POLITECNICO DI TORINO
Repository ISTITUZIONALE

Efficient Management of Composite Heterogeneous Applications at the Network Edge

Original
Efficient Management of Composite Heterogeneous Applications at the Network Edge / Adeppady, M., Yu, Y., Rahmanian, A., Ali-Eldin Hassan, A., Chiasserini, C.F.. - In: IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT. - ISSN 1932-4537. - 23:(2026), pp. 6550-6568. [10.1109/TNSM.2026.3709656]

Availability:
This version is available at: 11583/3012701 since: 2026-07-04T09:29:22Z

Publisher:
IEEE

Published
DOI:10.1109/TNSM.2026.3709656

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2026 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

09 November 2026

Efficient Management of Composite Heterogeneous Applications at the Network Edge

Madhura Adeppady, *Member, IEEE*, Yenchia Yu, *Student Member, IEEE*, Ali Rahmanian, *Non-Member, IEEE*, Ahmed Ali-Eldin Hassan, *Non-Member, IEEE* Carla Fabiana Chiasserini, *Fellow, IEEE*

***Abstract*—Edge computing is a promising paradigm for deploying latency-sensitive applications (Apps) as it brings resources closer to end users. Edge Apps often adopt a microservice (MS) architecture, breaking monolithic Apps into lightweight, containerized MSs that can be dynamically and independently deployed. However, managing such Apps involves three key challenges: (i) optimizing the placement of MSs to reduce both response time and resource overhead, (ii) handling MS migration or relocation as users move while minimizing App service disruption (App downtime), and (iii) enabling MS sharing across Apps while ensuring performance guarantees. We formulate this as an optimization problem, named Multi-microservice Application Placement (MAP), prove its NP-hardness, and introduce STEP (State and Topology-aware Edge-MS Placement), a polynomial-time heuristic. STEP distinguishes itself from prior work by: (i) jointly considering stateful and stateless MS characteristics in deployment decisions, (ii) exploiting MS shareability to reduce resource usage, (iii) balancing response latency, App downtime, and resource utilization, and (iv) leveraging multiple versions of the same MS to adapt quality of service to available edge resources. Our results in a small-scale scenario show that STEP achieves near-optimal performance with only 7% higher CPU cost than the optimal solution. Large-scale real-time experiments on a Kubernetes cluster demonstrate that STEP consistently outperforms competing methods, achieving up to 50% lower deployment costs while delivering 50% gain in app quality and saving 15% in radio resources with over 90% request success rates.**

***Index Terms*—Mobile edge computing, Stateless and stateful microservices, Application deployment and migration, Service management**

I. Introduction

Edge computing has emerged as an alternative to cloud computing for deploying latency-sensitive applications (Apps) by bringing computing resources closer to end users. In edge computing, servers are deployed near base stations (BSs), enabling computation offloading from mobile devices while significantly reducing response latency and bandwidth consumption. Increasingly, edge Apps adopt a microservice (MS) architecture, where monolithic Apps are composed of multiple lightweight, independently deployable containerized MSs, making them well-suited for dynamic edge environments.

MSs at the edge can be categorized into stateful and stateless based on their operational characteristics [1]. Stateful MSs maintain user session and interaction data across requests, preserving state over time. Conversely, stateless MSs process each request independently without storing user-specific data. A key feature of MS architecture is shareability: MSs can be shared across multiple Apps to improve resource efficiency. However, while stateless MSs are shareable across Apps requested by different users, stateful MSs are typically shareable only among Apps requested by the same user. For different users, stateful MS shareability depends on whether or not their state includes user-specific information. For example, a registry MS that stores registration data of an unmanned aerial vehicle (UAV) can be shared across different UAVs and their requested Apps, as it maintains generic user information. In contrast, an autopilot MS that manages flight parameters for individual UAVs cannot be shared, as the state information is unique to each user's mission. Additionally, to enhance flexibility in resource-scarce environments (common in edge servers), an MS can have multiple versions, each offering a different Quality of Service (QoS) level (e.g., accuracy in a classification task) as well as a different level of complexity and amount of resources required for its execution.

Despite its benefits, an MS architecture at the edge introduces relevant challenges in providing performance guarantees with minimal resource consumption.

First, since Apps are composed of several MSs, it is extremely difficult to manage resources at the individual MS level so as to minimize the deployment cost while ensuring tolerable response latency. The network orchestrator must allocate appropriate computational speed to each MS, manage Resource Blocks (RBs) on the communication link between users and servers, and determine MS placement that minimizes the communication overhead between MSs and between a user and the entry MS of a requested App.

Second, as users move between BSs coverage areas and connect to different servers, the deployment topology of MSs must be updated to honor response latency requirements. This triggers the migration of stateful MSs and the relocation of stateless MSs, both causing service disruption (App downtime). For stateful MSs, the entire container along with user session data must be migrated to maintain state continuity, while for stateless MSs, container relocation is enough since they do not maintain any user state. Thus, the orchestrator must additionally ensure that App downtime during migration remains within acceptable limits.

Third, MS shareability further complicates the orchestrator's decisions since shared MSs impact the performance of multiple Apps across different users. Notably, the allocation of computational resources may need to be adjusted dynamically, as new Apps start using an existing MS or Apps using a shared MS terminate.

This work was funded through the SNS JU 6G-INTENSE project (Grant No. 101139266).

These challenges motivate the need for a holistic MS management framework that can optimize deployment topology while balancing the trade-offs across response latency, App downtime, and resource utilization.

Limitation of state-of-the-art approaches. A large body of recent approaches address resource management to reduce App response latency. Notably, [2], [3] focus on analyzing individual MS contributions to response latency and apply autoscaling, or determine MS-to-server allocation [4], [5] to reduce costs while meeting target end-to-end response latency. However, these approaches have limitations when applied to edge environments. First, they do not consider the impact of MS migration/relocation triggered by user mobility. Second, they lack the flexibility of adapting MS versions based on resource availability, which is crucial for efficiently managing limited resources at the edge. Third, not all of them exploit MS shareability as a way to reduce the consumption of computational, hence also energy, resources. Other prior approaches targeting edge environments focus on MS pre-deployment based on predicted user mobility [6], [7]. These works consider Apps as composed of only stateful MSs and use abstract migration models that do not capture the complexity of real MS migrations. Moreover, these approaches simply skip MS migration when destination servers lack sufficient resources.

Recent approaches have made further progress: SR-CL [8] jointly optimizes migration and resource allocation but treats Apps as monolithic single-MS services; CMMF [9] addresses DAG-based migration but overlooks stateless MSs and shareability; E2MS [10] minimizes service disruption and communication costs during migration but assumes non-shareable MSs with a single fixed quality version; and Edge-Mi [11] addresses task migration between edge and cloud based on mobility patterns but does not consider MS shareability or quality adaptation. Works such as [12] provide realistic migration models but focus only on migration rather than jointly optimizing placement and resource allocation. Moreover, none of these approaches simultaneously handle stateless and stateful MS coexistence, MS shareability across users, and dynamic quality adaptation under edge resource constraints.

Key insights and contributions. We propose State and Topology-aware Edge-MS Placement (STEP), an algorithm for the orchestrator to make deployment and resource allocation decisions that minimize deployment costs while ensuring that response latency and App downtime remain within tolerable limits. STEP first constructs a Dynamic Network Topology Graph (DNTG) that represents the key system components and their interactions. This graph captures possible deployment choices for MSs of the Apps requested by users. Additionally, it identifies dynamic conditions at the edge, such as App deployment requests, user migrations, and App terminations. Then, upon any of the above events, the orchestrator builds a decision graph that encodes feasible deployment and resource allocation choices to meet App requirements while minimizing deployment costs. This graph is expanded to enforce additive constraints and ensure decision feasibility. Finally, deployment decisions are derived by identifying the minimum-cost path in the decision graph.

We summarize our key contributions as follows:

- We develop a system model that captures all key aspects of edge servers, users, Apps, and MSs composing these Apps, and derive analytical expressions for the performance metrics of interest. Further, we develop a testbed that allows us to perform a solid experimental analysis motivating our work and design choices, and then validate and thoroughly evaluate our proposed solution.
- We formulate the Multi-microservice Application Placement (MAP) problem, which aims to minimize deployment cost while fulfilling all performance requirements. We prove that MAP is NP-hard and propose the STEP algorithm, which has polynomial time complexity. Importantly, STEP differentiates from prior art in the following ways: *(i)* It considers Apps composed of both stateful and stateless MSs, which determines MS shareability and App downtime; *(ii)* It addresses multiple performance metrics, such as response latency, App downtime, and resource consumption, by leveraging an expanded graph that ensures these objectives are met; *(iii)* Its low complexity design makes it suitable to handle dynamic scenarios where demand for Apps and user mobility vary over time; *(iv)* It leverages MS shareability by reusing already deployed MS instances for newly required Apps or during App migration; *(v)* It increases deployment flexibility by considering different versions of the same App with individual MSs that may offer different QoS levels and, correspondingly, different complexity and resource consumption.
- Finally, in a small-scale scenario, we show that STEP closely matches the optimal solution. In a large-scale Kubernetes cluster deployment, STEP significantly outperforms existing methods with up to 50% cost reduction, superior resource efficiency, and 50% increasement of the app quality compared to baseline approaches.

We mention that a preliminary version of our work has appeared in our conference paper [13] where we sketched a first version of the STEP algorithm using a simplified migration model and presented a limited, initial set of results.

Paper organization. Sec. III describes our target scenario and testbed setup. Sec. IV gives experimental evidences on why we need a holistic deployment algorithm for multi-MS App placement, and Sec. V introduces our system model. Sec. VI introduces the problem formulation and its complexity analysis. Sec. VII-A presents the proposed low-complexity orchestration algorithm STEP, while Sec. VIII highlights the improvement of STEP against state-of-the-art approaches. Finally, Sec. II summarizes related work and highlights our key contributions, while Sec. IX concludes the paper.

II. Related Work

Our study pertains to three main areas: edge-centric MS deployment, MS scaling, and MS migration.

Edge-centric MS deployment. Multi-component App placement is addressed, e.g., in [4], which accounts for both user mobility and network dynamics. It first determines MS deployment by matching the physical (edge servers) and App graphs using the Hungarian algorithm, ignoring communication costs; then a local search refines the solution by incorporating these costs. Similarly, [5] proposes a greedy algorithm

that first assigns an app to the server with the lowest resource-based assignment cost, neglecting communication costs, and then updates the assignment costs for the remaining apps by incorporating communication costs with the one already placed one. The learning-based approach in [6] proactively deploys the app MSs on the nearest edge server, considering the dependencies of the MS and user mobility. It also determines how many successor MSs to predeploy and selects the edge servers for their placement. This approach is extended in [7] to support complex MS structures by considering branching, parallel paths, and cycles in pre-deployment decisions. MS migration is addressed in [10], which selects candidate App MSs for migration by accounting for communication, image pull, and MS downtime costs, to minimize service disruption. In [14], a policy gradient-based reinforcement learning algorithm is proposed to make migration decisions considering initialization, computation, and migration delays.

SR-CL [8] addresses the joint problem of determining when and where to migrate service instances and the optimal allocation of computational resources to these service instances. For migration decisions, a deep reinforcement learning approach is used, while convex optimization handles optimal resource allocation. CMMF [9] proposes a framework for DAG-based service migration in mobile edge computing. The framework uses a two-stage approach: deep reinforcement learning with LSTM networks for edge cloud selection, followed by network flow algorithms with topological sorting for MS placement. Adapt-SD [15] optimizes MS deployment by minimizing computational, storage, and network costs while meeting service access time constraints, but focuses solely on initial placement decisions without considering dynamic migration scenarios. The study in [16] proposes a reinforcement learning-based approach to determine the optimal sequence of edge clouds to serve mobile users' microservice requests, formulating the problem as a Markov decision process that balances service delay and migration costs.

Edge-Mi [11] presents an MS-based task migration framework that allows migration between edge servers and cloud infrastructure based on mobility patterns and resource requirements. The approach decomposes Apps into MSs that can be selectively migrated from edge servers to cloud resources (or vice versa) when mobile devices move out of coverage areas or when edge resources become insufficient. Nautilus [17] provides a three-tier runtime system with communication-aware mapping using graph algorithms, contention-aware resource management through deep reinforcement learning, and load-aware migration that migrates MSs from overloaded to underutilized nodes based on minimizing additional network communication costs. A closed-loop particle swarm optimization approach is introduced in [18] for service migration. This approach considers QoS and application characteristics to trigger service migration. The study in [19] presents a location-aware intelligent scaling framework for composite Apps in mobile edge computing environments. The approach considers user mobility patterns and dynamically adjusts MS scaling decisions based on location information to optimize resource utilization and service performance.

Recent works have further explored learning-based and predictive approaches for task migration and resource allocation at the edge. DDPG [20] proposes a deep deterministic policy gradient algorithm enhanced with a dual experience pool, to jointly optimize task migration and resource allocation in cooperative edge computing environments. A proactive service migration framework MAPSM [21] is proposed for Internet of Vehicles (IV) that combines RNN and Markov chain-based mobility prediction with coordinated container pre-migration and communication handover to minimize end-to-end delay and total migration time. MATM [22] integrates a Mamba-based trajectory prediction module with a Soft Actor-Critic DRL decision-making module to enable proactive task migration and resource allocation in vehicular edge computing. However, all of the three above approaches consider monolithic tasks or single containerized services, and do not address Apps composed of multiple MSs.

Unlike STEP, the above approaches do not account for the coexistence of stateless and stateful MSs in the App. Moreover, they overlook scenarios where migration is infeasible due to resource-constrained servers. These approaches either optimize migration decisions or resource allocation dynamically, but not both jointly, and fail to leverage MS shareability across different users or applications to reduce resource consumption. In contrast, STEP offers the flexibility to reduce the quality version of MSs when needed and provides a holistic solution that simultaneously addresses deployment, migration, and resource allocation decisions.

MS scaling. App response latency guarantees are provided in Erms [2], where a shared MS environment is considered and the latency target of each MS of the Apps is given by a piecewise linear function of its workload. Further, if an MS is shared among multiple Apps, Erms gives higher priority to Apps with more latency-sensitive MSs to optimize request execution. PEMA [3] proposes an iterative feedback-based approach that initially allocates abundant resources to all MSs to meet App SLA requirements, and then it iteratively selects the best resource configuration for each MS by exploiting monotonic resource reduction opportunities. Although these approaches work well for the cloud, they fail to address the limited resource availability at the edge and the need for migration, which introduces an additional performance metric for the App downtime. DeepScaling [23] uses a three-component system with spatio-temporal graph neural networks for workload forecasting, deep neural networks for CPU utilization estimation, and an improved Deep Q-Network for adaptive autoscaling policy generation to minimize resource costs while meeting SLAs in large-scale production microservice environments.

All the above MS scaling schemes work well for resource allocation in cloud environments but differ from STEP in three key ways. First, they assume abundant resources and focus on meeting SLA requirements through resource optimization, while STEP operates under strict edge resource constraints and adjusts MS quality versions accordingly. Second, they do not handle MS migration, which introduces App downtime as a critical performance metric that STEP addresses through joint optimization of deployment, migration, and resource allocation decisions. Third, while some consider MS sharing only for prioritizing latency-sensitive applications, STEP uses MS

TABLE I
COMPARISON OF STEP WITH PROMINENT RELATED APPROACHES

Approach	Stateful + Stateless MS	MS Shareability	Multi-version QoS	User Mobility	Joint Placement + Migration	Resource Allocation	Learning-based
SR-CL [8]	✗	✗	✗	✓	✓	✓	✓
CMMF [9]	✗	✗	✗	✓	✓	✗	✓
E2MS [10]	✗	✗	✗	✓	✓	✗	✗
Edge-Mi [11]	✗	✗	✗	✓	✗	✗	✗
DDPG. [20]	✗	✗	✗	✗	✓	✓	✓
MAPSM [21]	✗	✗	✗	✓	✓	✗	✓
MATM [22]	✗	✗	✗	✓	✓	✓	✓
STEP (ours)	✓	✓	✓	✓	✓	✓	✗

shareability across users and applications as a core mechanism to reduce resource consumption at the edge.

MS migration. In the area of container migration, existing studies focus on either stateful migration [24], [25], [26], [27], [12] or stateless relocation [28], [29], [30] of MSs. Although relevant, these approaches are orthogonal to our work, as they focus on *how* to migrate MS containers rather than *where* to migrate them. We stress that any of these migration models can be integrated into our approach.

Finally, in our conference paper [13], we proposed a preliminary version of the STEP algorithm for managing composite edge applications using a simplified migration model. Building upon this foundation, in this work we propose an enhanced orchestration framework that includes a more realistic parallel migration model as well as an improved and lower-complexity version of STEP, with enhanced decision graph construction and pruning steps. Further, we significantly extended our Kubernetes-based testbed and the related experimental analysis, and we provide a formal problem formulation with NP-hardness proof, and an extensive experimental evaluation of STEP against a richer set of state-of-the-art benchmarks in both small-scale and large-scale scenarios.

Progress beyond state-of-the-art. As highlighted in Table I, some of the above approaches either optimize migration decisions or resource allocation dynamically, but not both jointly. Moreover, they treat applications as monolithic services rather than chains of heterogeneous MSs, and thus do not account for the coexistence of stateful and stateless MSs in the App and their distinct migration behaviors. They also overlook scenarios where migration is infeasible due to resource-constrained servers, and fail to leverage MS shareability across different users or applications to reduce resource consumption. Furthermore, learning-based approaches, despite their ability to make adaptive decisions in dynamic environments, are limited to monolithic services and do not extend to composite multi-MS applications. Finally, none of the above approaches consider multiple versions of the same MS to adapt application quality to the available edge resources, which is crucial for efficiently managing limited resources at the edge. In contrast, STEP addresses all these limitations by providing a holistic solution that simultaneously handles deployment, migration, and resource allocation decisions for composite Apps composed of heterogeneous stateful and stateless MSs.

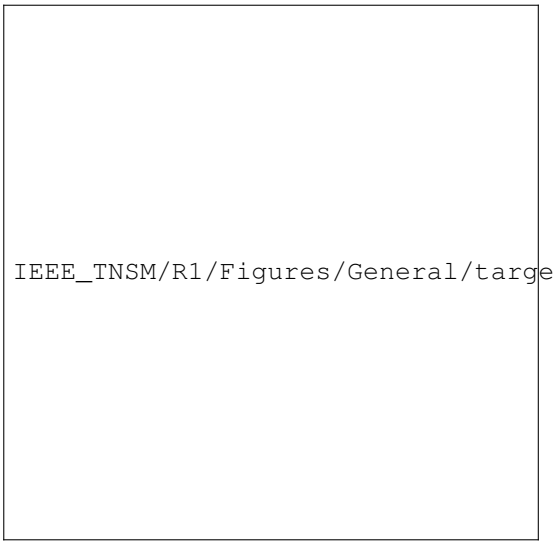

Fig. 1. Exemplary target scenario: Apps are deployed at the edge as chains of stateful / stateless MSs. Mobile users (i.e., UAVs) access the Apps through cellular BSs, with User_1 (User_2) requesting App_1–App_2 (App_3–App_4).

III. TARGET SCENARIO AND TESTBED

Our study focuses on managing MS-based applications in an edge computing system for mobile users. In the following, first we introduce the system we focus on. Then we describe the testbed that we developed to conduct our quantitative analysis of the trade-offs that exist in the system under study and serve as empirical foundation of our work. We use the testbed also to implement, test, and benchmark our solution.

Reference scenario. Fig. 1 depicts our reference scenario, including an edge computing system with an orchestrator and multiple edge servers. The latter are interconnected with each other physically or logically (i.e., through virtual links each composed of multiple physical links) and managed by the orchestrator. Edge servers can host Apps, each composed of (possibly) multiple containerized MSs, offered to mobile users.

Each server may be connected with one or more BSs, which enable the mobile users to access the Apps deployed at the edge. Each mobile user can issue requests for one or multiple Apps. During runtime, mobile users may move and change the BS with which they connect. For example, User_2 may handover its connection from the BS connected with edge server 2 to the BS connected with edge server 1. When the MS deployment remains static, the requests from User_2 to App_3 and App_4 would need to be routed from server 1 to server 2, thus increasing latency and bandwidth consumption. The

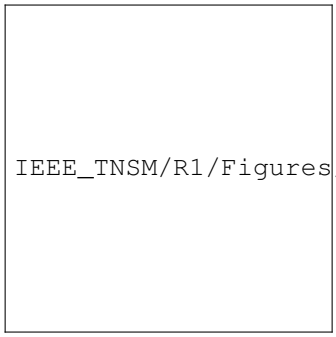

Fig. 2. Open radio cell location (left) and edge server topology (right) in the city of Turin, Italy.

orchestrator can thus make the MSs migrate/relocate to keep the Apps latency sufficiently low. Similarly, when the load on an App varies, the resources allocated to its MS instances may be insufficient or excessive, and the orchestrator may need to migrate/relocate MSs to optimize resource allocation.

Testbed. We have recreated the above system architecture in our testbed for the development and testing of our solution. Notably, we have used the approach in [31] to design edge server topologies and the open radio cell information available for the geographical area we consider (namely, an area in the city of Turin, Italy) in `OpenCelliD` (https://www.opencellid.org/) (Fig. 2 (left)). We clustered the radio cells into 4 clusters using the K-means algorithm and deployed one edge server at the geographical center of each cluster (Fig. 2 (right)). To limit the number of physical links, we consider 1 Gbps optical links only between edge servers that are within a 3 km distance. Also, to ensure enough bandwidth for multi-hop communication between edge servers through logical links, we slice the bandwidth of inter-server physical links so that 60% is used for direct communication and the rest for multi-hop communication. The radio cells within a cluster are viewed as a virtual BS that is co-located with the edge server, and the communication bandwidth between the mobile user and the BS is estimated based on the 3GPP TS 38.306 standard [32] with the channel quality indicator (CQI) being negatively correlated with the user-BS distance.

We have implemented the emulation testbed of this edge computing system using Kubernetes (https://kubernetes.io/), as depicted in Fig. 3. We use Kind (https://kind.sigs.k8s.io/) to create an edge cluster of 8 edge nodes on a host server, with each node featuring 2 CPU cores at 2.8GHz and 4GB of memory. The communication bandwidth between edge nodes is set through the `tc` Linux tool (https://man7.org/linux/man-pages/man8/tc.8.html). We use the host server as the edge orchestrator node, and deploy four main components of our testbed: *(i)* User simulator, which simulates the mobile users movement and estimates the communication bandwidth to the connected BS; *(ii)* System orchestrator, which runs the real-time edge orchestration algorithm and updates the MS deployments via native Kubernetes API; *(iii)* System performance monitor, which monitors the resource consumption on each edge nodes using Prometheus (https://prometheus.io/) and visualize the measurements via Grafana (https://grafana.com/); and *(iv)* Mongo DB, which is a database that stores the user measurements and logs for analysis. On each edge node, we deploy a request generator, which produces application requests on behalf of users connected to the corresponding BS, informed by the User Simulator. The measurements of each request, including application quality and response latency, are persisted in MongoDB. Finally, we leverage the customized HydraGen tool [33] to generate emulated applications with MS quality selectors, enabling comprehensive validation of the dynamic edge orchestration strategies.

Fig. 3. Edge system emulation testbed.

IV. Experimental Analysis

We now use our testbed to empirically analyze the trade-offs that exist in the system and that are affected by orchestration decisions on MS deployment. The considered trade-offs address two main aspects: *(i)* the mobile user quality of experience (QoE), represented as average App response latency, App QoS, and user request success rate, and *(ii)* the efficiency of the App deployment solution, i.e., edge server resource consumption, edge server load balancing, and the edge system capability to serve multiple users.

To ease the analysis, we consider here a base scenario[1] with one mobile user, namely, a UAV, requesting an App and moving at the constant speed of 10 m/s along a 5 km-long straight trajectory. Along the trajectory, 3 BSs (and corresponding edge servers) are placed at equal distance. The SNR varies depending on the user distance from the BS to which it is connected and varies in the $[4, 24]$ dB range, which corresponds to a CQI varying between $[5, 15]$. Also, the edge server colocated with the first BS has 5 CPU cores available, while the other two have just 2 CPU cores each. The communication bandwidth between edge servers is set to 1 Gbps, except for the link between the edge server with 5 CPU cores and one of the edge servers with 2 CPU cores, which is set to 100 Mbps to account for a varying capacity over the network. When the

[1]We will use our testbed to represent more complex scenarios while evaluating our solution in the later sections.

mobile user connects to a BS, by default the BS allocates 6 RBs for the communication with the user. The mobile user issues 1 request/s for an App composed of 3 MSs. Each request has the size of 5 MB, and the inter-MS communication payload size is 5 MB. Every MS can be deployed in one of three possible versions, with each version corresponding to a different quality level, namely, low, medium, and high, consuming 0.25, 0.5, and 0.75 10^9 CPU cycles (resp.), to process one user request. Unless otherwise specified, we allocate 1 CPU core for each MS when it is deployed on the edge system. Finally, we set the target App response latency to 1 s. For simplicity, the migration/relocation time and the amount of data to transfer for each MS migration/relocation is considered to be constant across the different instances of that MS, and multiple MSs can be migrated/relocated in parallel, thus the overall App downtime equals the duration of the slowest MS migration/relocation. For each considered baseline, we run a system emulation lasting 30 minutes and resulting in 12 handovers.

At every user handover, the orchestrator reassesses the current Apps deployment and makes changes according to the following three MS deployment schemes (hereinafter also referred to as baselines), representative of the main strategies that have been proposed in the existing literature [34]:

- *App-level closest deployment (ALC).* It deploys MSs belonging to the same App on the same edge server, selecting the closest server to the mobile user that requests the App.
- *MS-level closest deployment (MLC).* It deploys each MS of an App on the edge server that is closest to the mobile user among those with room to host the MS. For low computing load, the deployment coincides with of the App-level closest strategy.
- *MS-level closest-least-loaded deployment (MCLL).* It deploys each MS of an App on the least loaded server; in the case of ties, it selects the one closest to the mobile user.

Next, we use the three strategies above to investigate the fundamental trade-offs that exist in the system under study and establish the motivation of our work.

1. App response latency vs. RB usage. Fig. 4a presents the average App response latency as a function of the number of RBs allocated to the mobile user, with the violin plot highlighting the distribution of the App latency measurements. As expected, under all three deployment strategies, the average response latency decreases significantly, up to 40%, as the RB allocation to the UAV grows, i.e., the user's data rate increases. Among the three baseline strategies, MLC always performs best. Specifically, the latency target (horizontal black line) for ALC, MLC, and MCLL strategies are satisfied when allocating at least 5 RBs, 4 RBs, and 7 RBs, respectively. However, the latency distributions (violin plots) reveal considerable variability: even when the average latency meets the target, some responses (ALC: 33%, MLC: 23%, MCLL: 45%) still exceed it. *This highlights the critical impact of variable bandwidth due to the user mobility and channel conditions, which must be managed carefully for latency-sensitive applications*.

2. App response latency vs. CPU usage. Fig. 4b illustrates the average App response latency and the corresponding distribution versus the per-MS CPU allocation. Similarly, Fig. 4c depicts the average CPU usage in the edge system as a function of the allocated per-MS CPU. As expected, increasing the per-MS CPU allocation reduces by up to 40% the App response latency under all strategies. Notice that such an increased CPU allocation leads to 15–55% higher CPU usage, depending on the deployment strategy. Also, due to the imbalanced edge server capacity and the different MS placement strategies of the baselines, the increase of the average CPU usage is not always linear (Fig. 4c), and the per-MS CPU allocation significantly influences a strategy performance (Fig. 4b). When allocating fewer than 1.2 CPU cores for each MS, MLC yields the lowest latency. From 1.2 to 1.6 CPU cores, instead, ALC performs best. However, beyond 1.6 cores, the ALC strategy becomes unable to yield a feasible deployment, as no single edge server can accommodate all the MSs, and MLC becomes the best strategy again. Furthermore, to ensure that the average App response latency meets the target, at least 0.8 CPU cores/MS are required when MLC or ALC are adopted, and 1 CPU core/MS is required when MCLL is used. The violin plot in Fig. 4b underscores that, as the CPU allocation grows, the response latency rarely exceeds the target, specifically, with probability 16% and 18% for MLC and MCLL, respectively, when 2 cores per MS are allocated, and with probability 16% for ALC in the case of 1.6 per-MS core allocation. Importantly, this is because *the faster the MS processing, the larger the communication latency margin to mitigate the effects of bandwidth variation due to the high user mobility*.

TABLE II
AVERAGE STANDARD DEVIATION OF EDGE SERVERS' LOAD

Per-MS allocated CPU cores	ALC	MLC	MCLL
0.6	44.03%	43.91%	14.60%
1	36.17%	46.33%	20.61%
1.4	46.04%	32.04%	24.23%
1.8	–	37.50%	31.13%

3. CPU usage vs. Load balancing. Next, we look at the trade-off between average CPU usage and load balance across edge servers. Table II presents the average standard deviation of CPU usage, calculated as the average of the standard deviation of the edge servers' load at each measurement time instance. Fig. 4c highlights that ALC and MCLL achieve, respectively, the lowest and the highest average CPU usage. Conversely, the results in Table II underline that MCLL consistently gives the lowest average standard deviation, i.e., the most balanced load distribution across the edge servers. *This highlights that there is a clear trade-off between overall CPU utilization and load balance*.

4. MS quality vs. Request success rate. For critical applications, it is crucial to ensure an App request success rate, which is computed here as the percentage of requests that can be served honoring the 1 s-target response latency over the total number of requests. Fig. 5 shows the request success rate (left y-axis) and per-MS per-request CPU cycle requirement (right y-axis) as the MS quality varies. The different plots depict the results for different combinations of BS-user RB allocation and per-MS CPU allocation. In all scenarios, as a higher MS quality level is selected, the per-MS per-request

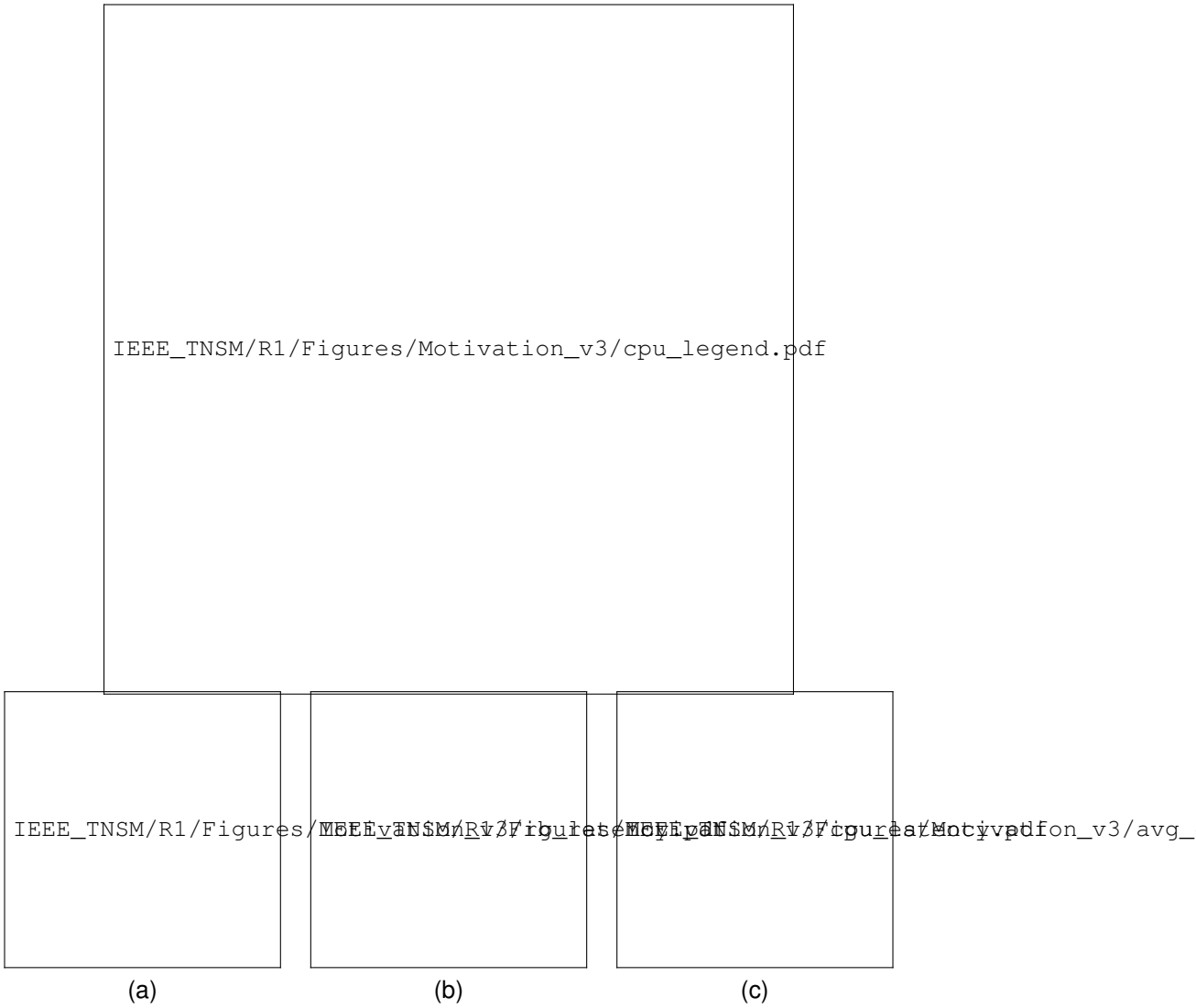

Fig. 4. Average App response latency for a varying number of allocated RBs (a); Average App response latency (b) and Average edge system CPU usage (c) for a varying per-MS CPU core allocation. The performance is shown under the three baseline orchestration algorithms, namely, ALC, MLC, and MCLL.

CPU cycle demand grows, resulting in longer processing times and, hence, decreasing success rate. Also, the comparison of Fig. 5a with Fig. 5c highlights a 5%–10% improvement in request success rate across all MS quality levels and baselines, indicating a consistent benefit from increased RB allocation. *These results underscore the importance of being able to adapt the MSs (hence, the App) quality level to the network and computing resource availability . Also, the trade-off between success rate and resource consumption depends substantially in the deployment strategy.* In more detail, looking at Fig. 5a and Fig. 5b, one can observe up to 20% increase in the request success rate, especially when medium or high quality MSs are selected under the ALC and MCLL strategies. Further, comparing Fig. 5a and Fig. 5d, we observe that the change of CPU and RB allocations impact the performance significantly in the case of medium and high quality MSs, but have a minor impact for the low quality MSs.

5. MS sharing and quality levels vs. Users' success rate. Finally, we explore the trade-off between the level of MS sharing, the level of MS quality, and the ability to successfully serve users' requests. For consistency, we consider the same settings for all mobile users, except for their initial position as they are uniformly distributed along the service area. Fig. 6a and Fig. 6b show the average request success rate when medium quality MSs are selected, and, respectively, none of the MSs and all common MSs, are shared among the users. Further, Fig. 6c and Fig. 6d present the average request success rate when, respectively, low and high quality MSs are selected and common MSs are shared among the users. In all the considered settings, the results indicate that the MLC and MCLL strategies can consistently support a higher number of users compared to ALC. When comparing Fig. 6a and Fig. 6b, one can notice that MS sharing substantially increases the number of users the edge can accommodate – up to 3 additional users for ALC and up to 5 for MLC and MCLL. However, for more than 3 users, the average request success rate slightly decreases due to the increased queuing time of the requests at the shared MS, which leads to increasingly larger App response latency.

Importantly, there is an additional degree of freedom that

(a) 1 CPU, 6 RBs (b) 1.5 CPU, 6 RBs (c) 1 CPU, 9 RBs (d) 1.5 CPU, 9 RBs

Fig. 5. User request success rate and per-MS per-request CPU demand as functions of selected MS quality, under different per-MS CPU allocation and per-User RB allocation CPU and RB allocations for the three baseline algorithms.

(a) Medium quality, not shared (b) Medium quality, shared (c) Low quality, shared (d) High quality, shared

Fig. 6. Average per-user request success rate when different quality of MSs are used and the same MSs are shared/not shared among the users. The performance is shown under the three baseline algorithms.

can be combined with shareability. Looking at Fig. 6b and Fig. 6c, we observe that selecting low-quality MSs further expands the edge's ability to accommodate users (up to 7) under the MLC and MCLL strategies while still ensuring high success rates. Instead, the comparison between Fig. 6b and Fig. 6d underlines that selecting high-quality MSs significantly reduces the request success rates, even with few UAVs. *Thus, sharing MSs along with a sensible selection of MS quality (possibly favoring lower quality) substantially increases the edge ability to successfully serve users' requests.*

Summary and major findings. Our experimental analysis clearly shows that efficiently orchestrating MS deployment or relocation at the edge is a complex, multi-faceted challenge—one that cannot be solved with simplistic or static strategies. This complexity arises from the interplay of multiple factors, including computing resource allocation, MS quality selection, bandwidth provisioning, MS sharing policies, and load balancing mechanisms. These factors jointly determine the system's ability to support mobile applications both reliably and efficiently.

Notably, our results reveal that:

(i) Resource–QoE alignment is essential: bandwidth and computing resources must be dynamically allocated in accordance with user QoE requirements, particularly under fluctuating channel conditions.

(ii) Distributed MS placement boosts scalability: deploying the MSs that compose an application across different edge servers can substantially increase the number of users served without compromising QoE.

(iii) Adaptive sharing and quality adjustment enhance resilience: system performance can be further improved by sharing common MSs across multiple applications and, when resources are constrained, switching to lower-quality App versions—provided that QoE remains acceptable.

(iv) No one-size-fits-all strategy exists: the optimal approach depends on the prevailing system constraints, operational conditions, and specific application requirements.

Furthermore, our findings underscore that performance is

highly sensitive to the real-time state of the edge system, making static management schemes inherently limited. It is therefore crucial to adopt flexible, adaptive orchestration solutions that can respond to system dynamics in real time, managing the inherent trade-offs between application performance and operational costs.

V. System Model and Time Performance Metrics

This section outlines the key assumptions made to build a mathematically tractable, yet accurate, model. Then it gives analytical expressions for the two performance metrics, i.e., App response latency and App downtime.

A. Assumptions

To build a model that captures all relevant system aspects while retaining tractability without loss of generality, we make the following assumptions:

• *Edge server and base stations:* Each BS is co-located with an edge server, thus we refer to the user-edge server link as the wireless link between the user and the corresponding BS.

• *MS deployment:* An App entry MS is placed on the edge server co-located with the BS serving the requesting user.

• *MS migration/relocation:* The orchestrator may decide to change the MS deployment topology upon a user's handover, App request arrival, or App termination; such deployment remains unchanged until the next occurrence of these events.

• *MS version:* MS instances implementing different versions of the MS (i.e., providing different levels of QoS) have the same communication interfaces. Further, notice that, at the beginning of a migration procedure, stateless MSs can switch to different versions according to the available resources at the destination server. Conversely, stateful MSs *cannot* change version during migration, due to their need to retain user-specific state information.

• *MS shareability:* Recall that stateless MSs can be shared among Apps requested by different users. Conversely, stateful MSs can always be shared among the Apps requested by the same user but may or may not be shared between different users, depending on whether their state contains user-specific information.

• *App termination:* The App termination events are triggered when an App completes all its assigned tasks, or when operational conditions require termination. For example, when a UAV completes its assigned tasks or returns to its base, it triggers an App termination event at the orchestrator.

B. System Model and Time Performance Metrics

Consider an edge system consisting of a set $\mathcal{S}$ of servers available to serve users. Every server is equipped with computational capacity (memory M_s, CPU C_s) and co-located with a BS that enables connectivity with mobile users. These edge servers are interconnected through a multi-hop wired network infrastructure.

Each mobile user u, $u{\in}\mathcal{U}$, generates requests for Apps, whose set is denoted by $\mathcal{A}^u$, that need to be deployed on these servers. As they move, users may connect with different BSs using an allocated bandwidth $B_{u,s}$, and each App request must be served within a specified target delay, $l_{\mathcal{A}_j}$, to ensure quality of service. Apps are composed of chains of MSs, where each MS performs a specific function in the application workflow. Each MS can be deployed in different quality versions ($q{\in}\mathcal{Q}_n$) depending on resource availability and performance requirements. To serve a single request, each MS version demands specific CPU cycles ($\tau_{n,q}$) and memory ($\mu_{n,q}$) resources. When multiple users request the same application, multiple instances of the same MS version may need to be deployed across the system.

As users move throughout the coverage area and network conditions change, the system dynamically migrates MSs between servers to maintain optimal performance and meet target delay requirements. During this migration process, the total application downtime, which consists of the downtime of

TABLE III
List of symbols representing system components and their descriptions

Symbol	Description
	Parameters for Servers
$\mathcal{S}$	Set of available servers
M_s	Memory capacity of server s [bytes]
C_s	Computation capacity of server s [CPU cycles/s]
V_s	Max. no. of RBs available at co-located BS of server s
s_0	Dummy server hosting virtually all possible MSs instances that may need to be actually deployed
	Parameters for Users
$\mathcal{U}$	Set of available users
$B_{u,s}$	Allocated bandwidth between user u and server s [bytes/s]
$\mathcal{A}^u$	Set of Apps requested by user u
	Parameters for Apps
$\mathcal{A}$	Set of available Apps
$\rho^u_{\mathcal{A}_j}$	No. of active requests from user u to App $\mathcal{A}_j$
$l_{\mathcal{A}_j}$	Tolerable latency for App $\mathcal{A}_j$ [s]
$\eta_{\mathcal{A}_j}$	Average size of packet entering App $\mathcal{A}_j$ (bytes)
$D_{\mathcal{A}_j}$	Max. tolerable downtime for App $\mathcal{A}_j$ [s]
	Parameters for MSs
$\mathcal{N}$	Set of available MSs
$\mathcal{Q}_n$	Set of possible versions of MS n
a_n	Set to 1 if MS n is stateful
b_n	Set to 1 if MS n is shareable
$\mu_{n,q}$	Required memory of MS n of version q [bytes]
$\tau_{n,q}$	Required computation of MS n of version q [CPU cycles]
r^q_n	Norm. dirty page rate of MS n of version q
e_{mn}	Avg. message size between MS m and MS n [bytes]
$w^{n,q}_{s,i}$	Set to 1 if instance i of MS n of version q is deployed on server s, 0 else
$x^{n,q}_{u,i}$	Set to 1 if user u is served by instance i of MS n of version q, 0 else
	Decision Variables
$y^{n,q}_{s,i}$	Binary, indicating whether instance i of MS n of version q is deployed on server s
$z^{n,q}_{u,i}$	Binary, indicating whether user u is served by instance i of MS n of version q
$\hat{\tau}^i_{n,q}$	Continuous, denotes allocated CPU to instance i of MS n of version q (CPU cycles/s)
$v_{u,s}$	Integer, denotes allocated RBs between user u and server s

its individual MSs, must be within an acceptable limit $D_{\mathcal{A}_j}$.

The system model components and their notation, with their associated parameters and variables, are summarized in Table III. Notice that, for convenience, we define a dummy server, s_0, hosting virtually all possible MSs instances that may need to be deployed to provide an App.

The App *response latency* experienced by user u requesting for the App $\mathcal{A}_j$ consists of two contributions: (i) the processing latency of individual MSs composing the App ($d^{\text{proc}}_{u,\mathcal{A}_j}$), and (ii) the communication latency between two successive MSs, and between user u and the entry MS of the App $\mathcal{A}_j$ ($d^{\text{com}}_{u,\mathcal{A}_j}$). The processing latency accounts for the CPU demand of each MS version weighted by the load from all users sharing the same MS instance, normalized by the allocated CPU cycles/s. The communication latency captures both the wireless transmission delay between the user and its associated BS, and the inter-server link delays incurred when consecutive MSs of the same App are deployed on different servers. The formal expressions for all mathematical equations are reported in Appendix A.

Next, we derive the expression of an *App downtime* due to the migration or relocation of any of its composing MSs. Since MSs are containerized, MS migration/relocation can be generalized to container migration/relocation. According to the statefulness and shareability of the MSs, three types of procedures can be used:

• **Stateless relocation**: It includes three steps: (1) create a new instance of the stateless MS container at the destination host, (2) redirect the requests to the new container instance, and (3) shut down the old instance at the source host.

• **Stateful migration**: It includes 6 steps: (1) checkpoint the running container at the source host, thus collecting both the MS state and the established connection state; (2) clear the network namespace, thus preventing network configuration conflicts in the following steps; (3) transfer the checkpoint image from source to destination host; (4) re-create and configure the network namespace at the destination to match the original one; (5) update the network flow of the connection by redirecting it towards the new network namespace; (6) restore the container from the checkpoint image.

• **User state migration**: This may happen in two cases: (i) when multiple users are using the MS container at the source host, or (ii) when a container of the same MS type is already running at the destination host. In these cases, only the user's state is transferred from the source to the destination host. It includes three steps: (1) extract the user's active state from the container at the source host; (2) transfer this state to a running container of the same MS type at the destination host; (3) redirect the user's request to the running container at the destination host.

Since an App may consist of multiple stateful and stateless MSs, we must guarantee that the *App downtime* $D^{\text{down}}_{u,\mathcal{A}_j}$, including the downtime for stateless relocations, stateful migrations, and user state migrations of all its MSs, remains within the App's tolerance limit. For simplicity and without loss of generality, we consider that all MS relocations/migrations (stateless relocations, stateful migrations, and user state migrations) are performed in parallel across all MSs of the App. We remark that the model can be easily modified to include other strategies as well, such as sequential MS migrations or hybrid approaches where some migrations are parallel while others are sequential. To prevent inter–edge-server link saturation during the parallel operation, we allocate enough bandwidth to each MS relocation/migration ensuring 50 Mbps data transfers.

Given the above assumption, for stateless MS containers, the introduced MS downtime is majorly at the container initiation phase, which is related to the size of the target container and the destination host characteristics. Thus, we define the stateless downtime of a single MS n at quality q being relocated to server s to be $d^{n,q}_s = f_{\text{stateless}}(\mu_{n,q}, s)$, where $f_{\text{stateless}}()$ is the function to estimate the stateless MS initiation time. Hence, the downtime of the stateless MSs within an App A_j requested by user u is given by the maximum of $d^{n,q}_s$ over all MS instances (i.e., $a_n{=}0$) that have been newly deployed on server s, serving user u, and belonging to App $\mathcal{A}_j$.

For stateful MS containers, the migration downtime and state data volume can be modeled and estimated in real-time using the Processing-Aware Migration (PAM) model in [24]. Note that, for those stateful MS instances which are migrated from s_0 (i.e., the dummy server hosting all possible MSs) to $s{>}0$, the MS downtime $d^{n,q}_{0,s}$ is considered to be equal to the stateless migration downtime. This is because stateful MS instances deployed on s_0 do not retain user or session-specific data. We then define the downtime of the stateful MS n at quality q, migrating from server s' to s, to be $d^{n,q}_{s',s} = f_{\text{stateful}(\mu_{n,q}, r^q_n, s', s, L_{s's})}$, where f_{stateful} is the expression estimating the stateful migration downtime in PAM. In this case, the downtime of stateful MSs composing an App A_j is given by the maximum of $d^{n,q}_{s',s}$ over all stateful MS instances (i.e., $a_n{=}1$) that are being migrated from server s' to server s, serving user u, and belonging to App $\mathcal{A}_j$.

In the case of user state migration of MS n of quality q for user u, the MS downtime is defined as $d^{n,q}{=}f_{\text{state}}(u)$. The downtime due to such procedures for an App is given by the maximum of $d^{n,q}$ over all MS instances that are stateful and (i.e., $a_n{=}1\ \ b_n{=}1$), where state of user u is migrated from an existing instance i' at source server s' to a different instance i at destination server s, and the MS belongs to App $\mathcal{A}_j$.

Finally, as all migration procedures (stateless, stateful, and user state) for all MSs are executed in parallel, the overall downtime for an App A_j requested by user u is defined as the maximum of stateless downtime, stateful migration downtime, and user state migration downtime.

VI. The Multi-microservice Application Placement Problem

This section presents the MAP problem that the orchestrator has to solve whenever any of these three events occurs: *(i)* a user u sends a new request for App $\mathcal{A}_j$; *(ii)* a user u stops using App $\mathcal{A}_j$; *(iii)* user's connection is handed over from one BS to another. Accordingly, we present below the problem formulation that optimizes the overall cost and QoS of the used Apps while meeting the system and App constraints, and then we characterize its time complexity.

For deploying the MSs composing an App, there will be a deployment cost which consists of two parts: resource

allocation cost and communication cost. The former refers to the cost of allocating computing resources to the MSs of the requested Apps on the servers, while the latter captures the cost of the communication between the user and the edge server where the entry MSs of the requested Apps are deployed, and between any two successive MSs of the requested Apps if they are deployed on different servers. Denoting by MCS_u the modulation and coding scheme used by user u, the deployment cost is as follows:

$$C(y,z,\hat{\tau},v)=\sum_{n\in\mathcal{N}}\sum_{q\in\mathcal{Q}_n}\sum_{i}\sum_{s>0} y_{s,i}^{n,q}\Big\{\frac{\mu_{n,q}}{M_s}+\frac{\hat{\tau}_{n,q}^{i}}{C_s}+ \sum_{u\in\mathcal{U}}\sum_{\mathcal{A}_j\in\mathcal{A}} z_{u,i}^{n,q}\cdot 1_{n=\mathcal{A}_j[0]}\cdot\frac{B_{u,s}}{V_s\text{MCS}_u}\Big\}. \quad (1)$$

We define the QoS offered by the App $\mathcal{A}_j$ to user u as the average normalized qualities of MS instances of the App $\mathcal{A}_j$ serving user u, i.e.,

$$\alpha_{u,\mathcal{A}_j}(z)=\frac{\sum_{n\in\mathcal{N}}\sum_{q\in\mathcal{Q}_n}\sum_i \frac{q\cdot 1_{n\in\mathcal{A}_j}\cdot z_{u,i}^{n,q}}{q_{n,\max}}}{|\mathcal{A}_j|}. \quad (2)$$

The average normalized QoS of all the Apps requested by user u and across all the users is given by, respectively,

$$\alpha_u(z)=\frac{\sum_{\mathcal{A}_j\in\mathcal{A}^u}\alpha_{u,\mathcal{A}_j}}{\sum_{\mathcal{A}_j\in\mathcal{A}^u}1_{\mathcal{A}_j\in\mathcal{A}}}, \quad \text{and} \quad Q(z)=\frac{\sum_{u\in\mathcal{U}}\alpha_u}{|\mathcal{U}|}. \quad (3)$$

The formulation of the MAP problem, including the objective function (with $\beta\in[0,1]$ balancing cost and quality objectives) as well as the constraints, is reported in the below colored box.

Multi-MS Application Placement (MAP) Problem

$$\min_{\boldsymbol{y},\boldsymbol{z},\hat{\boldsymbol{\tau}},\boldsymbol{v}} [\beta\cdot C(y,z,\hat{\tau},v)-(1-\beta)\cdot Q(z)] \quad (4a)$$

$s.t$ Apps, MSs, system constraints are met.

The three categories of constraints are described below, with their mathematical formulation provided in Appendix B.

Apps constraints ensure that, for each user u requesting App $\mathcal{A}_j$, both the end-to-end latency and the experienced downtime remain within their respective maximum tolerable values.

MSs constraints define the placement and assignment of MS instances across servers. Every instance of MS n of quality q must be placed on exactly one server, while unused instances are assigned to a dummy server. For shareable MSs, at most one instance per server is allowed, since the orchestrator can dynamically scale the CPU allocation of that instance to handle traffic from multiple users, provided that the computing budget at the server hosting the MS is not exceeded. For non-shareable MSs, each deployed instance is exclusively assigned to a single user, and if a user u is already being served by such an instance, it continues to be served by the same instance in the new deployment topology, ensuring service continuity. Furthermore, a user can access only one instance of a specific MS at any given time, and the entry MSs of all Apps requested by the same user must be co-located on the same server.

System constraints ensure that the total memory, CPU, and radio resources allocated across all MS instances and users do not exceed the available capacity at each server.

To provide intuition on the formulation, consider a simple scenario where a user requests an App $\mathcal{A}_1$ composed of two MSs, n_1 and n_2, deployed over two edge servers, s_1 and s_2. Let n_1 be stateless and n_2 be stateful. The latency constraint captures the end-to-end delay of $\mathcal{A}_1$. For instance, if n_1 is placed on s_1 and n_2 on s_2, the total latency includes user-to-s_1 communication (decided by $v_{u,s}$), processing at n_1 (decided by CPU allocation to n_1), inter-server communication between s_1 and s_2, and processing at n_2 (decided by CPU allocation to n_2), which must satisfy the latency bound. When the user moves (e.g., towards s_2), migrating n_1 to s_2 may reduce communication delay, but introduces downtime. This is captured through constraints limiting the allowable service disruption. Consider a second user requesting another App that also requires n_1. Since n_1 is stateless, it can be shared across the two users, avoiding redundant deployment. Instead, n_2, being stateful, may require a separate instance depending on user-specific state. This reflects the shareability constraints in the formulation. Finally, if multiple versions of an MS are available, selecting a lower-quality version reduces resource usage at the cost of higher processing, reflecting the trade-off captured in the formulation. This example illustrates how placement, resource allocation, and migration decisions are jointly constrained in the model.

Problem Complexity. The following theorem holds:

Theorem 1. *The MAP problem is NP-hard.*

Proof. We prove NP-hardness of the MAP problem by showing that the Multi Dimensional Bin Packing (MDBP) Problem [35], which is NP-hard, can be reduced to an instance of our placement problem in polynomial time. Consider a simplified version of our placement problem, where each App consists of a single, non shareable stateful MS. This simplified version is an instance of MDBP, which, given a set of multi-dimensional items, aims to pack the items into bins to minimize the number of used bins. Additionally, MDBP aims to achieve the above objective in such a way that the total size of the items in each dimension does not exceed the bin capacity. The mapping between MDBP and an instance of our placement problem can be performed in polynomial time by considering: (i) items are Apps, each with multiple dimensions, i.e., resource demand, experienced latency, and tolerable downtime; (ii) bins are servers and the number of used bins is the deployment cost; (iii) bin capacity along the various dimensions is the amount of available resources at the server, the App's target delay, and the App's maximum tolerable downtime. Thus, the thesis is proven. □

VII. The STEP Algorithm

In light of the complexity of the MAP problem, we introduce the low-complexity STEP (State and Topology-aware Edge-MS Placement), which, consistently with the MAP problem definition, runs at the orchestrator whenever a request for a new App is received, an App is terminated, or a user

handover occurs. STEP includes three main steps. *First,* we create a multi-layer DNTG based on the BSs to which users are connected and the Apps they request (Sec. VII-A). *Second,* starting from the above DNTG, we build a (smaller) decision graph by retaining only those vertices that are relevant to service deployment decisions, thus ultimately reducing the decision-making complexity (Sec. VII-B). *Third*, to identify the set of feasible decisions and select the one that minimizes the deployment cost, we translate the decision graph into an expanded graph by embedding the weights in the decision graph into the vertices of the expanded graph (Sec. VII-C).

A. DNTG Construction

The DNTG comprises four hierarchical layers that represent the key system components and their interactions, as depicted in Fig. 7. The bottom-most layer is the user layer, consisting of one vertex per user $u \in \mathcal{U}$. The infrastructure layer (bottom) consists of one vertex per server $s \in \mathcal{S}$. The service layer (top) represents the complete space of possible MS instances. Since each MS $n \in \mathcal{N}$ can be implemented in $q \in \mathcal{Q}_n$ different quality versions with i possible instances, this layer includes a total of $|\mathcal{N}| \cdot \mathcal{Q} \cdot i$ vertices. Finally, the App layer (top-most) contains vertices presenting the App set $\mathcal{A}$.

The purpose of the DNTG is to model all possible deployment choices of MSs for the Apps requested by the users. We identify the *contact events* between users and servers, indicating user connection handover events, as well as between users and Apps, indicating App request arrival or termination events. We refer to the time interval between any two successive contact events in the network as *frame*. Within a frame, neither the MSs deployment is changed nor links are created or removed. Let F denote the number of frames present in the considered traces, and Δt_k is the duration of the generic frame k ($1 \leq k \leq F$). All ongoing contact events during frame k are said to be active in that frame.

At frame k, each user u in the network is represented by a vertex u^k in the user layer of the DNTG, while each edge server s is mapped onto the vertex s^k in the infrastructure layer. The instance i of MS n of quality q_n at frame k is represented by a vertex (n^k, q_n^k, i^k) and App $\mathcal{A}_j$ is mapped onto the vertex $\mathcal{A}_j^k$. Then let $\mathcal{U}^k$, $\mathcal{S}^k$, $\mathcal{D}^k$, and $\mathcal{A}^k$ be the set of vertices representing, respectively, the users, edge servers, MS instances, and Apps in the DNTG at time frame k.

Within each frame k, the DNTG can contain the following directed edges connecting vertices corresponding to users, servers, MSs, or Apps (i.e., $u^k \in \mathcal{U}^k$, $s^k \in \mathcal{S}^k$, $(n^k, q_n^k, i^k) \in \mathcal{D}^k$, $\mathcal{A}_j^k \in \mathcal{A}^k$):

- $(u^k, \mathcal{A}_j^k)$ exists if user u has requested App $\mathcal{A}_j$; the weight $w(u^k, \mathcal{A}j^k)$ of this edge is equal to $(\eta_{\mathcal{A}_j}, \rho^u_{\mathcal{A}_j})$;
- (u^k, s^k) exists if user u is connected to server s; its weight $w(u^k, s^k)$ is set to $V_s \text{MCS}_u$;
- $(\mathcal{A}_j^k, (n^k, q_n^k, i^k))$ exists if MS n is the entry MS of App $\mathcal{A}_j^k$; its the weight $w(\mathcal{A}_j^k, (n^k, q_n^k, i^k))$ is set to $(\eta_{\mathcal{A}_j}, \rho^u_{\mathcal{A}_j})$;
- $((n^k, q_n^k, i^k), (m^k, q_m^k, j^k))$ exists if both MS n and MS m are part of the same App and n communicates with m; the weight of this edge $w((n^k, q_n^k, i^k), (m^k, q_m^k, j^k))$ is set to e_{mn};

Fig. 7. DNTG structure example with four hierarchical layers. From bottom to top: User Layer (1 user), Infrastructure Layer (2 edge servers plus the dummy server s_0), Service Layer (2 MS instances with quality variants), and App Layer (1 App) across two consecutive frames (k and $k+1$). Virtual vertices α (source) and ω (destination) are added to enable path-based optimization.

- $(s^k, \hat{s}^k)$ exists if there exists a wired network connection between servers s_k and $\hat{s}_k$; its weight is set to $B_{s,\hat{s}}$.

Additionally, the following directed edges connect the vertices representing the same node across consecutive frames:

- (s^k, s^{k+1}) that connects server vertex s^k at frame k to server vertex s^{k+1} at frame $k+1$. The weight here represents the remaining resources (CPU, memory, and resource blocks) at server s;
- $((n^k, q_n^k, i^k), (n^{k+1}, q_n^{k+1}, i^{k+1}))$ connecting MS instance vertex at frame k to its corresponding vertex at frame $k+1$. The weight represents the properties of the container hosting that specific MS instance (e.g., dirty page rate);
- $(\mathcal{A}_j^k, \mathcal{A}_j^{k+1})$ connecting an App vertex at frame k to its corresponding vertex at frame $k+1$. The weight of this edge equals the deployment cost and QoS of the App $\mathcal{A}_j$.

Next, we use the above DNTG to formulate a min-cost problem, which allows for a low-complexity solution of the MAP problem. To this end, we add to the DNTG two virtual vertices, α and ω, representing, respectively, the source and destination of a path over the graph. The graph is then completed with 0 weight edges (α, u^1) from α to any user vertex $u^1 \in \mathcal{U}^1$, and edges $(\mathcal{A}_j^k, \omega)$ from any App vertex $\mathcal{A}_j^k$ to ω, where $1 \leq k \leq F$.

B. The Decision Graph

For each frame of the DNTG, we construct a decision graph $G=(V, E)$ that captures possible deployment decisions to meet the App's requirements while reducing the deployment cost. This graph is a simplified version of the corresponding DNTG frame, containing only the vertices from each layer that are relevant to the deployment decisions in that frame;

for simplicity, we drop from the notation the dependency on the specific frame k.

As a preliminary step, we identify changes in the current frame compared to the previous one, with vertices included in G varying based on the event type. For new App requests, G includes the requesting user u, the requested App $\mathcal{A}_j$, its associated MSs, and servers with sufficient resources for deployment. In contrast, for user migration or App termination events, G also includes all Apps requested by u, as the migration of an App may require the migration of MSs shared with other Apps, while termination necessitates CPU reallocation. To simplify the decision graph further, we merge the service and infrastructure layers of the DNTG frame into a deployment layer, where each vertex is represented as (n, q_n, i, s), indicating that the i-th instance of MS n of quality q_n is deployed on server s, as shown in Fig. 8.

Further, for each vertex in this layer, we create replicas corresponding to predefined CPU allocations (e.g., $\{0.05\text{G}, 0.1\text{G}, 1.5\text{G}\ldots\}$ cycles/s) and RB allocations (e.g., $\{1, 2, \ldots\}$), and these replica nodes are of the form $(n, q_n, i, s, \hat{\tau}^i_{n,q}, v_{u,s})$. Thus, selecting a vertex in the deployment layer directly determines the MS instance, quality, server, allocated resources, as well as the associated deployment cost and service quality. To reduce the size of the decision graph, we consider only one unused instance per (n, q_n) and include only those already deployed instances that are eligible for reuse. This significantly reduces search space while preserving all feasible deployment decisions.

The above pruning steps directly determine how STEP scales with the number of users, MS instances, and edge servers. Since the decision graph is constructed locally for the affected user's Apps only, its size remains proportional to the number of MS instances associated with the affected user and their requested Apps, rather than the total number of users or MS instances in the system. As the number of users grows, the number of MS instances eligible for sharing increases, causing a marginal expansion of the decision graph and a slight increase in decision time. Also, servers that cannot satisfy the App response latency constraint are excluded from the decision graph entirely, ensuring that increasing the number of edge servers does not significantly inflate the decision time.

Since certain deployment decisions in G may violate MS shareability and entry MS placement constraints, we ensure feasibility by pruning G as follows:

• **For entry and non-entry MS vertices**: Since entry MSs of Apps requested by a user must be placed on the same server, preferably the one to which user u is connected, we exclude vertices where server differs from the user's connected server. Further, since entry MS vertices also represent RB allocations, we ignore those with $v_{u,s} = 0$. Conversely, only non-entry MS vertices with $v_{u,s} = 0$ are considered.

• **Multiple MS instances on same server**: If a server already has another instance of the same MS and quality, and the MS is shareable, we ignore additional instances of the same MS configuration on the same server to allow MS reusability.

Every generic edge (v, v') in E has the following properties:

• **Response latency** $D(v, v')$**:** This represents the response latency contribution if the edge (v, v') is included in the final deployment. By default, response latency is an additive constraint, i.e., summing the response latency attributes of all edges included in the final deployment yields the App response latency. $D(v, v')$ is calculated as follows: (i) If v is an App vertex and v' is a deployment layer vertex, then $D(v, v')$ is set to the sum of communication latency from the user to the entry MS of the App (based on RB allocation specified in v') and processing latency of v' (based on CPU allocation specified in v'). (ii) If both v and v' are deployment layer vertices, then $D(v, v')$ is set to the sum of communication latency if servers of v and v' are different and the processing latency of v'. (iii) If the edge (v, v') is from user layer to App layer or from the last MS of the App to ω or from α to the user layer, the response latency is set to 0.

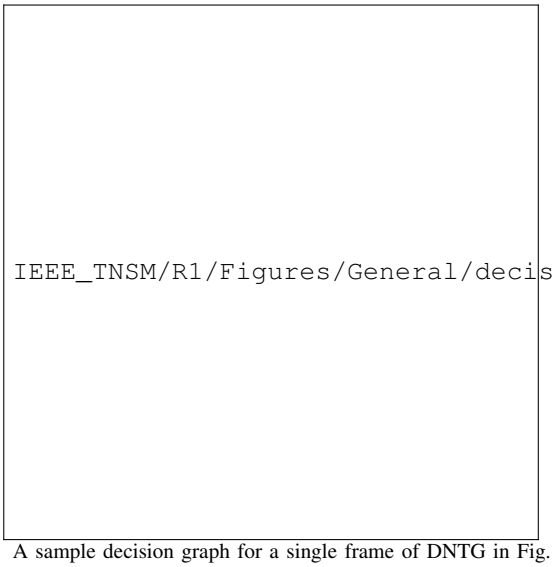

Fig. 8. A sample decision graph for a single frame of DNTG in Fig. 7.

• **Migration downtime** $M(v, v')$**:** It represents the contribution of edge (v, v') to the App's migration downtime when foreseen in the final deployment. Unlike response latency, migration downtime is not directly an additive property, since migrations of MSs can be executed simultaneously, and the App's total migration downtime is determined by the maximum downtime among its MSs. We then convert migration downtime as an additive constraint by adopting an approach that flags violations rather than tracking cumulative contributions. For each edge (v, v'), we define $M(v, v')$=K if the downtime of the MS represented by v' exceeds the maximum allowed downtime $D_{\mathcal{A}_j}$ for the App and $M(v, v')$=0 otherwise, where $K > 1$. This transformation ensures any path containing at least one MS with downtime greater than the maximum allowed downtime will have a total migration downtime weight exceeding 1. Like before, if the edge (v, v') is from user layer to App layer or from the App last MS to ω, or from α to the user layer, migration downtime is set to 0. Based on the above additive properties, for any generic edge (v, v') of G for user u requesting $\mathcal{A}_j$, we assign a multi-dimensional weight defined as:

$$w(v, v') = \left(\frac{D(v, v')}{l_{\mathcal{A}_j}}, M(v, v') \right). \tag{5}$$

C. The Expanded Graph: Guaranteeing Additive Constraints and Decision Feasibility

Given the decision graph for user u requesting App $\mathcal{A}_j$, we outline the procedure to find the set of feasible deployment decisions that meet the end-to-end performance constraints. Note that this procedure needs to be repeated for each App requested by user u in case of user migration or App termination events. Since finding a path between a source and destination vertex that satisfies two or more end-to-end constraints is NP-hard [36], we adopt the approach from [36], [37] to construct an expanded graph. We now describe the procedure to build such an expanded graph for our problem, given a positive integer resolution parameter γ:

• For each vertex in the deployment and App layers of the decision graph, create $(\gamma+1)^2$ corresponding vertices, where the exponent 2 reflects the number of additive constraints. We denote these vertices, whose number will be equal to the number of additive constraints, as $v^{0,0}, v^{0,1}, \ldots, v^{0,\gamma}, \ldots, v^{\gamma,\gamma}$.

• For every generic edge (v, v') from App to deployment layer or within the deployment layer, create directed edges from each vertex $v^{i,j}$ to vertex $v'^{i+\lceil \gamma w_0(v,v') \rceil, j+\lceil \gamma w_1(v,v') \rceil}$, if such a vertex exists. Here, $w_0(v, v')$ and $w_1(v, v')$ represent response latency and migration downtime, respectively.

The weights present in the decision graph become embedded into the vertices of the expanded graph. This construction ensures that any path from α to ω in the expanded graph honors the considered additive constraints. Then to find the path that minimizes the deployment cost in addition to honoring the additive constraints, we define edge weights in the expanded graph as follows. For edges from App to the deployment layer or within the deployment layer, we associate each edge (v, v') with the cost of deploying the MS associated with v'. All other edges in the expanded graph have a weight of 0. To enable MS sharing, if the MS associated with the vertex v' is already deployed, we assign an edge weight of 0 to (v, v'). Then we apply Dijkstra's algorithm to such an expanded graph and find the minimum cost path.

As also noted in [37], a smaller value of γ results in fewer vertices and paths in the expanded graph, introducing a quantization error that may lead to suboptimal solutions, since fewer levels of KPI target consumption can be distinguished. Conversely, a larger value of γ increases the number of vertices and paths explored, improving solution quality at the cost of a higher decision time. As γ increases, STEP can get arbitrarily close to the optimal solution within the decision graph, though larger values of γ would lead to increased decision times.

CPU recalibration. After finding the minimum cost path for App $\mathcal{A}_j$ requested by user u (or the minimum cost paths for each App $\mathcal{A}_j \in \mathcal{A}^u$ in case of migration or termination events), we perform an additional CPU recalibration step. Since we initially used a predefined set of CPU cycles/s allocations, the resulting CPU allocations in the deployment decision may not be optimal. In the recalibration phase, we fix all deployment decisions from the minimum-cost path, specifically the MS-to-server mappings, selected MS instances and versions for the users, and RB allocations to optimize only the CPU allocations. This transforms the problem into a convex optimization where we minimize the total CPU cost while ensuring that the response latency remains within the App's requirement. This transformation makes the problem convex, hence solvable in polynomial time.

VIII. Performance Evaluation

We first compare STEP against the optimum, the baseline algorithms described in Sec. IV, and state-of-the-art benchmarks introduced in Sec. VIII-A, in a small-scale scenario, which makes it feasible to compute the optimal. We then evaluate STEP and its alternatives in a large-scale scenario by implementing the solution provided by these schemes in our testbed and experimentally assessing their performance.

A. Benchmarks

Among the benchmarks used in the small-scale scenario, the **Optimal** solution is obtained by solving the formulated MAP problem via the Gurobi solver, while the **ALC**, **MLC**, and **MCLL** baselines operate as discussed earlier in Sec. IV. In addition, we include the following two benchmarks:

• **E2MS** [10]: Minimizes a cost function consisting of service disruption cost during migration, communication cost, and image pull cost. It assumes non-shareable MSs, operates with a single fixed quality version for each MS, and does not support dynamic reuse or quality adaptation. Since E2MS can produce multiple solutions with identical optimal cost but different CPU allocations, we added upper bounds on CPU allocations. This prevents computationally expensive searches for minimal CPU usage and ensures a fair comparison with other methods.

• **SR-CL** [8]: A reinforcement learning-based approach that selects migration actions to minimize a reward function defined as the negative sum of migration delay, communication delay, and processing delay, and follows a convex optimization step for CPU allocation after placement decisions. Similar to E2MS, it does not support MS sharing across Apps or users and considers a single fixed quality version for each MS. Since SR-CL treats Apps as monolithic one-MS services only, we consider it only in the large-scale single-MS App deployment scenario, and exclude it from the multi-MS scenarios.

B. Small-scale scenario

We focus on a scenario with four MSs: two stateless MSs, one stateful shareable MS, and one stateful non-shareable MS. These MSs are used to create four Apps, each composed of two MSs: App_1 (MS1→MS2), App_2 (MS1→MS3), App_3 (MS2→MS3), and App_4 (MS2→MS4). The system infrastructure includes four servers that can host the Apps MSs, serving four users; App requests follow a Poisson process. To evaluate our system's performance under a high migration rate and load, we set the App request rate to 2 per unit time and the normalized load per App at 0.2, which yields a total system normalized load of 0.8. The user migration also follows a Poisson process with a rate of 4 migrations per unit time, allowing us to analyze the system's behavior under a consistent user mobility pattern. For simplicity, we set $Q_n = \{1\}$ for each MS n (i.e., each MS can only have a

Fig. 9. Small-scale scenario: Performance of optimal vs. STEP and benchmark solutions. CPU cost (top left), memory cost (top right), response latency (bottom left), migration downtime (bottom right).

single version), though multiple instances of the same MS can be deployed. We set the target response latency of each App to 1 s, and the maximum tolerable App downtime to 1 s. Other parameters, such as CPU cycle and memory requirements, and the migration downtime of the MSs are configured based on experimental values from our testbed. Notice that, since here we consider the server load to be low, both *ALC* and *MLC* result in the same deployment topology. Thus, we show only the *ALC* and *MCLL* baselines in the comparison. Also, in these baseline solutions, CPU and RB allocations are selected from a few predefined possible levels, while ensuring that the response latency of the deployed Apps does not exceed their required maximum value. We also evaluate STEP-ns, a variant of STEP with shareability disabled, to isolate the contribution of STEP's resource allocation strategy from the system features it leverages, namely, MS shareability and multi-version quality selection.

During each event in the system, we measured various performance metrics and report the average over independent runs, along with the corresponding 95% confidence intervals. The percentage improvement or degradation is annotated relative to the Optimal solution. Fig. 9 (top left) shows the normalized CPU cost incurred while deploying the MSs of the requested Apps for the different approaches. Remarkably, STEP exhibits only a 7% increase in CPU cost compared to the Optimal, demonstrating efficient resource allocation. All other approaches have higher CPU costs than STEP and Optimal, since they do not optimize CPU allocations. STEP-ns achieves lower CPU cost than STEP because dedicated MS instances only need to satisfy the latency requirement of a single App, whereas shared instances require higher CPU allocation to satisfy multiple Apps simultaneously. Fig. 9 (top right) presents the normalized memory costs for deploying the MSs of the requested Apps. Since ALC, MCLL, and E2MS do not support sharing of MSs across different users or the Apps requested by the same user, they deploy a higher number of MSs, leading to a higher memory cost than Optimal and STEP. Importantly, the memory cost of STEP closely matches that of Optimal. STEP-ns incurs the same memory cost as the baselines because it deploys the same number of MS instances to serve user requests, without benefiting from shareability.

As illustrated in Fig. 9 (bottom left), *ALC* and *MCLL*, and *E2MS* achieve lower response latency, but this comes at the cost of higher CPU consumption without providing a real advantage as all schemes can meet the Apps latency requirement of 1 s. Finally, Fig. 9 (bottom right) shows that also all downtimes are within the maximum tolerable downtime. E2MS achieves lower downtime because minimizing this metric is one of its objectives, while other approaches incur higher migration downtime compared to the optimal solution. Interestingly, STEP-ns incurs higher migration downtime than STEP because shareability reduces migration overhead: when the destination server already hosts a shared MS instance, stateless MSs require no migration, and stateful MSs only require data migration rather than full container migration.

TABLE IV
SENSITIVITY ANALYSIS OF γ: CPU COST & DECISION LATENCY

γ	CPU Cost	Decision Latency (ms)
3	0.0218 ± 0.0006	2.4 ± 0.2
9	0.0218 ± 0.0006	2.5 ± 0.2

To characterize the trade-off between solution quality and decision time, we evaluate STEP with γ=3, 9 in the small-scale scenario where the optimal solution can be computed. Table IV reports the deployment cost and decision latency for different γ's. Remarkably, STEP achieves near-optimal CPU cost already at γ=3, yielding stable values of both cost and decision latency across the tested values, with low computational overhead.

C. Large-scale scenario

We now focus on a large-scale scenario with ten MSs: four stateless MSs (MS1, MS5, MS6, MS7), three stateful shareable MSs (MS2, MS3, MS4), and three stateful non-shareable MSs (MS8, MS9, MS10). Further, each MS has two quality levels (lower and higher, marked with level index 1 and 2). These MSs are used to create 6 Apps: App_1-App_6 (the composition of the Apps will be mentioned later for the different test cases). Using our testbed (Sec. III), we also emulate up to 25 users (UAVs) moving along pre-defined paths in the edge system coverage area and request Apps with a rate of 1 request/s. The App-user relationship is as follows:

$$\begin{aligned}
\text{User}_1, \text{User}_7, \text{User}_{13}, \text{User}_{19}, \text{User}_{25} &\rightarrow \text{App}_1 \\
\text{User}_2, \text{User}_8, \text{User}_{14}, \text{User}_{20} &\rightarrow \text{App}_2 \\
\text{User}_3, \text{User}_9, \text{User}_{15}, \text{User}_{21} &\rightarrow \text{App}_3 \\
\text{User}_4, \text{User}_{10}, \text{User}_{16}, \text{User}_{22} &\rightarrow \text{App}_4 \\
\text{User}_5, \text{User}_{11}, \text{User}_{17}, \text{User}_{23} &\rightarrow \text{App}_5 \\
\text{User}_6, \text{User}_{12}, \text{User}_{18}, \text{User}_{24} &\rightarrow \text{App}_6 \,.
\end{aligned}$$

As a user moves, the MCS used on the radio link between the user and the BS with which the user communicates varies,

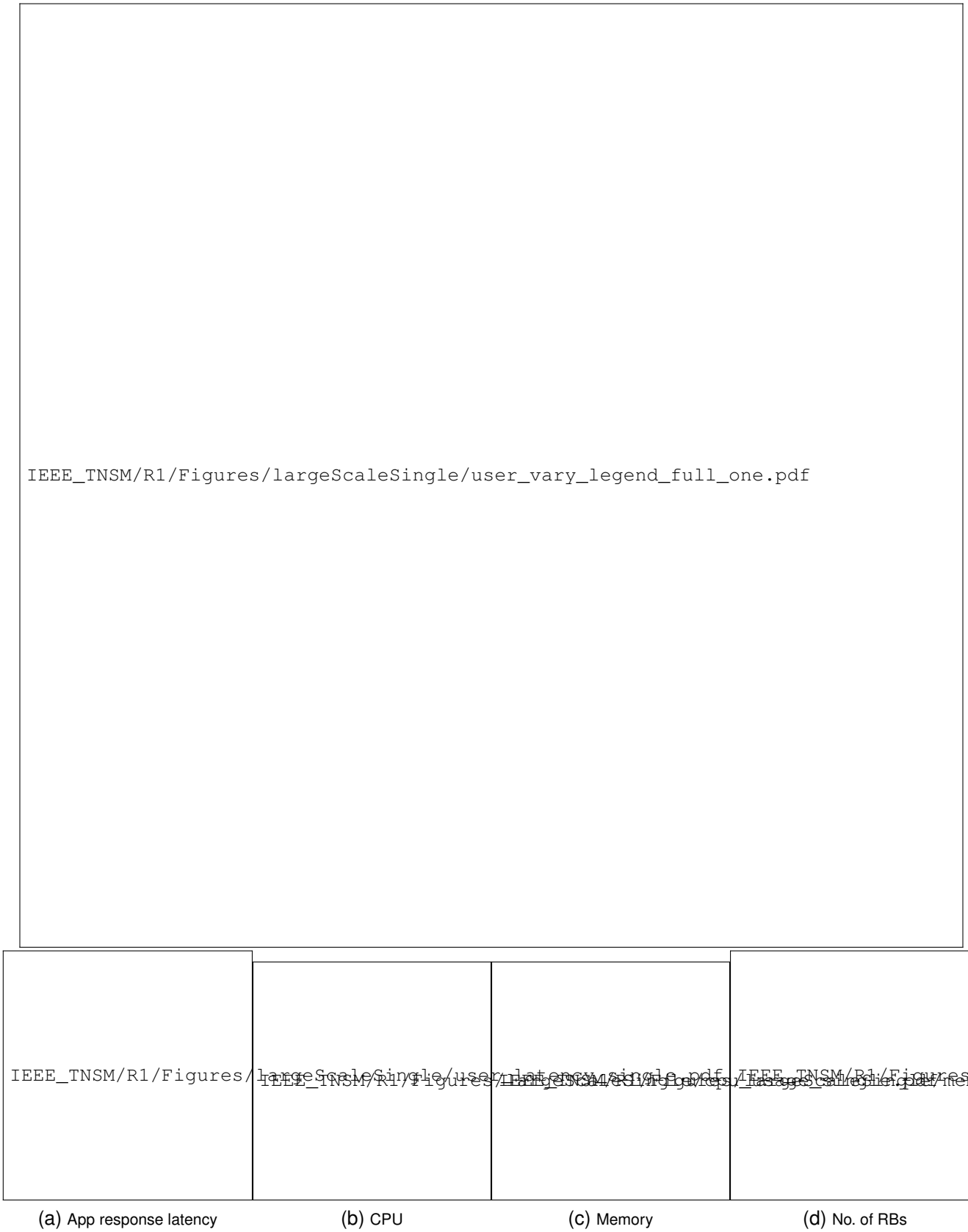

Fig. 10. Large-scale scenario with single-MS Apps: App response latency (a) of each user and CPU (b), Memory (c), and RB (d) usage of each edge server when the MS deployment is orchestrated by STEP or its alternatives. Each marker represents the averaged measurement for each user/edge server during emulation. The normalized standard deviation of each measurement in the case of 25 users is reported in Table V.

according to the user-BS distance. Also, a user always connects with the BS (and the edge server co-located with it) that offers the best connectivity quality. Upon a handover, STEP (or the baseline algorithm used for comparison) is executed at the

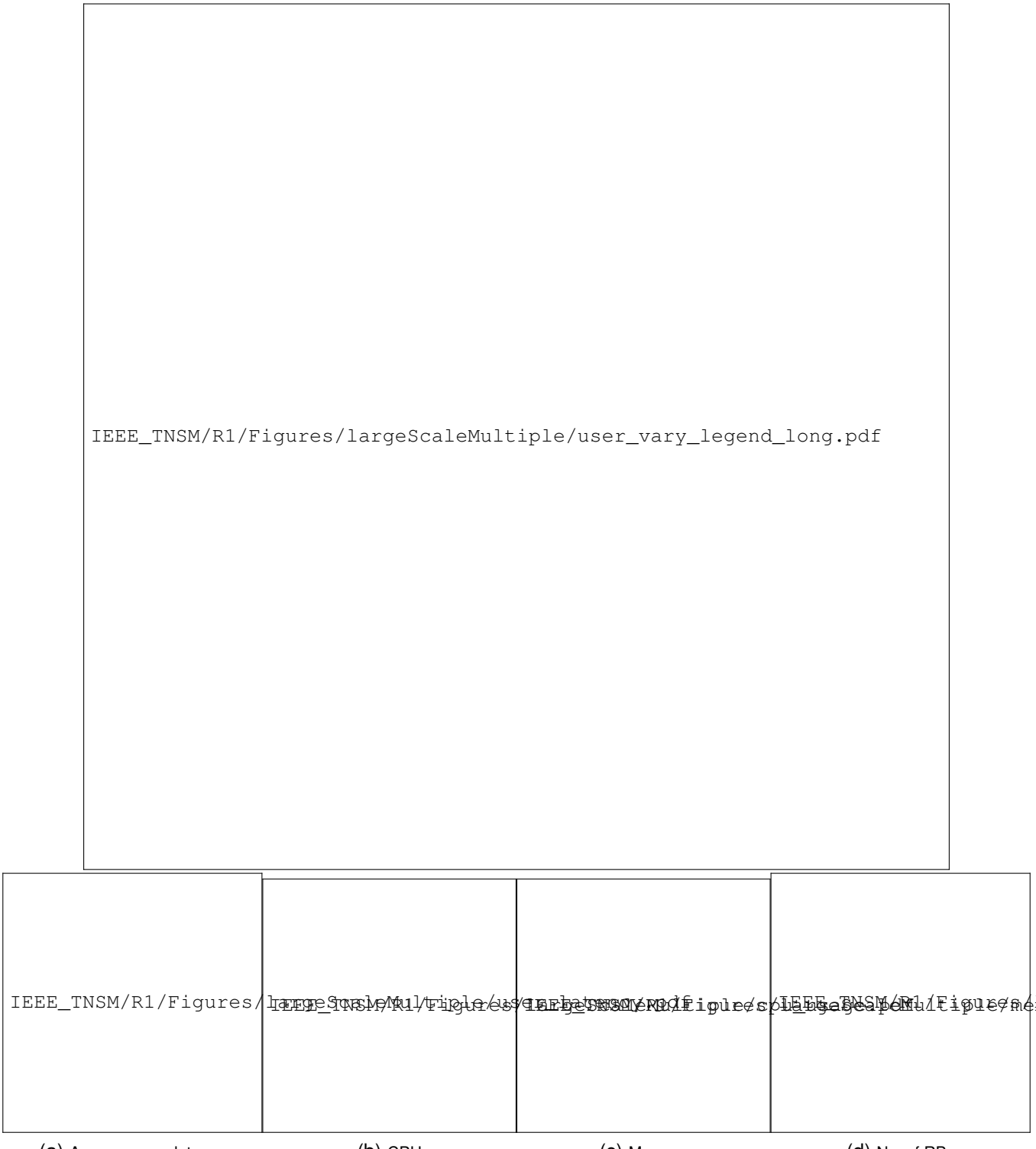

Fig. 11. Large-scale scenario with multiple-MS Apps: App response latency (a) of each user, and CPU (b), Memory (c), and RB (d) usage of each edge server when the MS deployment is orchestrated by STEP and its alternatives. Each marker represents the averaged measurement of each user/edge server during emulation. The normalized standard deviation of each measurement in the case of 25 users is reported in Table V.

Kubernetes orchestrator for re-assessing the Apps deployment, and the orchestrator starts MS migration/relocation as needed. During an experiment, lasting 1,500 s, the rate of events that may cause changes in the MSs deployment (i.e., user arrival/handover/departure) is set to 3 events/min. For all Apps, we set the target response latency to 1 s and the maximum tolerable downtime to 2 s. Other parameters such as CPU cycle and memory requirements, as well as downtime of the MSs are configured based on the values measured in our testbed.

We first evaluate the orchestration performance when each App is composed of only one MS: App_1 (MS1), App_2 (MS2), App_3 (MS3), App_4 (MS4), App_5 (MS5), and App_6 (MS6),

TABLE V
STANDARD DEVIATION NORMALIZED TO THE AVERAGE OF EACH MEASUREMENT FOR 25 USERS (SINGLE-MS ; MULTIPLE-MS)

Algo.	Latency [%]	CPU [%]	Memory [%]	RB [%]
STEP	22.46 ; **19.79**	**46.36** ; **33.34**	**30.88** ; **42.51**	51.18 ; 51.03
ALC	23.03 ; 20.74	54.33 ; 64.67	51.33 ; 52.60	29.05 ; 28.99
MCLL	22.57 ; 20.33	50.04 ; 36.05	47.77 ; 49.42	**27.84** ; **27.77**
E2MS	18.16 ; 32.29	49.03 ; 41.66	47.99 ; 104.81	51.11 ; 51.03
SR-CL	**17.91** ; –	53.48 ; –	49.58 ; –	64.74 ; –

and only the low quality level MS (quality index 1) can be used. In the test, we vary the number of users in the edge from 10 to 25 when STEP and the considered baseline algorithms are adopted, and show the average App response latency of each user in Fig. 10a. The average CPU, memory, and RB usage at each edge server are instead illustrated in Figures 10b, 10c, and 10d, respectively. To further highlight performance variability, Table V reports the normalized standard deviation of each metric in the 25-user case.

From Fig. 10a, we observe that in such a simple scenario all orchestration algorithms achieve a similar (low) value of latency (about 500 ms), well below the 1 s target maximum, thereby ensuring nearly 100% request success rate. Notably, ALC, MCLL, and SR-CL reduce latency by about 30 ms compared to STEP, but at the cost of 2.5% higher CPU usage and 15% higher RB usage (see Figures 10b and 10d). This highlights STEP's advantage in balancing user experience with edge resource consumption. In addition, compared to E2MS, STEP achieves on average 25 ms lower application latency and lower CPU usage, while consuming the same amount of RBs, demonstrating the superiority of STEP in MS placement. As shown in Fig. 10c, STEP reduces edge memory usage compared to the other schemes when the number of users at the edge is high, reflecting the benefits of its MS sharing capability. Finally, Table V provides additional insights into performance stability by presenting the standard deviation of the measured performance normalized to its average value. Looking at the single-MS App case, once can notice that, while SR-CL and E2MS achieve lower normalized standard deviation in user latency, they exhibit significantly higher variability in CPU, memory, and RB usage. This indicates less stable and less balanced resource allocation. In contrast, although STEP exhibits higher variability in radio resources occupation, it shows consistently moderate variability in latency, and significantly lower variability in CPU and memory consumption than all its alternatives.

Then we evaluate the orchestration performance of STEP when the composition of the Apps is complex: App_1 (MS1→MS2→MS7), App_2 (MS1→MS3→MS8), App_3 (MS4→MS6→MS7), App_4 (MS5→MS6), App_5 (MS7→MS8), and App_6 (MS9→MS10), and each MS has two quality level options (indexed with 1 and 2). We remark that, since SR-CL can only orchestrate monolithic services composed of one MS only, it is excluded from this test. Similar to the previous tests, we vary the number of users served by the edge Apps from 10 to 25, and show the average App response latency of each user in Fig. 11a, and the average CPU, Memory, and RB usage of each edge server in Figures 11b, 11c, and 11d, respectively. Table V reports the normalized standard deviation of each metric in the 25-user case. To further clarify the performance difference of the considered schemes, Table VI presents the key orchestration performance metrics including the deployment cost $C(y, z, \hat{\tau}, v)$, average App quality $Q(z)$, request success rate, and latency and App quality fairness (based on Jain's fairness index).

TABLE VI
LARGE-SCALE SCENARIO: ORCHESTRATION PERFORMANCE (β=0.5)

Index	$\lvert\mathcal{U}\rvert$	STEP (β=0.5)	ALC	MCLL	E2MS
Avg. Dep. cost $C(y, z, \hat{\tau}, v)$	15	**1.84 ± 0.12**	1.99 ± 0.25	2.05 ± 0.12	2.49 ± 0.56
	20	**2.33 ± 0.15**	2.39 ± 0.21	2.51 ± 0.11	3.19 ± 0.70
	25	**2.68 ± 0.13**	3.15 ± 0.29	3.21 ± 0.10	3.93 ± 0.88
Avg. App quality level $Q(z)$	15	**1.52 ± 0.12**	1	1	1
	20	**1.44 ± 0.21**	1	1	1
	25	**1.49 ± 0.19**	1	1	1
Avg. orch. decision latency [ms]	15	16.6 ± 9.6	1.3 ± 0.6	**0.7 ± 0.5**	13.1 ± 8.5
	20	23.9 ± 9.8	**0.7 ± 0.5**	0.8 ± 0.5	15.6 ± 10.3
	25	43.4 ± 8.5	**0.6 ± 0.2**	0.8 ± 0.5	23.2 ± 30.5
Avg. orch. operation latency [s]	15	**0.37 ± 0.43**	0.46 ± 0.29	0.47 ± 0.30	0.49 ± 0.34
	20	**0.38 ± 0.42**	0.46 ± 0.28	0.46 ± 0.30	0.49 ± 0.33
	25	**0.34 ± 0.41**	0.49 ± 0.50	0.57 ± 0.29	0.52 ± 0.32
Avg. App downtime per-event [s]	15	**0.52 ± 0.50**	0.69 ± 0.39	0.67 ± 0.39	0.60 ± 0.37
	20	**0.52 ± 0.50**	0.68 ± 0.39	0.66 ± 0.39	0.61 ± 0.37
	25	**0.48 ± 0.50**	0.74 ± 0.38	0.71 ± 0.37	0.63 ± 0.34
Avg. num. mig. MS per-event	15	**0.62 ± 0.63**	1.87 ± 1.15	1.76 ± 1.15	1.18 ± 0.97
	20	**0.61 ± 0.63**	1.86 ± 1.14	1.71 ± 1.14	1.21 ± 0.99
	25	**0.57 ± 0.63**	2.00 ± 1.09	1.74 ± 1.10	1.20 ± 0.92
Request success rate	15	94.93%	93.87%	97.06%	**99.74%**
	20	98.55%	89.72%	93.37%	**97.02%**
	25	95.32%	85.23%	85.22%	**97.78%**
Latency fairness	15	0.95	0.95	**0.96**	0.95
	20	0.95	0.95	**0.96**	0.91
	25	0.94	0.95	**0.97**	0.93
App quality fairness	15	0.97	1	1	1
	20	0.98	1	1	1
	25	0.96	1	1	1

Fig. 11a highlights that E2MS yields the lowest App response latency at around 450 ms, but at the expense of the highest CPU usage (Fig. 11b). STEP, ALC, and MCLL all achieve around 700 ms latency, yet their trade-offs differ. On the one hand, STEP consumes about 10% more CPU and 1.7% more memory than ALC and MCLL (Figures 11b–11c), mainly because it prioritizes higher MS quality levels (as confirmed by the values in Table VI). On the other hand, STEP requires about 15% fewer RBs than ALC and MCLL (Fig. 11d). According to Table V, STEP achieves the lowest normalized standard deviation for user latency, and substantially lower CPU and memory usage in the case of 25 users, which again demonstrates robustness and stability of STEP in complex application scenario.

Overall, Table VI indicates that STEP consistently achieves the lowest deployment cost and highest App quality across all cases and exhibits the slowest cost increase as the number of users grows. Although STEP incurs a higher orchestration decision latency (on the order of tens of milliseconds), it achieves the lowest migration operation latency, resulting in an overall *reduction of total orchestration time (decision +*

operation) by up to 20% compared to its alternatives. These results are consistent with the scalability properties of the decision graph construction discussed in Sec. VII-B. Moreover, STEP yields the smallest App downtime and the minimum number of migrated microservices per event, demonstrating its effectiveness in limiting service disruption and improving system stability. While E2MS excels in request success rate due to aggressive resource consumption, STEP achieves the second-best success rate (never falling below 90%), together with higher App quality. This comes at the cost of a slightly lower latency compared to the benchmarks, but still below the maximum tolerable value, and quality fairness that, however, results from the higher-quality MSs it yields.

TABLE VII
LARGE-SCALE SCENARIO: ORCHESTRATION PERFORMANCE FOR 25 USERS AND VARYING β

β **value**	**0.25**	**0.50**	**0.75**	**0.90**
Avg. CPU usage [%]	24.03 ± 8.12	23.59 ± 8.31	8.69 ± 9.82	6.07 ± 9.47
Avg. Memory usage [%]	20.13 ± 7.84	19.85 ± 7.49	14.08 ± 8.96	13.06 ± 9.60
Avg. RB usage [%]	25.60 ± 8.34	25.59 ± 9.55	25.58 ± 8.64	25.99 ± 9.65
Avg. Dep. cost $C(y, z, \hat{r}, v)$	2.79 ± 0.15	2.68 ± 0.13	1.93 ± 0.18	1.8 ± 0.12
Avg. App quality $Q(z)$	1.50 ± 0.15	1.49 ± 0.15	1.44 ± 0.16	1.34 ± 0.10

Moreover, to validate STEP's orchestration performance in different application scenarios, we evaluate the impact of varying β values in the objective function (4a). We recall that a larger β places more emphasis on minimizing system resource usage, while a smaller β prioritizes user experience. The results in Table VII highlight that the average deployment cost $C(y, z, \hat{r}, v)$ decreases as β grows, with a reduction of up to 35.48%. A similar trend can be observed for CPU and memory usage, while RB usage remains nearly constant across different β values. However, this improvement in resource efficiency comes at the expense of user performance, in which the average App quality decreases by up to 10%. This confirms that, through the β weight in the objective, STEP can strike the desired trade-off between system efficiency and user experience.

In summary, STEP demonstrates robust orchestration efficiency across a variety of scenarios. In the case of simpler Apps compositions, it effectively balances latency with CPU, RB, and memory usage, outperforming baselines in overall resource efficiency. In the case of more complex Apps, STEP achieves the lowest deployment cost and consistently delivers high App quality with acceptable fairness, while maintaining competitive latency and success rates. These results confirm STEP's ability to strike an excellent balance between user experience and system resource utilization, validating its suitability for large-scale multi-user edge environments.

IX. CONCLUSIONS

We addressed the efficient management of composite Apps consisting of stateful and stateless MSs at the network edge. We formulated the MAP problem to minimize deployment costs while meeting performance constraints of the requested Apps and proved its NP-hardness. Our low complexity solution, STEP, exploits multi-layer DNTG to model deployment choices, and effectively increases the system performance by enabling MS shareability, CPU recalibration, and MS version adaptation. Evaluation on a small-scale scenario demonstrated STEP's near-optimal performance with only 7% higher CPU cost, reduced memory and CPU costs compared to the baseline approaches. Large-scale Kubernetes cluster experiments further validated STEP's efficiency, achieving up to 50% lower deployment costs than competing methods while delivering 50% higher the app quality and 15% lower radio resources.

An important real-world application of our framework is in edge-AI scenarios, where AI models are deployed as MSs at the edge and naturally exist in multiple versions, each offering different levels of inference accuracy and resource demands. The ability of STEP to dynamically select MS versions based on available resources and QoS requirements makes it directly applicable to such scenarios. Future research directions include extending STEP to support proactive migration based on predicted user mobility, anticipating handover events, and pre-deploying MSs before they need to be used.

REFERENCES

[1] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, J. He, and C. Xu, "An in-depth study of microservice call graph and runtime performance," *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[2] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, J. He, G. Yang, and C. Xu, "Erms: efficient resource management for shared microservices with sla guarantees," in *ACM ASPLOS*, 2023.

[3] M. R. Hossen, M. A. Islam, and K. Ahmed, "Practical efficient microservice autoscaling with qos assurance," in *ACM HPDC*, 2022.

[4] T. Bahreini and D. Grosu, "Efficient placement of multi-component applications in edge computing systems," in *ACM/IEEE SEC*, 2017.

[5] ——, "Efficient algorithms for multi-component application placement in mobile edge computing," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2550–2563, 2022.

[6] K. Ray, A. Banerjee, and N. C. Narendra, "Proactive microservice placement and migration for mobile edge computing," in *IEEE/ACM SEC*, 2020.

[7] ——, "Learning-based microservice placement and migration for multi-access edge computing," *IEEE Transactions on Network and Service Management*, 2024.

[8] Z. Chen, S. Huang, G. Min, Z. Ning, J. Li, and Y. Zhang, "Mobility-aware seamless service migration and resource allocation in multi-edge iov systems," *IEEE Transactions on Mobile Computing*, vol. 24, no. 7, pp. 6315–6332, 2025.

[9] L. Zeng, C. Zhang, Z. Wang, H. Du, and X. Jia, "Toward collaborative and latency-aware microservice migration in mobile edge computing," *IEEE Internet of Things Journal*, vol. 12, no. 13, pp. 25 286–25 299, 2025.

[10] Y. Liu, B. Yang, X. Ren, Q. Liu, S. Liu, and X. Guan, "e^2ms: An efficient and economical microservice migration strategy for smart manufacturing," *IEEE Transactions on Services Computing*, 2024.

[11] M. Alam, R. Matam, and F. A. Barbhuiya, "Edge-mi: Edge-based microservices for mobility-aware task migration scheme," in *IEEE Future Networks World Forum (FNWF)*, 2024, pp. 405–410.

[12] P. Bellavista, S. Dahdal, L. Foschini, D. Tazzioli, M. Tortonesi, and R. Venanzi, "Kubernetes enhanced stateful service migration for ml-driven applications in industry 4.0 scenarios," in *2024 IEEE Annual Congress on Artificial Intelligence of Things (AIoT)*, 2024, pp. 25–31.

[13] M. Adeppady, Y. Yu, A. Rahmanian, A. Ali-Eldin, and C. F. Chiasserini, "Efficient management of composite edge applications," in *IEEE GLOBECOM*, December 2025.

[14] Z. Tang, F. Mou, J. Lou, W. Jia, Y. Wu, and W. Zhao, "Multi-user layer-aware online container migration in edge-assisted vehicular networks," *IEEE/ACM Transactions on Networking*, vol. 32, no. 2, p. 1807–1822, Nov. 2023.

[15] B. Tang, F. Guo, B. Cao, M. Tang, and K. Li, "Cost-aware deployment of microservices for iot applications in mobile edge computing environment," *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 3119–3134, 2023.

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

[17] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 8, pp. 1825–1840, 2022.

[18] S. Velrajan and V. Ceronmani Sharmila, "Qos-aware service migration in multi-access edge compute using closed-loop adaptive particle swarm optimization algorithm," *Journal of Network and Systems Management*, vol. 31, no. 1, Dec. 2022.

[19] B. Tang, W. Xu, L. Zhang, B. Cao, M. Tang, and Q. Yang, "Location-aware dynamic scaling of microservices in mobile edge computing," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2025.

[20] J. Li, Q. Jiang, V. C. M. Leung, Z. Ma, and K. Kwarteng Abrokwa, "Deep-reinforcement-learning-based joint optimization of task migration and resource allocation for mobile-edge computing," *IEEE Internet of Things Journal*, vol. 12, no. 13, pp. 24 431–24 440, 2025.

[21] X. Zhao, Y. Shi, S. Chen, J. Liu, B. Ji, and S. Mumtaz, "Mapsm: Mobility-aware proactive service migration framework for mobile-edge computing in consumer internet of vehicles," *IEEE Transactions on Consumer Electronics*, vol. 71, no. 2, pp. 3753–3766, 2025.

[22] Y. Yin, P. Hou, S. Cui, J. Huang, J. Wang, Z. Lu, and P. C. K. Hung, "Mobility-aware assisted deep reinforcement learning for collaborative task migration and resource allocation in vehicular edge computing," *IEEE Transactions on Vehicular Technology*, pp. 1–14, 2026.

[23] Z. Wang, S. Zhu, J. Li, W. Jiang, K. K. Ramakrishnan, Y. Zheng, M. Yan, X. Zhang, and A. X. Liu, "Deepscaling: microservices autoscaling for stable cpu utilization in large scale cloud systems," in *SoCC*. ACM, 2022, p. 16–30.

[24] A. Calagna, Y. Yu, P. Giaccone, and C. F. Chiasserini, "Processing-aware Migration Model for Stateful Edge Microservices," in *IEEE ICC*, 2023.

[25] ——, "Design, modeling, and implementation of robust migration of stateful edge microservices," *IEEE Transactions on Network and Service Management*, 2023.

[26] D. Tazzioli, R. Venanzi, and L. Foschini, "Stateful service migration support for kubernetes-based orchestration in industry 4.0," in *IEEE ISCC*, 2024.

[27] T. Chanikaphon and M. A. Salehi, "Ums: Live migration of containerized services across autonomous computing systems," in *IEEE GLOBECOM*, 2023.

[28] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *ACM Middleware*, 2020.

[29] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *USENIX HotCloud*, 2019.

[30] N. Daw, U. Bellur, and P. Kulkarni, "Xanadu: Mitigating cascading cold starts in serverless function chain deployments," in *ACM Middleware*, 2020.

[31] B. Xiang, J. Elias, F. Martignon, and E. Di Nitto, "A dataset for mobile edge computing network topologies," *Data in Brief*, 2021.

[32] 3GPP, "User Equipment (UE) radio access capabilities," https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3193, 2017.

[33] M. R. S. Sedghpour, A. O. Duque, X. Cai, B. Skubic, E. Elmroth, C. Klein, and J. Tordsson, "Hydragen: A microservice benchmark generator," in *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. IEEE, 2023, pp. 189–200.

[34] F. A. Salaht, F. Desprez, and A. Lebre, "An overview of service placement problem in fog and edge computing," *ACM Computing Surveys (CSUR)*, 2020.

[35] C. Chekuri and S. Khanna, "On multidimensional packing problems," *SIAM Journal on Computing*, 2004.

[36] G. Xue, A. Sen, W. Zhang, J. Tang, and K. Thulasiraman, "Finding a path subject to many additive qos constraints," *IEEE/ACM Transactions on Networking*, 2007.

[37] J. Martin-Perez, F. Malandrino, C. F. Chiasserini, M. Groshev, and C. J. Bernardos, " KPI Guarantees in Network Slicing ," *IEEE/ACM Transactions on Networking*, vol. 30, no. 02, pp. 655–668, Apr. 2022.

APPENDIX A
RESPONSE LATENCY AND MIGRATION DOWNTIME EXPRESSIONS

The processing latency and communication latency experienced by user u requesting App $\mathcal{A}_j$ are given by:

$$d^{\text{proc}}_{u,\mathcal{A}_j} = \sum_{n\in\mathcal{N}}\sum_{q\in\mathcal{Q}_n}\sum_{i}\sum_{s>0} y^{n,q}_{s,i}\cdot z^{n,q}_{u,i}\cdot 1_{n\in\mathcal{A}_j}\cdot \frac{\sum_{u'\in\mathcal{U}}\sum_{\mathcal{A}_{j'}\in\mathcal{A}} z^{n,q}_{u',i}\cdot \rho^{u'}_{\mathcal{A}_{j'}}\cdot 1_{n\in\mathcal{A}_{j'}}\cdot \tau_{n,q}}{\hat{\tau}^{i}_{n,q}} \quad (6)$$

$$d^{\text{com}}_{u,\mathcal{A}_j} = \sum_{n\in\mathcal{N}}\sum_{q\in\mathcal{Q}_n}\sum_{i}\sum_{s>0} y^{n,q}_{s,i}\cdot z^{n,q}_{u,i}\cdot 1_{n=\mathcal{A}_j[0]}\Big\{\frac{\eta_{\mathcal{A}_j}\rho^{u}_{\mathcal{A}_j}}{B_{u,s}} + \sum_{\substack{m\in\mathcal{N}\\ m\neq n}}\sum_{q'\in\mathcal{Q}_m}\sum_{i'}\sum_{\substack{s'>0\\ s\neq s'}} y^{m,q'}_{s',i'}\cdot z^{m,q'}_{u,i'}\cdot 1_{m\in\mathcal{A}_j}\cdot 1_{e_{mn}>0}\cdot d_{s,s'}\Big\} \quad (7)$$

where, $d_{s,s'}$ is the delay of the link between s and s', i.e.,

$$d_{s,s'} = \sum_{u\in\mathcal{U}}\sum_{\mathcal{A}_j\in\mathcal{A}}\sum_{\substack{m,n\in\mathcal{N}\\ m\neq n}}\sum_{\substack{q\in\mathcal{Q}_n\\ q'\in\mathcal{Q}_m}}\sum_{i,i'} y^{n,q}_{s,i}\cdot z^{n,q}_{u,i}\cdot y^{m,q'}_{s',i'}\cdot z^{m,q'}_{u,i'} \cdot 1_{n\in\mathcal{A}_j}\cdot 1_{m\in\mathcal{A}_j}\cdot \frac{\rho^{u}_{\mathcal{A}_j}\cdot e_{mn}}{B_{s,s'}}. \quad (8)$$

Here, $B_{s,s'}$ represents Network bandwidth between two edge servers s, s' in bytes/s.

The downtime expressions for stateless relocation, stateful migration, and user state migration are respectively given by:

$$\delta^{\text{stateless}}_{u,A_j}(t) = \max_{\substack{n\in N,q\in Q_n\\ s>0,i}} d^{m,q}_{s}\cdot(1-a_n)\cdot (1-w^{n,q}_{s,i})\cdot y^{n,q}_{s,i}\cdot z^{n,q}_{u,i}\cdot 1_{n\in\mathcal{A}_j}. \quad (9)$$

$$\delta^{\text{stateful}}_{u,A_j}(t) = \max_{\substack{n\in N,q\in Q_n,i\\ s',s>0,s\neq s'}} d^{n,q}_{s',s}\cdot a_n\cdot y^{n,q}_{s,i}\cdot w^{n,q}_{s',i}\cdot z^{n,q}_{u,i}\cdot 1_{n\in\mathcal{A}_j}. \quad (10)$$

$$\delta^{\text{state}}_{u,\mathcal{A}_j} = \max_{\substack{n\in\mathcal{N},q\in\mathcal{Q}_n,\\ s',s>0,s\neq s',i,i',i\neq i'}} d^{n,q}\cdot a_n\cdot b_n\cdot x^{n,q}_{u,i'}\cdot w^{n,q}_{s',i}\cdot y^{n,q}_{s,i}\cdot z^{n,q}_{u,i}\cdot 1_{n\in\mathcal{A}_j}. \quad (11)$$

Finally overall App downtime is then:

$$D^{\text{down}}_{u,\mathcal{A}_j} = \max[\delta^{\text{stateless}}_{u,\mathcal{A}_j}, \delta^{\text{stateful}}_{u,\mathcal{A}_j}, \delta^{\text{state}}_{u,\mathcal{A}_j}]. \quad (12)$$

APPENDIX B
MATHEMATICAL FORMULATION OF MAP CONSTRAINTS

This appendix provides the full mathematical formulation of the constraints of the MAP problem introduced in Section V, organized into the three categories described therein.

App constraints: The constraints (13) and (14) ensure that the total delay and downtime of $\mathcal{A}_j$ for u are within their respective maximum tolerance level.

$$d^{\text{proc}}_{u,\mathcal{A}_j} + d^{\text{com}}_{u,\mathcal{A}_j} \leq l_{\mathcal{A}_j}, \forall u\in\mathcal{U}, \forall\mathcal{A}_j\in\mathcal{A}^u \quad (13)$$

$$D^{\text{down}}_{u,\mathcal{A}_j} < D_{\mathcal{A}_j}, \forall u\in\mathcal{U}, \forall\mathcal{A}_j\in\mathcal{A}. \quad (14)$$

MSs constraints: Every instance of MS n of quality q must be placed, as guaranteed by Constraint (15), while unused instances of the MSs must be on the dummy server, as ensured by Constraint (16). If MS n is shareable among the Apps requested by different users, at most one instance of n per server is possible (Constraint (17)). If user u is served by an instance of a stateful non-shareable MS, constraint (18) ensures that it is served by the same MS instance in the new deployment topology. Constraint (19) ensures that if an instance of stateful, non shareable MS is deployed, it must be used by a single user. Constraint (20) imposes that a user can access only one instance of a specific MS at any given time. Constraint (21) makes sure that the entry MSs of Apps requested by the same user must be deployed on the same server.

$$\sum_{s\in\mathcal{S}} y_{s,i}^{n,q}=1, \forall i, \forall n\in\mathcal{N}, \forall q\in\mathcal{Q}_n \tag{15}$$

$$y_{0,i}^{n,q}=1-\min(1,\sum_{u\in\mathcal{U}} z_{u,i}^{n,q}), \forall i, \forall n\in\mathcal{N}, \forall q\in\mathcal{Q}_n \tag{16}$$

$$b_n\cdot\sum_{i} y_{s,i}^{n,q}\leq 1, \forall s>0, \forall n\in\mathcal{N}, \forall q\in\mathcal{Q}_n \tag{17}$$

$$a_n(1-b_n)(z_{u,i}^{n,q}-x_{u,i}^{n,q})\geq 0, \forall u\in\mathcal{U}, \forall n\in\mathcal{N}, \forall q\in\mathcal{Q}_n, \forall i \tag{18}$$

$$\sum_{u\in\mathcal{U}} z_{u,i}^{n,q}\cdot(1-b_n)=\sum_{s>0} y_{s,i}^{n,q}\cdot(1-b_n), \forall i, \forall n\in\mathcal{N}, \forall q\in\mathcal{Q}_n \tag{19}$$

$$H\sum_{q\in\mathcal{Q}_n}\sum_{i} z_{u,i}^{n,q}\geq\sum_{\mathcal{A}_j\in\mathcal{A}} 1_{n\in\mathcal{A}_j}\, 1_{\mathcal{A}_j\in\mathcal{A}^u}, \forall u\in\mathcal{U}, \forall n\in\mathcal{N} \tag{20}$$

$$\sum_{m,n\in\mathcal{N}}\sum_{q\in\mathcal{Q}_n}\sum_{q'\in\mathcal{Q}_m}\sum_{i,i'} z_{u,i}^{n,q}\cdot z_{u,i'}^{m,q'}\cdot 1_{n=\mathcal{A}_j[0]}\cdot 1_{m=\mathcal{A}'_j[0]}\cdot$$
$$(y_{s,i}^{n,q}-y_{s,i'}^{m,q'})\leq 0, \forall s\in\mathcal{S}, \forall\mathcal{A}_j, \mathcal{A}'_j\in\mathcal{A}, \forall u\in\mathcal{U}. \tag{21}$$

System constraints: Constraints (22) and (23) ensure (resp.) that the total memory and CPU allocated to the MS instances of the requested Apps do not exceed the server's capability. Constraint (24) ensures that the sum of RBs allocated to all users connected to server s does not exceed the number of available RBs.

$$\sum_{n\in\mathcal{N}}\sum_{q\in\mathcal{Q}_n}\sum_{i} y_{s,i}^{n,q}\cdot\mu_{n,q}\leq M_s, \forall s>0, \tag{22}$$

$$\sum_{n\in\mathcal{N}}\sum_{q\in\mathcal{Q}_n}\sum_{i} y_{s,i}^{n,q}\cdot\hat{\tau}_{n,q}^{i}\leq C_s, \forall s>0 \tag{23}$$

$$\sum_{u\in\mathcal{U}} v_{u,s}\leq V_s, \forall s>0 \tag{24}$$

Madhura Adeppady is a post-doc at Politecnico di Torino, from which she received her Ph.D. degree in 2024.

Yenchia Yu is a Ph.D. student at Politecnico di Torino, from which he received his M.Sc. degree in 2022.

Ali Rahmanian is a post-doc at Chalmers University of Technology, Sweden.

Ahmed Ali-Eldin Hassan is an Associate Professor at Chalmers University of Technology, Sweden.

Carla Fabiana Chiasserini is a Full Professor with the Politecnico di Torino, Italy, a WASP Guest Professor with Chalmers University of Technology, Sweden, and a Research Associate with CNR and CNIT.