag{1}$$

C. Completion Time Model

We use $\mathcal{D}_h$ to represent the completion time for the service request u_h, which encompasses both the processing time of the microservices and the communication time for data transfer between them. $\mathcal{D}_h$ is formally defined as:

$$\mathcal{D}_h = d^h_{in} + \sum_{m_i \in M_h} d^h_c(m_i) + \sum_{e_{m_i \to m_j} \in E_h} d^h_l(e_{m_i \to m_j}) + d^h_{out}. \tag{2}$$

Here, d^h_{in} denotes the latency experienced by a user in uploading a service request to the initial edge server v_s hosting the first microservice in the service chain requested by u_h. This metric captures the delay incurred in transmitting the

request from the user device to the designated edge server, which is given by $d_{in}^h = \mathbb{1}_{[v_k \neq v_s]} \cdot \sum_{l_{i,j} \in \pi(v_k, v_s)} r_{in}^h / b(l_{i,j})$. We assume that u_h is associated with the edge server v_k, where $u_h \in U_k$, and use an indicator function to ensure that the transmission time is only calculated if the associated edge server v_k of u_h is different from v_s, the edge server hosting the first microservice of the requested service chain. Specifically, the indicator function is defined as $\mathbb{1}_{[v_k \neq v_s]} = 1$ when $v_k \neq v_s$, and $\mathbb{1}_{[v_k \neq v_s]} = 0$ otherwise. The data size requested by user u_h is represented by r_{in}^h, and $b(l_{i,j})$ denotes the transmission rate of link $l_{i,j}$ along the path from v_s to v_k, which is expressed as $b(l_{i,j}) = B(l_{i,j}) \cdot \log_2(1 + \gamma \cdot g_{i,j}/N)$ [20]–[22]. Here, $B(l_{i,j})$ is the bandwidth value of the communication link and γ is the transmission power of the edge server v_i. $g_{i,j}$ is the channel gain between the edge servers equipped with m_i and m_j and N is the noise power. The processing time $d_c^h(m_i)$ for microservice quantifies the time required to execute m_i on edge server v_k and is defined as $d_c^h(m_i) = q(m_i)/c(v_k)$. Here, $q(m_i)$ represents the computing requirements of m_i, while $c(v_k)$ denotes the computational capacity of v_k, capturing the relationship between workload and server performance. We use $f : v_k \rightarrow \mathbb{R}$ to denote the physical edge server on which the user u_h is located, where $f(u_h) = k$. Additionally, $d_l^h(e_{m_i \rightarrow m_j})$ captures the communication delay between dependent microservices m_i and m_j, where $d_l^h(e_{m_i \rightarrow m_j}) = \sum_{l_{i,j} \in \pi(v_a, v_b)} r_{m_i \rightarrow m_j}^h / b(l_{i,j})$. Let $v_a \leftarrow \texttt{loc}^h(m_i)$ represent that v_a is the edge server chosen by user u_h to process m_i. Similarly, $v_b \leftarrow \texttt{loc}^h(m_j)$ represents the edge server responsible for the microservice m_j. To facilitate communication between two non-adjacent edge servers (i.e., servers that are not directly connected), we define a routing path $\pi(v_a, v_b)$ from v_a to v_b. Furthermore, we use d_{out}^h to represent the time taken to return the results to the user, where $d_{out}^h = \mathbb{1}_{[v_d \neq v_s]} \cdot \sum_{l_{i,j} \in \pi^*(v_d, v_s)} r_{out}^h / b(l_{i,j})$, which selects the shortest return path $\pi^*(v_d, v_s)$ for the transmission according to the minimum number of hops.

D. Problem Formulation

In this paper, we adopt the perspective of the service provider and aim to optimize the total costs of ownership while minimizing user response time over a long-term average period. By achieving this, we aim to improve the overall quality of service offered to users.

***Definition** 1 (objective function):* The problem is formulated as minimizing the weighted total cost of services ($\mathcal{K}_k$) and completion time ($\mathcal{D}_h$),

$$\min \quad \lambda \sum_{v_k \in V} \mathcal{K}_k + (1 - \lambda) \sum_{u_h \in U} \mathcal{D}_h. \tag{3}$$

To meet the QoS and the environment capacity, the objective is subject to several constraints, which are shown as follows:

***Definition** 2 (constraints):* The QoS maintenance constraints (4)-(5) ensure that the completion time and the provisioning costs do not exceed the maximum tolerance limits. The capacity constraint (6) restricts the edge storage and the binarity of the provisioning decision, which is strictly defined and cannot be violated.

$$\mathcal{D}_h \leq \mathcal{D}_h^{\max}, \forall u_h \in U \tag{4}$$

$$\sum_{v_k \in V} \mathcal{K}_k \leq \mathcal{K}^{\max}, \forall m_i \in M \tag{5}$$

$$\sum_{m_i \in M} x(i,k) \cdot \phi(m_i) \leq \Phi(v_k), x \in \{0,1\}, \forall i, \forall k \tag{6}$$

These constraints collectively ensure the feasibility and efficiency of the proposed optimization framework.

To better address the optimization problem, we reformulate the original model by introducing structured decision variables that normalize relationships among user service requests.

***Definition** 3 (decision variable):* The deployment decision variable $x(i,k) \in \{0,1\}$ indicates whether microservice m_i is deployed on edge server v_k, and the service decision variable $y(h,i,k) \in \{0,1\}$ denotes whether m_i serves request u_h on v_k. Here, we regard $d^h(m_i)$ as a *transmission-computation* cycle within a completion time of a particular user request, formulated as $d^h(m_i) = d_c^h(m_i) + d_l^h(e_{p \rightarrow m_i})$ where p indicates the preceding microservice. Each completion time consists of several cycles, depending on the number of microservices in the request dependency chain. Thus, the equation (2) can be reformulated as:

$$\mathcal{D}_h = \sum_{m_i \in M_h} \sum_{v_k \in V_i} y(h,i,k)(d^h(m_i) + d_{out}^h). \tag{7}$$

By integrating the decision variables, we further derive the reformulation as follows:

***Definition** 4 (reformulation):* The objective is reformulated as Equation (8), and the constraints are modified accordingly to accommodate the reformulation. Equation (9) enforces that only one microservice instance can be assigned to a particular user request at any given time. Equation (10) ensures the existence of an assigned instance, while Equation (11) restricts the binary of both deployment and assignment decisions.

$$\begin{aligned} \min \quad & \lambda \sum_{v_k \in V} \sum_{m_i \in M} \kappa(m_i) \cdot x(i,k) \\ & + (1-\lambda) \sum_{u_h \in U} \sum_{m_i \in M_h} \sum_{v_k \in V_i} y(h,i,k)(d^h(m_i) + d_{out}^h). \end{aligned} \tag{8}$$

$$\text{s.t.} (4) - (6)$$

$$\sum_{k \in V_i} y(h,i,k) = 1, \ \forall u_h \in U, \forall m_i \in M_h \tag{9}$$

$$y(h,i,k) \leq x(i,k), \quad \forall u_h \in U, \forall i, \forall k \tag{10}$$

$$x(i,k) \in \{0,1\}, y(h,i,k) \in \{0,1\}, \forall i, \forall k. \tag{11}$$

Following the above transformation, the original mixed-integer programming problem is reformulated as an integer linear program (ILP). However, as shown in Figure 2, solving it with an optimizer becomes computationally infeasible at scale due to the exponential increase in variables and constraints. To tackle this, we propose a decoupling approach and introduce a two-stage algorithm that partitions the original problem into two interconnected sub-problems: microservice provisioning and routing. By leveraging iterative interactions and solution exchanges between these sub-problems, the proposed approach progressively achieves joint optimization of the overall objective, resulting in effective microservice provisioning and routing solutions. Our approach partitions the large-scale edge network into regions based on a communication threshold,

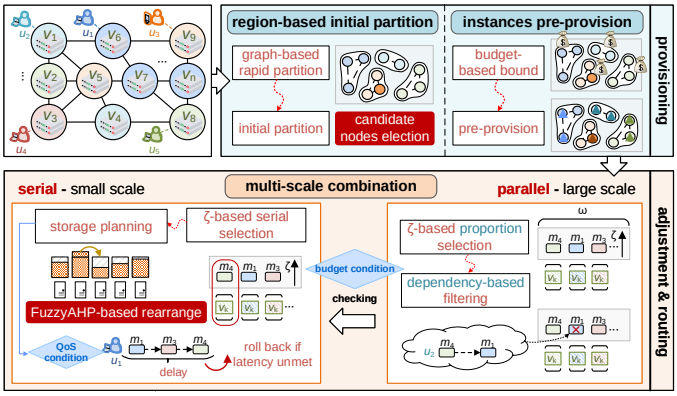


Fig. 5: The overview of the SoCL framework.

capturing both user locations and key hub nodes. A distributed and rapid deployment process is employed to achieve a coarse-grained allocation of microservices across these regions. Building on this initial allocation, a gradient-based merging strategy with backtracking is introduced to balance the trade-off between cost and latency, refining the placement and improving overall system performance.

IV. Algorithm Design

This section presents the three key steps of the microservice provisioning and routing process, forming the foundation of the SoCL framework, as depicted in Figure 5.

A. Region-Based Initial Partition

For large-scale user requests in microservice provisioning, directly searching for deployment and routing strategies across multiple edge servers results in an exponentially large search space and high computational complexity. To address this, we propose a region-based initial partitioning module that reduces the problem complexity while preserving flexibility for further optimization. This method constructs and reorganizes a virtual graph based on candidate nodes and incorporates user request locations and strategically important nodes. Coarse-grained microservice allocation is achieved by applying a threshold ξ to filter links with sufficient communication strength. Algorithm 1 provides step-by-step details and Figure 6 shows an example with m_1.

For each microservice $m_i \in M$, we identify all edge servers v_k hosting requests for m_i and collect these nodes in $V(m_i)$. The subgraph of G for m_i is represented as $G(m_i) = \{V(m_i), L(m_i)\}$, where $L(m_i)$ denotes the physical communication links. In practical scenarios, the spatial distribution of edge network nodes often results in indirect connectivity between certain nodes. To effectively model their relationships, virtual communication links are constructed using shortest-path routing, enabling accurate representation of data transmission. Let $l'_{k,q}$ denote the virtual link between edge nodes v_k and v_q, and let $\mathbb{B}(l'_{k,q})$ represent its channel speed, calculated as the harmonic mean of the bandwidths of all direct links within the shortest path $\pi^*(v_k, v_q)$. Specifically, $\mathbb{B}(l'_{k,q}) = 1/\sum_{l_{i,j}\in\pi^*(v_k,v_q)} \frac{1}{b(l_{i,j})}$, using these virtual links, the edge nodes in $V(m_i)$ are reconnected to form a virtual

Algorithm 1 Region-Based Initial Partitioning

Input: G, U, M;
Output: initial partition $\mathcal{P}$;

for each $m_i \in M$ **do**
  for each $v_k \in V$ **do**
    Construct the node set $V(m_i)$ of $G(m_i)$;
    Reconnect $G'(m_i)$ based on virtual links;
  for each $l'_{k,q} \in L'(m_i)$ **do**
    Add $l'_{k,q}$ to $G'(m_i)$ with $\mathbb{B}(l'_{k,q}) > \xi$;
  Generate the initial partitions $\mathcal{P}(m_i)$ by $G(m_i)$;
  for each partition $p_s(m_i) \in \mathcal{P}(m_i)$ **do**
    for each $v_k \in V \setminus V(m_i)$ **do**
      Choose v_k with $\mathcal{H}(v_k) > 2$;
      for each $v_a \in p_s(m_i)$ **do**
        Reorder $p_s(m_i)$ by $\arg\min\{\chi_{v_a}\}$;
      Check Δ^k with $v_a \in p_s(m_i)$;
      Add v_k to set $p_s(m_i)$ verified $\Delta^k < 0$;
Return initial partition $\mathcal{P}$;

graph $G'(m_i)$ with an updated set of links $L'(m_i)$. Based on $G'(m_i)$, a fast partitioning procedure is applied to group edge nodes into clusters using a threshold ξ. Virtual links with $\mathbb{B}(l'_{k,q}) > \xi$ are retained, ensuring sufficient communication strength while filtering out weaker connections to balance computational efficiency and connectivity. Subsequently, initial partitions are generated for $G(m_i)$, represented as $\mathcal{P} = \{\mathcal{P}(m_i)|_{m_i\in M}\}$. Edge nodes in $V(m_i)$ connected by $L(m_i)$ form node partitions $\mathcal{P}(m_i) = \{p_s(m_i)\}$, where s denotes the group number of each partition.

All further steps extend $V(m_i)$ to include nodes that, while not directly hosting user requests for m_i, provisioning m_i on them will benefit nearby users, minimizing the impact on completion time and reducing the spatial cost associated with m_i. These nodes are referred to as *candidate nodes*. To determine whether a node qualifies as a candidate node, a critical parameter is given as follows:

***Definition** 5 (proactive factor):* Let Δ^η denote the proactive factor of the edge server v_η, i.e. the expected deviation in the completion time of user requests resulting from the provision of the microservice m_i on v_η relative to $v_a \in p_s(m_i)$, which is calculated by the following equation:

$$\Delta^\eta = [\sum_{v_i\in p_s(m_i)} r_i/\mathbb{B}(l'_{i,\eta})]|_{m_i\to v_\eta} - [\sum_{v_i\in p_s(m_i)\setminus\{v_a\}} r_i/\mathbb{B}(l'_{i,a})]|_{m_i\to v_a}. \quad (12)$$

Here, r_i represents the total number of user requests for m_i on edge server v_i within the set $p_s(m_i)$. The channel speeds $\mathbb{B}(l'_{i,\eta})$ and $\mathbb{B}(l'_{i,a})$ are computed based on the shortest path between the respective edge nodes. If the instance of m_i is provisioned on v_η, i.e., $m_i \to v_\eta$, users on edge nodes in $p_s(m_i)$ must access v_η remotely, as m_i is not locally provisioned, resulting in an overall delay determined by the first term. Conversely, if m_i is placed on an edge node already hosting user requests, such as $v_a \in p_s(m_i)$, i.e., $m_i \to v_a$, the delay for users on v_a is eliminated since their requests

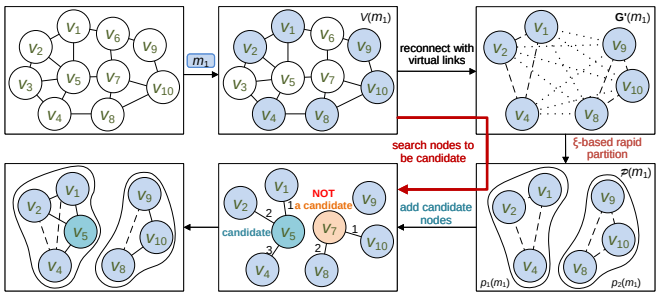


Fig. 6: Illustration of the region-based initial partitioning.

are served locally. In this case, only the delays for requests originating from other edge nodes are considered, as captured by the second term. Based on this analysis, we formally define the concept of a candidate node.

***Definition** 6 (candidate node):* A candidate node $v_\eta^{(m_i)}$ for microservice m_i is a node that does not directly host user requests but enhances communication efficiency by reducing completion time, satisfying $\Delta^\eta < 0$.

The filtering process begins to identify candidate nodes in lines 8-12 of Algorithm 1, which are subsequently assigned to their respective partitions $p_s(m_i)$. Line 9 selects nodes from the set $V(m_i)$ that satisfy $\mathcal{H}(v_\eta^{(m_i)}) > 2$, as established in Theorem 1, indicating their suitability as candidate nodes. The detailed proof is included in the appendix A for brevity. Line 13 further confirms a node as a candidate if it satisfies $\Delta^\eta < 0$, demonstrating its potential to improve the objective function. To increase the efficiency of the validation, the communication intensity χ_{v_k} is precalculated for each node to quantify the efficiency of communication between v_k and all other nodes, which is calculated as follows: $\chi_{v_k} = \sum_{v_q \in V \setminus \{v_k\}} \mathbb{B}(l'_{i,j})$, and the node v_q can be any other node except v_k. The checking process of Δ^η is carried out in ascending order of χ_{v_k}, prioritizing nodes with lower communication intensity, as they are more likely to satisfy the condition $\Delta^\eta < 0$. Once a node $v_a \in p_s(m_i)$ meets this condition, it is promptly identified as a candidate and added to the set, after which further calculations are terminated to enhance computational efficiency.

***Theorem** 1:* A candidate node $v_\eta^{(m_i)}$ must satisfy $\mathcal{H}(v_\eta^{(m_i)}) > 2$, where $\mathcal{H}(v_\eta^{(m_i)})$ denotes the number of direct connections that provide sufficient connectivity for $v_\eta^{(m_i)}$ to effectively serve as a candidate for microservice m_i.

B. Instance Pre-provisioning

The pre-provisioning module works before the combination and uses budget-based constraints to optimize the provisioning process. During this stage, an upper bound on the number of instances for each microservice is determined based on the maximum budget to ensure that resource constraints are guaranteed. Based on the region-based initial partitioning, this module evaluates the contribution of the microservices within each cluster and pre-positions the instances accordingly, narrowing the solution space and providing a basis for efficient provisioning of instances in the subsequent optimization phase.

1) budget-based bound: To determine the bound for each microservice, we first consider the total cost constraint $\mathcal{K}^{\max}$

Algorithm 2 Instance Pre-provisioning

Input: initial partition $\mathcal{P}$;
Output: pre-provisioning $\mathcal{P}^t$;

1: **for** each $v_k \in V$ **do**
2: **for** each $m_i \in M$ **do**
3: Calculate $\left|\mathbb{U}_{v_k}^{m_i}\right|$;
4: **for** each $\mathcal{P}(m_i) \in \mathcal{P}$ **do**
5: **for** each $p_s(m_i) \in \mathcal{P}(m_i)$ **do**
6: Update $\left|\mathbb{U}_{p_s(m_i)}\right|$;
7: Calculate $\varepsilon_s(m_i)$;
8: **if** $\varepsilon_s(m_i) \cdot \overline{\mathcal{N}}(m_i) \geq |p_s(m_i)|$ **then**
9: Pre-provision $p_s^t(m_i) \leftarrow m_i$;
10: **else**
11: **for** each $v_k \in p_s(m_i)$ **do**
12: Update $\mathbb{D}_{p_s(m_i)}(v_k)$;
13: **while** $|p_s^t(m_i)| < \varepsilon_s(m_i) \cdot \overline{\mathcal{N}}(m_i)$ **do**
14: Pre-provision with $\arg\min\{\mathbb{D}_{p_s(m_i)}(v_k)\}$;
15: **Return** pre-provisioning $\mathcal{P}^t$;

and the individual deployment cost $\kappa(m_i)$ of each microservice. We use $\mathcal{K}^u(m_i)$ to represent the maximum limited remaining budget for m_i, which is given by $\mathcal{K}^u(m_i) = \mathcal{K}^{\max} - \sum_{m_j \in M \setminus \{m_i\}} \kappa(m_j)$. Based on that, we use $\mathcal{N}^u(m_i)$ to denote the maximum tolerant number of instances of m_i, which is determined by the integer part, where $\mathcal{N}^u(m_i) = \lfloor (\mathcal{K}^{\max} - \mathcal{K}^\iota(m_i))/\kappa(m_i) \rfloor$. Since the deployment of microservices depends on the location of user requests, the quantity of each microservice based on $\overline{\mathcal{N}}(m_i)$ will not exceed the number of nodes $|V(m_i)|$ on which the user requests are located. Then the bound for m_i can be $\overline{\mathcal{N}}(m_i) = \min\{|V(m_i)|, \mathcal{N}^u(m_i)\}$.

2) instance pre-provisioning: Given the predetermined bound, we assign the estimated number of instances to each partition by systematically evaluating the spatial distribution of edge nodes and the origin of user requests. We integrate node locations and user demand patterns to ensure that instance allocation within each partition is both balanced and meets the practical requirements of microservice delivery. The detailed steps are shown in Algorithm 2. The initial partition $\mathcal{P}$ serves as input, while the resulting pre-provisioning $\mathcal{P}^t$ for instances is generated as output, where $\mathcal{P}^t = \{\mathcal{P}^t(m_i)|_{m_i \in M}\}$. For each edge node, we first analyze the patterns of user requests and the distribution of demand for each microservice connected to it in lines 1-3. We use $\mathbb{U}_{v_k}^{m_i} = \{u_h \in U_k|_{m_i \in E_h}\}$ to denote the set of users connected to edge node v_k and have requested microservice m_i. The cardinality $\left|\mathbb{U}_{v_k}^{m_i}\right|$ represents the total number of these users. Each user $u_h \in U_k$ contributes to $|\mathbb{U}_{v_k}^{m_i}|$ by requesting m_i as part of its service dependency chain. We then analyze the distribution of user requests and the characteristics of the demand within each partition. For a given partition $p_s(m_i)$, the total number of user requests is represented by the intermediate variable $\mathbb{U}_{p_s(m_i)}$, defined as $|\mathbb{U}_{p_s(m_i)}| = \sum_{v_k \in p_s(m_i)} |\mathbb{U}_{v_k}^{m_i}|$, which captures the total demand for m_i across all edge nodes in the partition. On this basis, we introduce a quota allocation strategy for each

Algorithm 3 Multi-scale Combination

Input: initial partition $\mathcal{P}$, pre-provisioning $\mathcal{P}^t$;
Output: provisioning $\mathcal{X}$;

while $\sum_{k=1}^{|V|} \mathcal{K}_k \geq \mathcal{K}^{\max}$ **do** ▷ parallel: large-scale
  Update $\zeta = \{\zeta_{i,k}\}$ in Algorithm 4;
  Update $\Omega = \{\Omega_{i,k}\}$ with ω;
  Filter Ω by service dependency E_h;
  Parallel combine $\Omega_{i,k} \in \Omega$ from $\mathcal{P}^t$;
while $\delta > 0$ **do** ▷ serial: small-scale
  Calculate Q' of $\mathcal{P}^t$;
  Choose $\Omega_{i,k} \leftarrow \arg\min_{\zeta_{i,k}} \mathcal{P}^t$;
  Calculate Q'' of $\mathcal{P}^t \setminus \{v_k\}$ and update δ;
  Serial combine $\Omega_{i,k}$ from $\mathcal{P}^t$;
  Storage planning with Algorithm 5;
  Constraint checking on latency with Equation (4);
  if $\mathcal{D}_h > \mathcal{D}_h^{\max}$ **then**
    Roll-back: add v_k back to $\mathcal{P}^t$;
    Continue the while loop;
Return provisioning $\mathcal{X}$;

Algorithm 4 UPDATE_INSTANCE_SET Function

Input: pre-provisioning $\mathcal{P}^t$;
Output: latency loss list ζ;

for each $m_i \in M$ **do**
  if $\sum_{p_s^t(m_i) \in \mathcal{P}^t(m_i)} |p_s^t(m_i)| = 1$ **then**
    Continue for: $M \leftarrow M/\{m_i\}$;
  for each $v_k \in \mathcal{P}^t(m_i)$ **do**
    Calculate $\zeta_{i,k}$;
  Update $\Omega_{i,k} \leftarrow \{m_i | \arg\min \zeta_{i,k}\}$;
Return latency loss list ζ;

partition based on the ratio of group demand, $\varepsilon_s(m_i) = \left|\mathbb{U}_{p_s(m_i)}\right| / \sum_{p_s(m_i) \in \mathcal{P}(m_i)} \left|\mathbb{U}_{p_s(m_i)}\right|$, ensuring an allocation that reflects the dynamics of user requests and supports efficient network operation in line 7. If the allocated quota is sufficient, i.e., $\varepsilon_s(m_i) \cdot \overline{\mathcal{N}}(m_i) \geq |p_s(m_i)|$, instances are directly pre-provisioned across all nodes within the partition in line 9. Conversely, if the quota is insufficient to meet the node count, a selection process is employed to determine specific placement locations within the partition in lines 10-14. This selection process is guided by an instance contribution-based approach, designed to prioritize decisions that minimize the overall completion time for the group.

***Definition** 7 (instance contribution):* The instance contribution $\mathbb{D}_{p_s(m_i)}(v_k)$ represents the estimated overall completion time for the group $p_s(m_i)$ if v_k is the only node hosting an instance of m_i, which is defined as:

$$\mathbb{D}_{p_s(m_i)}(v_k) = \sum_{f(u_h) \in p_s(m_i) \setminus \{v_k\}} r_i / \mathbb{B}(l'_{f(u_h),k}) + q(m_i)/c(v_k). \quad (13)$$

Here, $f(u_h)$ denotes other nodes within $p_s(m_i)$ that have users requesting m_i. $\mathbb{B}(l'_{f(u_h),k})$ represents the communication rate between the nodes, while $q(m_i)/c(v_k)$ corresponds to the computational delay of m_i at v_k. The contribution of

Algorithm 5 Storage Planning

Input: pre-provisioning $\mathcal{P}^t$;
Output: intermediate provisioning $\mathcal{X}'$;

if $\sum_{v_k \in V} \Phi(v_k) \geq \sum_{m_i \in M} |\mathcal{P}^t(m_i)| \cdot \phi(m_i)$ **then**
  for each $m_i \in M$ **do** ▷ FuzzyAHP progress
    Record $\kappa(m_i), \phi(m_i)$;
    for each $v_k \in V(m_i)$ **do**
      for each $u_h \in U_k$ **do**
        Calculate and record $\left|\mathbb{U}_{v_k}^{m_i}\right|, \mathbb{R}_{v_k}^{m_i}$;
  Calculate local demand factor ρ;
  for each $v_k \in V$ **do**
    while $\sum_{m_i \in M} x(i,k) \cdot \phi(m_i) > \Phi(v_k)$ **do**
      Choose m_j with $\arg\min \rho_k$;
      Reorder $v_q \in V \setminus \{v_k\}$ by $\arg\max b(l_{k,q})$
      if $x(j,q) = 0$ and $\phi(m_j) \leq \overline{\Phi}(v_k)$ **then**
        Let $x(j,k) = 0$ and $x(j,q) = 1$;
        Continue the while loop;
  Record current placement $x(i,k)$ as $\mathcal{X}'$;
else
  Continue the while loop in Algorithm 3;
Return intermediate provisioning $\mathcal{X}'$;

m_i provisioning to v_k is inversely related to the value of $\mathbb{D}_{p_s(m_i)}(v_k)$; a smaller $\mathbb{D}_{p_s(m_i)}(v_k)$ means a greater potential to reduce the overall completion time for the group. Consequently, when selecting nodes for pre-provisioning, those with the smallest $\mathbb{D}_{p_s(m_i)}(v_k)$ values are preferred. When deploying microservices within a partition, this parameter serves as a critical metric for determining the suitability of node placement and ensures efficient allocation aligned with performance objectives.

C. Multi-scale Combination

The module in this subsection aims to minimize the objective function by combining instances at varying granularities: large-scale gradient descent merges multiple instances simultaneously, while small-scale gradient descent processes instances individually to identify the optimal trade-off.

An instance combination involves merging two instances of the same m_i into one to reduce provisioning costs, which may disrupt existing dependencies, requiring users to establish new connections to maintain their dependency chains – a process termed connection update. To quantify the latency changes, we use a mapping function $\psi : \mathcal{P}^t(m_i) \to \mathbb{R}$, representing the latency associated with user requests in the microservice-to-node connection. Here, $\mathcal{P}'^t(m_i)$ denotes the provisioning nodes before the removal of v_k, with latency calculated as $\psi(\mathcal{P}'^t(m_i)) = \sum_{u_h \in U_{i,k}} (r_i^h / b(l_{f(u_h),k}) + q(m_i)/c(v_k))$. After the removal of v_k, $\mathcal{P}''^t(m_i)$ represents the updated provisioning set, where users with m_i requests must find a new reliance through a connection update.

The new reliance node v_q for user u_h must satisfy three criteria: (1) $f(u_h)$ and v_q belong to the same group $p_s(m_i)$; (2) $v_q \in p_s^t(m_i)$, indicating inclusion in the updated provisioning

set; and (3) v_q has the highest channel speed with respect to $f(u_h)$, i.e., $v_q = \arg\max_{v_q \in p_s^t(m_i)} b(l_{f(u_h),q})$.

The selection of v_q is independent for each u_h, and depends on their current routings. The latency of the updated set $\mathcal{P}^{''t}(m_i)$ is given by $\psi(\mathcal{P}^{''t}(m_i)) = \sum_{u_h \in U_{i,k}} (r_i^h / \max_{v_q \in p_s^t(m_i)} b(l_{f(u_h),q}) + q(m_i)/c(v_q))$, where $U_{i,k} = \{u_h \mid v_k \leftarrow \texttt{1oc}^{u_h}(m_i)\}$ represents users relying on instance m_i at v_k for processing requests. The progress above will lead to an increase in total completion time, quantified as the latency loss.

***Definition** 8 (latency loss):* The latency loss $\zeta_{i,k}$ is defined as the increase in total completion time incurred when instance m_i is removed from edge node v_k, which is expressed as:

$$\zeta_{i,k} = \psi(\mathcal{P}^{''t}(m_i)) - \psi(\mathcal{P}^{'t}(m_i)). \tag{14}$$

The two-scale gradient descent combination is outlined in Algorithm 3. During the large-scale gradient descent in lines 1-5, we combine the instances of microservices in parallel. The operation in line 2 verifies and updates the microservices within the instance set in Algorithm 4. The input is the current provisioning state $\mathcal{P}^t$, and the output is a latency loss list $\zeta = \{\zeta_{i,k}\}$, sorted in descending priority order for each microservice. If no edge node hosts an instance of m_i, all user requests u_h connected to m_i will fail or have to rely on the cloud servers as a fallback option. To prevent this scenario, the microservice is skipped when the instance number of m_i is reduced to one, and m_i is excluded from further combination to ensure service continuity. We then calculate the latency loss for each instance on the edge servers, update the priorities of m_i in descending order, and return the latency loss list. On this basis, the ω is proposed as a hyperparameter to regulate the number of parallel combinations in line 3 of Algorithm 3, which limits the speed of parallel gradient descent. We update $\zeta_{i,k}$ of each instance and find the ω minimal percentage of instances $\Omega_{i,k}$ as the instance set Ω for the parallel combine. From these instances in Ω, we discard instances that have a dependency conflict in any user dependency chain in line 4, i.e, if a user u_h has a dependency chain E_h that contains $e^h_{m_i \to m_j}$, we call m_i and m_j dependency-conflicted, then we compare $\zeta_{i,k}$ and $\zeta_{i,k}$ and discard the instance $\Omega_{i,k}$ with a larger $\zeta_{i,k}$. Following the above procedure, the instances in the set Ω can be combined in parallel in line 5.

During the small-scale gradient descent progress in lines 6-15, the algorithm focuses on the trade-off between deployment cost and completion time. The calculation of $\zeta_{i,k}$ remains the same in line 8, but the instances are combined one by one in line 10 to get a smaller gradient. We calculate the value of the objective function of Equation (8) in lines 7 and 9 to get the gradient, which is evaluated by Q. Here, we conduct the serial refinement per microservice instance through small-scale gradient δ, where $\delta = Q' - Q'' + \Theta$. Line 7 calculates the value of the objective function Q' before the combination of the instance, and Q'' is the value after the combination in line 9 respectively. Having the gradient δ above, we regard $\delta < 0$ as the sign that the objective function starts to rise back, where Θ is a positive disturbance factor to prevent a small rise back to stop the whole progress.

The solution from each round of small-scale gradient descent may violate the storage constraint (Equation (6)) or the user completion time constraint (Equation (5)) if instances relied upon by certain areas are combined. To address this, a storage planning process (Algorithm 5) is applied in line 11, followed by a roll-back mechanism in lines 12 to 15. The process identifies less critical microservices on v_k based on the local demand factor in line 10, as defined below.

***Definition** 9 (local demand factor):* The local demand factor $\rho = \{\rho_k\}$ reflects the importance of providing instance m_i on server v_k, where $\rho_k = \{\rho_{v_k}^{m_i}\}$ denotes the priority list. A higher $\rho_{v_k}^{m_i}$ means a higher priority for the provision of m_i.

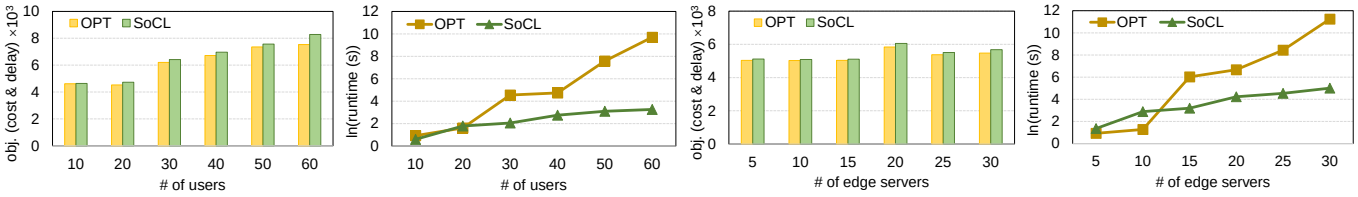
The factor ρ is calculated using the FuzzyAHP method in lines 2-7. The fuzzy properties of microservice m_i consider the deployment cost $\kappa(m_i)$, storage requirement $\phi(m_i)$, number of requesting users $\left|\mathbb{U}_{v_k}^{m_i}\right|$ and the order factor $\mathbb{R}_{v_k}^{m_i}$. To clarify, the ordering property $\mathbb{R}_{v_k}^{m_i}$ quantifies whether m_i is located as the first or last position within the user dependency chain, calculated as $\mathbb{R}_{v_k}^{m_i} = 3{uf}_{v_k}^{m_i} + 2{ul}_{v_k}^{m_i} + {um}_{v_k}^{m_i} / \left|\mathbb{U}_{v_k}^{m_i}\right|$, where ${uf}_{v_k}^{m_i}$, ${ul}_{v_k}^{m_i}$, and ${um}_{v_k}^{m_i}$ denote the number of users for whom m_i is the first, last, and intermediate dependency, respectively.

Having the less important instance m_i selected, we choose the target edge server nearby in line 11 with the fastest channel speed and try to migrate m_i on it if the target server does not have the same instance and its remaining storage $\overline{\Phi}(v_k)$ is capable of receiving the new microservice, which is checked in line 12. If no edge server can receive the instance, indicating that current storage is insufficient, line 17 will return to Algorithm 3 and continue the combination process to further reduce the number of instances. The roll-back mechanism in lines 12-15 will put the last combined instance back and not combine it during the later combination rounds when there is any violation of user completion time. The computational complexity of Algorithm 3 is $O(|M| \cdot |V|^3)$, where $|M|$ is the number of microservices and $|V|$ is the number of edge servers. Traversals over microservices, partitions, and sub-partitions, along with instance contributions and resource adjustments, collectively result in this worst-case complexity.

V. Evaluations

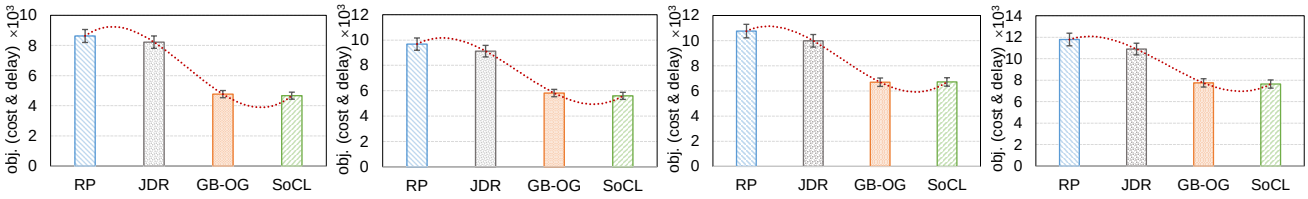
A. Basic Setting

Our prototype was implemented in Python 3.7 on a Windows 10 platform with an Intel(R) Xeon(R) Silver 4210R CPU, NVIDIA RTX5000 GPU, and 128GB RAM. The dataset, derived from Microservices (Version 1.0) [23], includes dependency analyses for 20 microservice-based projects. For evaluation, we used the `eShopOnContainers` project, configuring microservices with processing capabilities of [1, 3] GFLOPs and data flows of [1, 80] GB/s. Edge servers were equipped with [5, 20] GFLOPs computing power, [4, 8] storage units, and [20, 80] GB/s bandwidth. Base station locations were set near the National Stadium, in Beijing. Tests covered scenarios with 10 to 60 users (in increments of 10) and cost constraints between 5000 and 8000. Our SoCL was compared against Random Provisioning (RP), Joint Deployment



(a) # of users (obj). (b) # of users (runtime). (c) # of edge servers (obj). (d) # of edge servers (runtime).

Fig. 7: Gurobi (OPT) and SoCL: runtime and objective value under varying users and edge servers.



(a) # of users 80. (b) # of users 120. (c) # of users 160. (d) # of users 200.

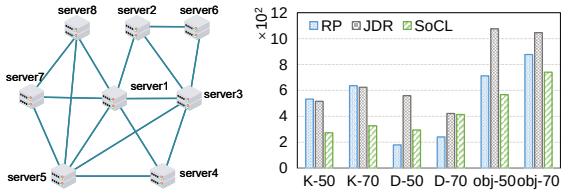
Fig. 8: obj. (cost & latency) for different numbers of user requests (10 servers).

and Routing (JDR) [11], and Greedy Combine with Objective Gradient (GC-OG), demonstrating robust performance under varying conditions.

B. Experiment Results

1) Comparison with Optimizer: We compared the performance of the Gurobi solver and SoCL at various user scales, focusing on objective value and runtime. The results in Figures 7 (a)-(d) show that although Gurobi achieves slightly better objective values, the differences are minimal. For example, at a user scale of 30, Gurobi's objective value is 6207.173 compared to SoCL's 6415.119, a difference of approximately 3.3%. However, SoCL significantly outperforms Gurobi in terms of runtime, especially as the user scale increases, as shown in Figures 7 (b) and (d). For 50 users, Gurobi requires 1958.646 seconds, while SoCL completes the task in just 22.3 seconds, achieving a speed improvement of two orders of magnitude. These results underscore SoCL's ability to deliver near-optimal solutions with significantly higher computational efficiency and demonstrate its scalability and practicality for large-scale scenarios where Gurobi's performance is no longer feasible.

We conducted extensive experiments on varying user scales (10 to 60 users) and edge node scales (5 to 30 nodes) to systematically compare the Gurobi solver (OPT) and SoCL in terms of objective value and runtime. The results indicate that while Gurobi can achieve the exact optimal solution for small-scale problems, its runtime increases exponentially as the number of users and edge nodes grows, making it impractical for large-scale scenarios. In contrast, SoCL achieves results very close to the optimal objective value while significantly reducing the solving time, demonstrating superior computational efficiency and scalability. For user scale variations, SoCL performs remarkably well. As the user scale increases to 50, Gurobi's runtime escalates dramatically to 1958.646 seconds, while SoCL maintains a low runtime of 22.3 seconds, further highlighting its computational efficiency. Similar trends are observed for edge node scale variations. When the number of edge nodes reaches 30, Gurobi's solving time skyrockets to



(a) edge network topology. (b) obj value.

Fig. 9: Results on Kubernetes with 8 edge servers.

76447.736 seconds, SoCL solves the same problem in just 149.04 seconds, achieving a speedup of several orders of magnitude. In terms of the objective value, SoCL remains competitive. For example, when the edge node count is 10, Gurobi obtains an objective value of 5025.102, while SoCL achieves 5085.852, with minimal deviation.

2) Comparision on Baselines: We compared the performance of four algorithms under user scales of 80, 120, 160, and 200, and conducted an in-depth analysis of the results. The results can be seen in Figures 8. Overall, SoCL achieved the lowest objective values across all user scales, demonstrating superior optimization capability and computational efficiency. The difference in performance between the algorithms became clearer as the user scale increased. The RP algorithm performed the worst due to its random placement and routing strategy, which led to highly unbalanced resource allocation and failed to optimize both provisioning costs and latency. As seen in Figure 8 (d), when the user scale reaches 200, RP exhibited the highest objective value of 11,785.1, reflecting its inability to handle large-scale scenarios due to unstructured random strategies. JDR attempted to optimize latency by categorizing microservices into single-user and multi-user groups, deploying the former close to user nodes and prioritizing the latter on high-capacity servers. However, by neglecting provisioning costs, JDR caused resource redundancy that led to consistently high objective values. For example, with 120 users in Figure 8 (b), JDR achieved an objective value of 9,123.2, lower than RP but still significantly higher than SoCL, with redundancy worsening as the number of users increased. GC-OG, which combines greedy strategies with objective gradient

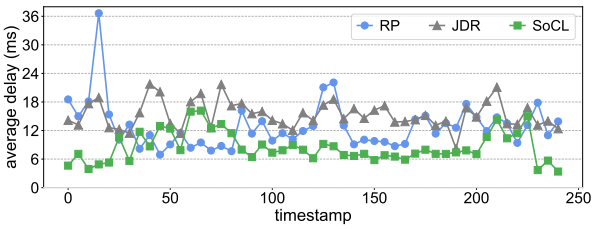


Fig. 10: Avg delay trace on Kubernetes with 16 edge servers.

descent, performed better by selecting instance combinations that most effectively reduced objective values. However, its low search efficiency became a limiting factor as user requests grew, resulting in an exponentially growing search space. With 120 users in Figure 8 (b), GC-OG achieved an objective value of 5,817.2, outperforming RP and JDR, but still outperforming SoCL while requiring 2,274.8 seconds of runtime, highlighting its inefficiency in search speed.

With the scaling of the user number, RP's objective value grew linearly and even accelerated, indicating its failure to scale to larger problems. JDR showed continuous increases due to redundant deployments and poor resource utilization.GB-OG worked well at small user scales, but as requests and budgets increased, its performance declined due to the exponential search space growth, with rising objective values that stayed above SoCL. In contrast, SoCL consistently achieved better results, with modest increases in objective value. As the user scale increased from 80 to 200, SoCL's objective value grew from 4669.58 to 7636.863, a significantly smaller increase compared to the other algorithms. This indicates that SoCL's efficient search and microservice provisioning strategy effectively avoids resource redundancy and search inefficiency, ensuring a balanced optimization of deployment cost and latency. Even under large-scale user requests, SoCL maintains low objective values and computational efficiency.

C. Evaluation on Kubernetes

1) Testbed Configurations: To further validate SoCL's effectiveness, we implemented a prototype on Kubernetes using Python 3.11-slim on CentOS 7.6. The experiment involved 17 machines, each with 2 GB of memory, 2 cores, and 1-2 Gbit/s bandwidth. Sixteen machines served as edge nodes providing computing services, while one acted as the master node for scheduling and latency recording. As shown in Figure 9 (a), we first implemented a small-scale evaluation under 8 edge nodes and one master node to evaluate the objective and their related delay and cost using the RP, JDR, and SoCL methods. Then, under 16 edge nodes and one master node, we traced user access latency over 4 hours, during which users issued requests every 5 minutes on average. Latency data was aggregated at the same interval to evaluate system performance.

2) Testbed Results: The algorithms RP, JDR, and SoCL are executed in our testbed. As shown in Figure 9 (b), we compared the total provisioning cost and latency of each algorithm under 50 and 70 users. The experimental results show that although the RP and JDR algorithms achieve a low completion time, this result depends on the full utilization of the maximum deployment cost constraint, which increases the total deployment cost and negatively affects the value of the objective function. Instead, the proposed SoCL method can balance the cost of deploying instances with the users' requirements and place the instances in the right place to increase the benefit. Meanwhile, as the user number increases, the total completion time of each algorithm rises accordingly, but we are surprised to find that the total latency of JDR begins to fall. We consider this to be a coincidence, as the user dependency was randomly generated. Furthermore, we analyzed the latency of the individual users. The median values of the RP, JDR, and SoCL algorithms are 2.795, 3.989, and 2.796 for 50 users and 2.528, 3.572, and 2.455 for 70 users, respectively, indicating that the SoCL method provides fewer instances on the edge servers but still serves most user requests well. We also conducted a 4-hour trace experiment on Kubernetes, where 50 users randomly moved among edge nodes and issued requests every 5 minutes with stochastic service dependencies. Figure 10 shows that SoCL consistently achieved the lowest average delay, around 8.50 ms per timestamp, outperforming RP and JDR. Although RP appeared better than JDR overall, its randomness caused unstable peaks (e.g., at timestamp 4), affecting QoS. JDR's delay was comparable to RP's but still inferior to SoCL. Evaluating delay stability via maximum latency, SoCL again excelled, with a maximum delay of just 48.84 ms, significantly lower than JDR's 90.04 ms and RP's 77.29 ms.

VI. Conclusion

This paper addresses the critical challenge of microservice provisioning and routing in scalable multi-user scenarios within serverless edge computing, emphasizing cost and latency optimization under resource constraints. We propose a scalable framework, SoCL, for efficient microservice provisioning, which optimally positions microservice instances to balance cost and latency based on user demands. Extensive experiments, including simulations and containerized deployments on Kubernetes, demonstrate that SoCL consistently outperforms baselines, handling large-scale user requests and network expansions effectively. Compared to the state-of-the-art ILP optimizer Gurobi, SoCL achieves significant reductions in cost and latency while delivering execution times up to one order of magnitude faster, with optimality gaps below 9.9%.

Although this work places limited emphasis on user browsing behavior and interests, future research will incorporate user behavior modeling and preference integration to support context-aware resource management. This advancement aims to address broader QoS challenges in edge networks and facilitate the development of more generalized and adaptive resource management solutions.

Appendix

A. Proof of Theorem 1

Proof: We establish the proof by demonstrating that a proactive candidate node $v_\eta^{(m_i)}$ with $\mathcal{H}(v_\eta^{(m_i)}) \leq 2$ cannot reduce the completion time of user requests. Specifically, consider the case where $\mathcal{H}(v_\eta^{(m_i)}) = 2$, meaning that $v_\eta^{(m_i)}$ has exactly

two neighbors. Each proactive candidate node must be directly connected to nodes hosting user requests for microservice m_i. This requirement assumes that its neighboring nodes, v_a and v_b, are edge servers serving user requests for m_i, with $b(l_{a,\eta})$ and $b(l_{b,\eta})$ denoting their respective communication capacities. Consequently, $v_\eta^{(m_i)}$ must lie on a path connecting the nodes v_a and v_b, which can be classified into the following two scenarios: (i). $v_\eta^{(m_i)}$ lies on the shortest path between v_a and v_b. In this case, users are distributed across v_a and v_b, generating data volumes r_a and r_b, respectively. Since the deployment cost $\kappa(m_i)$ is uniform across all edge nodes, the key factor is how placement impacts communication efficiency and latency. When deploying m_i at either v_a or v_b, the optimal location is the one with the higher data volume, i.e., $\max\{r_a, r_b\}$. Assuming $r_a < r_b$, placing m_i at v_b is preferable, as it minimizes transmission by serving the majority locally. In this case, data from v_b is forwarded to v_a, with total transmission time $r_b/b(l_{b,\eta}) + r_b/b(l_{a,\eta})$. Conversely, placing m_i at $v_\eta^{(m_i)}$ requires both r_a and r_b to be transmitted, yielding a total time of $r_a/b(l_{a,\eta})+r_b/b(l_{b,\eta})$, which is higher when $r_a > r_b$. Thus, $v_\eta^{(m_i)}$ is a less efficient option. When multiple instances are allowed, deploying directly at v_a and v_b eliminates inter-node transmission entirely, further improving efficiency. Therefore, in this scenario, $v_\eta^{(m_i)}$ does not offer latency advantages and should not be considered a candidate node. (ii). $v_\eta^{(m_i)}$ does not lie on the shortest path between v_a and v_b but instead on a longer alternative route. Any deployment on this path introduces additional communication delay, contrary to the goal of proactive node selection. Hence, such a node cannot enhance performance and is disqualified as a proactive candidate.In both scenarios, a node with $\mathcal{H}(v_\eta^{(m_i)}) \leq 2$ lacks sufficient connectivity to improve service latency. Only nodes with $\mathcal{H}(v_\eta^{(m_i)}) > 2$ can serve as effective proactive candidates, completing the proof of Theorem 1. ■

REFERENCES

[1] Deng, S., Xiang, Z., Taheri, J., Khoshkholghi, M. A., Yin, J., Zomaya, A. Y., & Dustdar, S. (2020). Optimal application deployment in resource constrained distributed edges. IEEE transactions on mobile computing, 20(5), 1907-1923.

[2] Fu, K., Zhang, W., Chen, Q., Zeng, D., & Guo, M. (2021). Adaptive resource efficient microservice deployment in cloud-edge continuum. IEEE Transactions on Parallel and Distributed Systems, 33(8).

[3] Lu, S., Yan, R., & Wu, J. (2023). Efficient Microservice Deployment with Dependencies in Multi-Access Edge Computing. In 2023 IEEE International Conference on Parallel and Distributed Systems (ICPADS) (pp. 2436-2443). IEEE.

[4] Yu, G., Chen, P., & Zheng, Z. (2020). Microscaler: Cost-effective scaling for microservice applications in the cloud with an online learning approach. IEEE Transactions on Cloud Computing, 10(2), 1100-1116.

[5] Tang, B., Guo, F., Cao, B., Tang, M., & Li, K. (2022). Cost-aware deployment of microservices for IoT applications in mobile edge computing environment. IEEE Transactions on Network and Service Management, 20(3), 3119-3134.

[6] He, X., Tu, Z., Wagner, M., Xu, X., & Wang, Z. (2022). Online deployment algorithms for microservice systems with complex dependencies. IEEE Transactions on Cloud Computing, 11(2), 1746-1763.

[7] He, X., Xu, H., Xu, X., Chen, Y., & Wang, Z. (2024). An Efficient Algorithm for Microservice Placement in Cloud-Edge Collaborative Computing Environment. IEEE Transactions on Services Computing.

[8] Tian, H., Xu, X., Lin, T., Cheng, Y., Qian, C., Ren, L., & Bilal, M. (2022). Dima: Distributed cooperative microservice caching for internet of things in edge computing by deep reinforcement learning. World Wide Web, 25(5), 1769-1792.

[9] Zhao, H., Deng, S., Liu, Z., Yin, J., & Dustdar, S. (2020). Distributed redundant placement for microservice-based applications at the edge. IEEE Transactions on Services Computing, 15(3), 1732-1745.

[10] Lv, W., Yang, P., Zheng, T., Lin, C., Wang, Z., Deng, M., & Wang, Q. (2023). Graph-Reinforcement-Learning-Based Dependency-Aware Microservice Deployment in Edge Computing. IEEE Internet of Things Journal, 11(1), 1604-1615.

[11] Peng, K., Wang, L., He, J., Cai, C., & Hu, M. (2024). Joint optimization of service deployment and request routing for microservices in mobile edge computing. IEEE Transactions on Services Computing.

[12] Liu, H., Zheng, W., Li, L., & Guo, M. (2022, July). Loadpart: Load-aware dynamic partition of deep neural networks for edge offloading. In 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS) (pp. 481-491). IEEE.

[13] Wang, L., Deng, X., Gui, J., Chen, X., & Wan, S. (2023). Microservice-oriented service placement for mobile edge computing in sustainable internet of vehicles. IEEE Transactions on Intelligent Transportation Systems, 24(9), 10012-10026.

[14] Ray, K., Banerjee, A., & Narendra, N. C. (2020, November). Proactive microservice placement and migration for mobile edge computing. In 2020 IEEE/ACM Symposium on Edge Computing (pp. 28-41). IEEE.

[15] Yang, K., Sun, P., Lin, J., Boukerche, A., & Song, L. (2022, July). A novel distributed task scheduling framework for supporting vehicular edge intelligence. In 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS) (pp. 972-982). IEEE.

[16] Jia, R., Chen, Z., Wu, C., Li, J., Guo, M., & Huang, H. (2024, September). RL-Cache: An Efficient Reinforcement Learning Based Cache Partitioning Approach for Multi-Tenant CDN Services. In 2024 IEEE International Conference on Cluster Computing (CLUSTER) (pp. 202-213). IEEE.

[17] Zeng, Y., Huang, Y., Liu, Z., & Liu, J. (2022, July). Distributed and Decentralized Edge Caching in 5G Networks Using Non-Volatile Memory Systems. In 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS) (pp. 425-435). IEEE.

[18] Chen, X., Bi, Y., Chen, X., Zhao, H., Cheng, N., Li, F., & Cheng, W. (2022). Dynamic service migration and request routing for microservice in multicell mobile-edge computing. IEEE Internet of Things Journal, 9(15), 13126-13143.

[19] Liu, D., He, C., Peng, X., Lin, F., Zhang, C., Gong, S., & Wu, Z. (2021, May). Microhecl: High-efficient root cause localization in large-scale microservice systems. In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) (pp. 338-347). IEEE.

[20] Gao, B., Zhou, Z., Liu, F., & Xu, F. (2019, April). Winning at the starting line: Joint network selection and service placement for mobile edge computing. In IEEE INFOCOM 2019-IEEE conference on computer communications (pp. 1459-1467). IEEE.

[21] Lu, S., Wu, J., Lu, P., Wang, N., Liu, H., & Fang, J. (2023). QoS-Aware Online Service Provisioning and Updating in Cost-Efficient Multi-Tenant Mobile Edge Computing. IEEE Transactions on Services Computing.

[22] Zhang, D., Wang, W., Zhang, J., Zhang, T., Du, J., & Yang, C. (2023). Novel edge caching approach based on multi-agent deep reinforcement learning for internet of vehicles. IEEE Transactions on Intelligent Transportation Systems, 24(8), 8324-8338.

[23] Imranur, M., Panichella, S., & Taibi, D. (2019). A curated dataset of microservices-based systems. Joint Proceedings of the Summer School on Software Maintenance and Evolution (CEUR-WS), Tampere, Finland.

[24] Li, H., Liu, H., Chen, A., Ma, X., Liu, Q., & Du, J. (2024, August). RIA: Return on Investment Auto-scaler for Serverless Edge Functions. In Proceedings of the 53rd International Conference on Parallel Processing (pp. 772-781).

[25] Luo, S., Xu, H., Ye, K., Xu, G., Zhang, L., Yang, G., & Xu, C. (2022, November). The power of prediction: microservice auto scaling via workload learning. In Proceedings of the 13th Symposium on Cloud Computing (pp. 355-369).

[26] Wang, K., Li, Y., Wang, C., Jia, T., Chow, K., Wen, Y.,& Zhang, L. (2022, August). Characterizing job microarchitectural profiles at scale: Dataset and analysis. In Proceedings of the 51st International Conference on Parallel Processing (pp. 1-11).