Efficient Management of Composite Edge Applications

Efficient Management of Composite Edge Applications

Madhura Adeppady[†], Yenchia Yu[†], Ali Rahmanian[‡], Ahmed Ali-Eldin Hassan[‡], Carla Fabiana Chiasserini[† ‡]
[†] Politecnico di Torino, Italy [‡] Chalmers University of Technology, Sweden

***Abstract*—Edge computing reduces latency for mobile applications (Apps) by processing data closer to users, while containerized microservices (MSs) enable their modular deployment. Managing such Apps involves three key challenges: (i) strategically placing MSs to minimize response latency and resource consumption, (ii) managing MS migration/relocation during user mobility or traffic load changes while limiting App downtime, and (iii) enabling MS sharing across Apps while ensuring target performance. We formulate this as an optimization problem (proven to be NP-hard) and propose STEP, a polynomial-time heuristic. In contrast to prior art, STEP (i) jointly considers stateful and stateless MSs in its decisions, (ii) exploits MS shareability to reduce resource usage, (iii) balances response latency, App downtime, and resource utilization, and (iv) leverages multiple versions of the same MS to adapt QoS to available edge resources. Results show that STEP achieves near-optimal performance with only 1.6% higher deployment cost while reducing CPU usage by 42% compared to baselines. Also, it enables real-time App deployment in a large-scale scenario on a Kubernetes cluster with sub-second order execution time and reduced deployment cost by 16–17% compared to its benchmarks.**

***Index Terms*—Service orchestration and management, edge computing, resource allocation**

I. Introduction

Edge computing is a promising paradigm for deploying latency-sensitive applications (Apps) as it brings resources closer to end users. Here, servers are placed near cellular base stations (gNBs), enabling mobile devices to offload computation, greatly reducing response latency and bandwidth usage. Edge Apps often adopt a microservice (MS) architecture, breaking monolithic Apps into lightweight, containerized MSs that can be dynamically and independently deployed.

MSs can be stateful, retaining user session data across requests, or stateless, processing requests independently without storing user data [1]. MS architecture enables shareability, allowing MS reuse across Apps for 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 their state includes user-specific information. For example, a registry MS with generic uncrewed aerial vehicle (UAV) registration data is shareable across UAVs and Apps, but an autopilot MS with UAV-specific flight parameters is not. To ensure flexibility, an MS can have multiple versions, each providing distinct Quality of Service (QoS) levels, complexity, and resource demands.

This work was supported by the innovation programme under the 6G-INTENSE project (Grant Agreement No. 101139266). Y. Yu's work was supported by Leonardo S.p.A.

Motivation. Despite its benefits, an edge MS architecture introduces challenges in ensuring performance guarantees with minimal resource consumption. *(i)* Managing resources for Multi-MS App is complex, requiring the orchestrator to allocate computational speed, manage Resource Blocks (RBs), and optimize MS placement to minimize costs while meeting target latency requirements. *(ii)* User mobility triggers MS redeployments, disrupting service. Stateful MSs require entire container migration with user session data, while for stateless MSs, container relocation is sufficient. The orchestrator must ensure App downtime is within acceptable limits. *(iii)* MS shareability complicates orchestration, as shared MSs impact multiple Apps and users, requiring dynamic updates to computing resources when Apps start/stop using shared MSs. These challenges need holistic MS deployment strategies balancing response latency, App downtime, and resource usage.

Limitation of state-of-art approaches. Recent approaches address resource management in ensuring App response latency [2]–[4]. Notably, [2], [3] focus on analyzing individual MS contributions to latency and applying autoscaling, or determining MS-to-server allocation [4] to reduce costs while meeting target latency. However, these approaches fall short in edge environments. *(i)* They ignore MS migration/relocation caused by user mobility. *(ii)* They lack the flexibility of adapting MS versions based on resource availability, which is crucial for resource-constrained edge nodes. *(iii)* Few exploit MS shareability to reduce resource consumption. Other edge-focused approaches pre-deploy MSs based on predicted user mobility [5]–[7]. These works *(i)* consider Apps composed of only stateful MSs, *(ii)* use abstract migration models that do not capture real MS migration complexities, and *(iii)* skip MS migration when destination servers lack resources.

To fill these gaps, we propose State and Topology-aware Edge-MS Placement (STEP), an algorithm for orchestrators to make cost-efficient deployment and resource allocation decisions while ensuring acceptable response latency and App downtime. STEP constructs a Dynamic Network Topology Graph (DNTG) representing the key system components and capturing deployment choices. Also, it identifies dynamic conditions at the edge, such as App deployment requests, user handovers, and App terminations. Upon such events, STEP builds a decision graph encoding feasible deployment and resource allocation choices to meet App requirements. This graph is expanded to enforce additive constraints and ensure decision feasibility. Finally, deployment decisions are made by finding the minimum-cost path in the expanded graph.

We summarize our key contributions as follows:

- We build a system model that captures all key aspects of

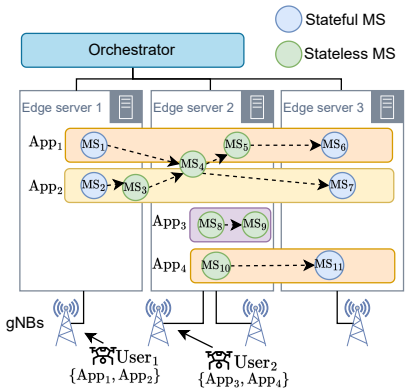

Fig. 1. Example target scenario: Apps are deployed at the edge as chains of stateful/stateless MSs. Mobile users (represented as UAVs) access the Apps via 5G gNBs, with User$_1$ (User$_2$) requesting App$_1$–App$_2$ (App$_3$–App$_4$).

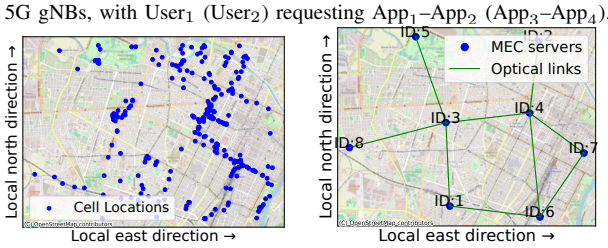

Fig. 2. Open radio cell location (left) and edge server topology (right).

edge environments and composite Apps.

- We formulate the Multi-microservice Application Placement (MAP) problem, which aims to minimize deployment and relocation costs while meeting all requirements. Owing to MAP's NP-hardness, we propose the polynomial-time STEP algorithm which differs from prior art as: *(i)* it considers Apps with stateful and stateless MSs; *(ii)* it addresses multiple performance metrics, such as response latency, App downtime, and resource usage; *(iii)* its has low complexity; *(iv)* it reuses deployed MS instances for new or migrating Apps; *(v)* it increases flexibility by considering different versions of the same App that may offer different QoS and resource consumption.
- Extensive performance analysis shows that STEP closely matches the optimal solution, while achieving sub-second scale execution time in a large-scale real-world cluster. Further, it reduces deployment costs by 16–17% compared to baselines.

II. Target Scenario and Testbed

We focus on managing Function-as-a-Service (FaaS) services at the network edge. Our scenario (see Fig. 1) features an edge computing system with an orchestrator and edge servers (connected via physical or virtual links). These servers host Apps, each comprising multiple containerized MSs. Each server may be connected with one or more gNBs, enabling mobile users to access edge Apps. As users move and switch gNBs, the orchestrator may migrate/relocate MSs of the requested Apps to keep latency sufficiently low. Similarly, when App load varies, the resources allocated to its MS instances may be insufficient or excessive, and the orchestrator may need to move MSs to optimize resource allocation.

We recreated the above system architecture in our kubernetes-based testbed (https://kubernetes.io/), with 8 virtual machines (VMs) (4 CPU cores @3.6 GHz, 4 GB RAM each) as edge servers. The orchestrator, running on the host server, manages real-time orchestration. Edge server topologies were designed following [8], using open radio cell data from `OpenCelliD` (https://www.opencellid.org/) (Fig. 2 (left)). The radio cells are clustered into 8 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 10 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 gNB that is co-located with the edge server, and the bandwidth between the mobile user and the gNB is estimated based on the 3GPP TS 38.306 [9] with the channel quality indicator (CQI) being negatively correlated with the user-gNB distance.

TABLE I
List of Symbols

Symbol	Description
	Parameters for Servers
$\mathcal{S}$	Set of servers including dummy server s_0 hosting all potential MS instances
M_s, C_s	Memory (B)&computation (CPU cycles/s) of server s
$B_{s,s'}$	Bandwidth between two edge servers s, s' (B/s)
V_s	Max. no. of RBs at gNB connected with server s
	Parameters for Users
$\mathcal{U}$	Set of available users
$B_{u,s}$	Allocated bandwidth between user u&server s (B/s)
	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 $\mathcal{A}_j$ (B)
$D_{\mathcal{A}_j}$	Max. tolerable downtime for App $\mathcal{A}_j$ (s)
	Parameters for MSs
$\mathcal{N}, \mathcal{Q}_n$	Set of available MSs and possible versions of MS n
a_n, b_n	Set to 1 if MS n is stateful (a_n) or shareable (b_n)
$\mu_{n,q}$	Required memory of MS n of version q (B)
$\tau_{n,q}$	Required comp. of MS n of version q (CPU cycles)
e_{mn}	Avg. message size between MS m and MS n (B)
$w^{n,q}_{s,i}$	1 if MS n, version q, instance i is on server s, else 0
$x^{n,q}_{u,i}$	1 if user u uses MS n, version q, instance i, else 0
	Decision Variables
$y^{n,q}_{s,i}$	Binary; 1 if MS (n,q,i) is on server s; else 0
$z^{n,q}_{u,i}$	Binary, 1 if user u is served by MS (n,q,i); else 0
$\hat{\tau}^i_{n,q}$	Continuous, CPU cycles/s allocated to MS (n,q,i)
$v_{u,s}$	Integer, allocated RBs between user u and server s

III. System Model and Time Performance Metrics

We now introduce our key assumptions and the system model parameters and variables (see Table I). Then we provide analytical expressions for the main performance metrics.

Assumptions. Without loss of generality, we consider that:

• *Edge server and gNBs:* Each gNB 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 gNB.
• *MS deployment:* Entry MSs of requested Apps are placed on the edge server co-located with the user's serving gNB.
• *MS migration/relocation:* The orchestrator may update the MS deployment upon user handover, App request, or termination; it then remains unchanged until the next such event.
• *MS version:* MS instances implementing different versions of the MS have the same communication interfaces. During migrations, stateless MSs may switch versions based on available resources at the destination, while stateful MSs *must* retain their versions due to user-specific states.
• *App termination:* App termination occurs when all tasks are completed or operational conditions necessitate termination, such as when a UAV finishes its tasks or returns to its base, triggering an event at the orchestrator.

Time Performance Metrics. The *response latency* for user u requesting App $\mathcal{A}_j$ comprises two parts: (i) the processing latency of App's MSs ($d^{\text{proc}}_{u,\mathcal{A}_j}$) and (ii) the communication latency between successive MSs, and from user u to App's entry MS ($d^{\text{comm}}_{u,\mathcal{A}_j}$). The two components are:

$$d^{\text{proc}}_{u,\mathcal{A}_j} = \sum_{n,q,i,s>0} y^{n,q}_{s,i} z^{n,q}_{u,i} \mathbb{1}_{n\in\mathcal{A}_j} \frac{\sum_{u',\mathcal{A}_{j'}} z^{n,q}_{u',i} \rho^{u'}_{\mathcal{A}_{j'}} \mathbb{1}_{n\in\mathcal{A}_{j'}} \tau_{n,q}}{\hat{\tau}^i_{n,q}}$$

$$d^{\text{comm}}_{u,\mathcal{A}_j} = \sum_{\substack{n,q,\\ i,s>0}} z^{n,q}_{u,i} \mathbb{1}_{n=\mathcal{A}_j[0]} \Big\{ \frac{\eta_{\mathcal{A}_j} \rho^u_{\mathcal{A}_j}}{B_{u,s}} + \sum_{\substack{m\neq n,q'\\ i',s'\neq s}} y^{m,q'}_{s',i'} z^{m,q'}_{u,i'} \mathbb{1}_{m\in\mathcal{A}_j} \mathbb{1}_{e_{mn}>0} d_{s,s'} \Big\}$$

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

$$d_{s,s'} = \sum_{\substack{u,\mathcal{A}_j,q,q'\\ m,n,m\neq n,i,i'}} y^{n,q}_{s,i} z^{n,q}_{u,i} y^{m,q'}_{s',i'} z^{m,q'}_{u,i'} \mathbb{1}_{n\in\mathcal{A}_j} \mathbb{1}_{m\in\mathcal{A}_j} \frac{\rho^u_{\mathcal{A}_j} e_{mn}}{B_{s,s'}}.$$

Next, we derive the expression of an *App downtime* due to its MSs' migration/relocation, which can be generalized to container migration/relocation. According to MS characteristics and shareability, three procedures are used:
• **Stateless relocation**: (1) create a new instance of the stateless MS container at the destination host, (2) redirect the requests to the new instance, and (3) shut down the old instance at the source host.
• **Stateful migration**: (1) checkpoint the running container at the source host, thus collecting the state of MS and the established connection; (2) clear network namespace, thus preventing network configuration conflicts in the following steps; (3) transfer the checkpoint image to the destination host; (4) re-create and configure 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 checkpoint.
• **User state migration**: It happens (i) when multiple users share MS container at the source host, or (ii) when a container of the same MS type is running at the destination host. Only the user's state is transferred from the source to the destination. It includes (1) extracting the user's active state from the container at the source; (2) transferring this to a running container of the same MS type at the destination; (3) redirecting the user's request to the container at the destination.

Since an App may include stateful and stateless MSs, we must guarantee that the *App downtime* $D^{\text{down}}_{u,\mathcal{A}_j}$, including downtime for stateless relocations, stateful migrations, and user state migrations of all its MSs, remains within the tolerance limit. For simplicity and without loss of generality, we consider that these three procedures can be parallelized; however, for each App and type of procedure, MS relocations/migrations are sequentially performed according to the MS order in the chain. The model can be easily modified to include other strategies.

For stateless MS containers, downtime occurs mainly during initiation, depending on MS container size and destination host characteristics. The stateless downtime of MS n at quality q relocated to server s is $d^{n,q}_s = f_{\text{stateless}}(\mu_{n,q}, s)$, where $f_{\text{stateless}}$ estimates initiation time. The total downtime of the stateless MSs in an App is: $\delta^{\text{stateless}}_{u,\mathcal{A}_j} = \sum_{n\in N}\sum_{q\in Q_n}\sum_{s>0}\sum_{i>0} d^{n,q}_s \cdot (1-a_n)\cdot[1-w^{n,q}_{s,i}]\cdot y^{n,q}_{s,i}\cdot z^{n,q}_{u,i}\cdot \mathbb{1}_{n\in\mathcal{A}_j}$. The migration downtime of stateful MS containers follows Processing-Aware Migration (PAM) model [10]. For stateful MS instances migrated from s_0 to $s>0$, the MS downtime is $d^{n,q}_{0,s} = d^{n,q}_s$, as MS instances on s_0 do not retain user/session data. Thus, we define stateful downtime of 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} estimates stateful migration downtime in PAM. The total downtime of stateful MSs of an App is: $\delta^{\text{stateful}}_{u,\mathcal{A}_j} = \sum_{n\in N}\sum_{q\in Q_n}\sum_{s',s>0 s\neq s'}\sum_i d^{n,q}_{s',s}\cdot a_n\cdot y^{n,q}_{s,i}\cdot w^{n,q}_{s',i}\cdot z^{n,q}_{u,i}\cdot \mathbb{1}_{n\in\mathcal{A}_j}$. The downtime of MS n of quality q for user u for state migration is defined as $d^{n,q} = f_{\text{state}}(u)$. The total downtime is then $\delta^{\text{state}}_{u,\mathcal{A}_j} = \sum_{n\in\mathcal{N}}\sum_{q\in\mathcal{Q}_n}\sum_{s',s>0,s\neq s'}\sum_{i,i\neq i'} d^{n,q}\cdot a_n\cdot b_n\cdot x^{n,q}_{u,i'}\cdot w^{n,q}_{s',i}\cdot y^{n,q}_{s,i}\cdot z^{n,q}_{u,i}\cdot \mathbb{1}_{n\in\mathcal{A}_j}$. Finally, assuming that these three procedures can be parallelized, the overall downtime for an App is given by: $D^{\text{down}}_{u,\mathcal{A}_j} = \max[\delta^{\text{stateless}}_{u,\mathcal{A}_j}, \delta^{\text{stateful}}_{u,\mathcal{A}_j}, \delta^{\text{state}}_{u,\mathcal{A}_j}]$.

IV. The MAP Problem

The MAP problem must be solved by the orchestrator when any of three events occurs: *(i)* a user requests a new App; *(ii)* a user stops using an App; *(iii)* a user connection is handed over between gNBs.

The App deployment cost includes resource allocation and communication cost. The former refers to the cost of allocating computation to the MSs of the requested Apps, while the latter covers the communication cost between the user and the entry MS server, and between servers when successive MSs are not co-located. Let MCS_u be the modulation and coding scheme used by user u, then the deployment cost is:

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

The QoS offered by App $\mathcal{A}_j$ to user u is defined as the average normalized qualities of its MS instances, and the average QoS across all Apps and users is denoted by $Q(z)$.

The orchestrator's objective is to minimize the deployment cost C and to maximize the Apps' QoS $Q(z)$, thus we can formulate the MAP problem as in the colored box below.

Multi-MS Application Placement (MAP) Problem

$$\min_{\boldsymbol{yz\hat{\tau}v}} (C(y,z,\hat{\tau},v) - Q(z)) \quad (2a)$$

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

Problem Complexity. The MAP problem is NP-hard.

Proof. By reduction from NP-hard multi-dimensional bin packing problem [11] to MAP in polynomial time.

V. The STEP Algorithm

Consistently with the MAP problem definition, STEP runs at the orchestrator whenever a request for a new App is received, an App is terminated, or a user handover occurs. It includes three main steps, as detailed below.

DNTG Construction: This is a 4-layer graph representing key system components and their interactions (Fig. 3). The user layer (bottom-most) has one vertex per user and the infrastructure layer (bottom) has one vertex per server. The service layer (top) captures possible MS deployments where each MS instance features a certain quality version. Finally, Apps are represented by vertices in the top-most layer.

The DNTG's purpose 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 (user connection handovers) and between users and Apps (App request arrival or termination). The time interval between any two successive contact events in the network is a *frame*. Within a frame, MS deployments and links remain unchanged. Active contact events are those ongoing within each frame. Each user u participating at frame k is represented by vertex u^k in the user layer, while each edge server s is mapped onto vertex s^k in the infrastructure layer. Vertex (n^k, q_n^k, i^k) represents the instance i of MS n of quality q_n at frame k, and App $\mathcal{A}_j$ is mapped onto vertex $\mathcal{A}_j^k$. At each frame k, the DNTG contains the following directed edges connecting vertices corresponding to users, servers, MSs, or Apps (i.e., $u^k \in \mathcal{U}^k$, $s^k \in \mathcal{S}^k$, $(n^k, q_n^k, i^k) \in \mathcal{D}^k$, $\mathcal{A}_j^k \in \mathcal{A}^k$):

- $(u^k, \mathcal{A}_j^k)$ with weight $(\eta_{\mathcal{A}_j} \rho^u_{\mathcal{A}_j})$, exists if u has requested $\mathcal{A}_j$;
- (u^k, s^k) with weight $V_s \text{MCS}_u$ exists if u is connected to s;
- $(\mathcal{A}_j^k, (n^k, q_n^k, i^k))$ exists if MS n is the entry MS of App $\mathcal{A}_j^k$; its the weight is set to $(\eta_{\mathcal{A}_j}, \rho^u_{\mathcal{A}_j})$;
- $((n^k, q_n^k, i^k), (m^k, q_m^k, j^k))$ with weight e_{mn} exists if MS n and MS m are in the same App and n communicates with m;
- $(s^k, \hat{s}^k)$ exists if there exists a wired network connection between servers s_k and $\hat{s}_k$; its weight is set to $B_{s,\hat{s}}$.

Also, the following directed edges connect the vertices representing the same node (server, MS instance, App) across consecutive frames: (s^k, s^{k+1}) (weight: CPU, memory, and RBs remaining at s); $((n^k, q_n^k, i^k), (n^{k+1}, q_n^{k+1}, i^{k+1}))$ (weight: MS container properties such as dirty page rate); $(\mathcal{A}_j^k, \mathcal{A}_j^{k+1})$ (weight: deployment cost and QoS of the App).

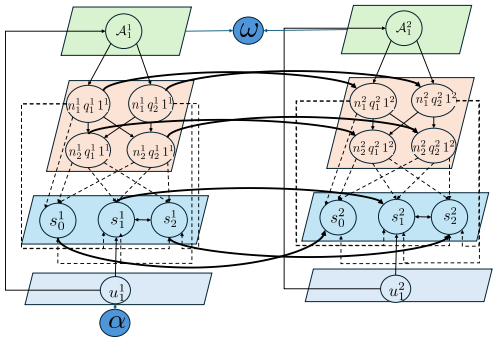

Fig. 3. A sample DNTG, consisting of 1 user (bottom-most), 2 servers (bottom), 2 MSs (top), and 1 App (top-most).

Next, we use the DNTG to formulate a min-cost problem as a low-complexity solution to MAP. To this end, we add virtual source α and destination ω vertices to the DNTG, and then connect α to all user vertices in $\mathcal{U}^1$ and all App vertices in $\mathcal{A}_j^k$ (where $k \geq 1$) $to \omega$ using zero-weight edges.

Decision Graph. For each DNTG frame, we construct a simplified decision graph $G=(V,E)$, including only the vertices relevant to deployment decisions in that frame.

We first identify changes in the current frame compared to the previous one, with G varying based on the event type. For new App requests, G includes the user, the requested App, its MSs, and servers with sufficient resources. Instead, for user migration or App termination, 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 needs CPU reallocation. We simplify the decision graph by merging the service and infrastructure layers into a deployment layer, where each vertex (n, q_n, i, s) represents instance i of MS n of quality q_n deployed on server s. Further, for each vertex in this layer, we create replicas corresponding to predefined CPU allocations (e.g., $\{0.05\text{G}, 0.1\text{G}, \ldots\}$ cycles/s) and RB allocations (e.g., $\{1, \ldots\}$), resulting in nodes 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, and the associated deployment cost. Since some decisions in G may violate MS shareability and entry MS placement constraints, we ensure feasibility by pruning G as follows: *(i)* Since entry MSs of Apps requested by a user must be placed on the same server (preferably the one to which the user 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 $v_{u,s}=0$; *(ii)* If a server has a shareable MS instance of a given quality, additional instances of the same MS configuration are ignored to allow reusability.

Also, every edge (v, v') in E has the following properties:

- *Response latency $D(v,v')$:* It represents the response latency contribution if the edge (v, v') is included in the deployment. By default, response latency is an additive constraint, i.e., summing these latency attributes of all the edges in the deployment yields the App response latency. If v is an App layer vertex

and v' is a deployment layer vertex, then $D(v, v')$ is the sum of communication latency from the user to the App entry MS (based on v''s RB allocation) and processing latency of v' (based on its CPU allocation). If both v and v' are deployment layer vertices, $D(v, v')$ is the sum of communication latency if their servers differ and processing latency of v'.

• *Migration downtime* $M(v, v')$*:* It represents the contribution of edge (v, v') to the App migration downtime when foreseen in the deployment. It is not directly an additive property, since the three types of migration/relocation procedures can occur simultaneously. However, procedures of the same type occur sequentially based on the MS order in the chain. We convert $M(v, v')$ to an additive constraint by tracking the cumulative time for each procedure type as we traverse the graph, starting from the App vertex. Then, for any generic edge (v, v') for u requesting $\mathcal{A}_j$, we assign a multi-dimensional weight defined as: $w(v, v') = (D(v, v')/l_{\mathcal{A}_j}, M(v, v')/D_{\mathcal{A}_j})$.

Expanded Graph. Given G, we now find feasible deployments that meet the end-to-end constraints. This procedure is repeated for each App requested by a user for user handover or App termination events. Since finding a path between a source and destination vertex satisfying end-to-end constraints is NP-hard [12], we construct an expanded graph [12].

Given a positive integer resolution parameter γ:

• For each vertex in the deployment and App layers of G, 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 in G become embedded into the vertices of the expanded graph, ensuring that any path from α to ω honors considered additive constraints. Then, to find the min-cost path, we associate each edge (v, v') from App to the deployment layer, or within the deployment layer, with the cost of deploying the MS in v'. All other edges in the expanded graph have 0 weight. To enable MS sharing, a 0 weight is assigned to (v, v'), if the MS in v' is already deployed. Then we apply Dijkstra's algorithm and find the minimum cost path.

CPU recalibration. Since the initially used predefined set of CPU allocations may not be optimal, we solve a problem with the same objective as MAP but considering only the response latency constraint. As all decision variables other than CPU allocations are fixed from the minimum-cost paths, the problem becomes convex and solvable in polynomial time.

VI. Performance Evaluation

We first compare the optimum against STEP and baseline algorithms in a small-scale scenario. We then assess the performance of STEP and baselines in a large-scale scenario **by implementing these schemes in our testbed and measuring the relevant performance metrics**. We consider two

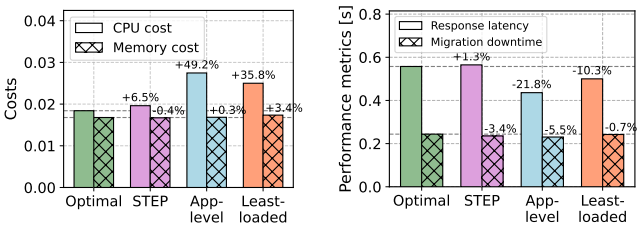

Fig. 4. Optimum vs. STEP and baselines: CPU & memory costs (left), response latency & App downtime (right).

baselines [13]: *(i) App-level closest (App-level):* it deploys MSs of the same App on the same server, selecting the closest one to the user requesting the App; *(ii) MS-level closest-least-loaded (least-loaded):* it deploys the MSs on the least loaded server; in the case of ties, it selects the one closest to the user.

Small-scale scenario. It includes 4 MSs: 2 stateless MSs, 1 stateful shareable MS, and 1 stateful non-shareable MS, forming 4 Apps: App_1 (MS1→MS2), App_2 (MS1→MS3), App_3 (MS2→MS3), and App_4 (MS2→MS4). Also, it includes 4 servers. App requests follow a Poisson process. To evaluate our system under a high migration rate and a high load, we set the App request rate to 2 per unit time and the normalized load per App to 0.2, achieving 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. For simplicity, Q_n={1} for each MS n, though multiple instances of the same MS are allowed. We set the target response latency of each App to 1 s, and the maximum tolerable App downtime to 1 s. Other parameters are configured based on the testbed measurement. In baseline solutions, CPU and RB allocations use predefined levels, ensuring response latency stays below target value. The optimum is obtained by solving the MAP problem in Gurobi.

During each event, we measure various performance metrics and annotate percentage improvement or degradation compared to the optimum in Fig. 4. The deployment cost for different approaches is shown in Fig. 4(left). Remarkably, STEP closely matches optimal CPU cost and substantially outperforms benchmarks, thanks to its effective CPU recalibration. While all schemes have similar memory costs, STEP's memory cost is even lower than the optimum, because it enables Apps to share common MSs, requiring fewer instances to serve requests. All approaches have lower App downtime than the optimum (Fig. 4 (right)). Since minimizing App downtime is not the optimal's goal, other methods can indeed perform better here. Finally, Fig. 4 (right) shows both *App-level* and *least-loaded* deployments achieve lower response latency, but at higher CPU cost without real advantage, as all schemes meet the Apps latency requirement of 1 s.

Large-scale scenario. It includes 10 MSs: 4 stateless MSs (MS1, MS5–MS7), 3 stateful shareable MSs (MS2–MS4), and 3 stateful non-shareable MSs (MS8–MS10). MS6 and MS8 have two quality versions each. These MSs form 6 Apps. Using our testbed, we emulate 10 users (UAVs) moving along pre-defined paths and requesting 1 App/s. As users move, MCS on user-gNB links varies with distance, and users connect to gNBs offering best connectivity. Upon a handover, STEP is executed at the Kubernetes orchestrator triggering MS

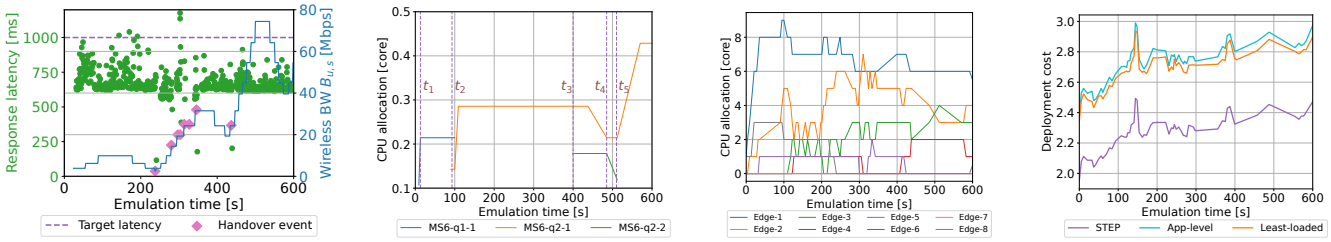

Fig. 5. Time evolution of the response latency for the App requested by User_6 as it moves across different gNBs (left most), CPU allocated to different instances of MS6 with different quality (left), no. of MS instances at each edge server (right), and MS deployment cost of STEP and its baselines (right most).

migration/relocation as needed. During the 600 s experiment, an average of 60 handovers were triggered causing MS re-deployments; on average, STEP was executed additional 10 times for new App requests. All Apps have 1 s target response latency is 1 s and 2 s maximum tolerable downtime. Other parameters are configured based on the testbed measurements.

Remarkably, STEP execution time is always in the sub-second range (min: 2 ms, avg: 196 ms, max: 581 ms), enabling real-time MS deployment updates. The number of MSs migrated/relocated per execution ranges from 0–2, with an average of 1.32, much smaller than typical MS chain, underscoring STEP's orchestration efficiency. Consequently, App downtime stays well within acceptable limits (0.25-0.5 s, avg: 0.75 s).

Fig. 5 (left most) shows the response latency experienced by User_6, and the corresponding allocated bandwidth, $B_{u,s}$, as time evolves. STEP maintains the App response latency below the target maximum, except during rare critical events (e.g., extremely poor connection quality or a handover) occur. Interestingly, as the user moves, it can connect to gNBs offering a better link quality (i.e., higher MCS) and a higher number of RBs, the orchestrator can reduce the allocated CPU (10% reduction in our experimental measurements) while still keeping the response latency sufficiently low.

Next, Fig. 5 (left) shows CPU allocation across different quality instances of an example MS (MS6). Notably, User_3 and User_4 request MS6 (resp.) via App_3 and App_4. Initially, User_3 requests App_3 and an instance of MS6 with quality level 1 (MS6-q1-1, for short) is created. At t_1, when User_4 requests App_4, STEP makes App_4 share MS6-q1-1 instance with App_3, scaling its CPU allocation from 0.1 to 0.2 cores. At t_2, User_3 moves to a different gNB and a new instance with quality 2, MS6-q2-1, is deployed, as the new server's higher computational capacity allows STEP to select a higher version. Shortly after, User_4 also connects to this gNB and shares MS6-q2-1, increasing its CPU allocation. At t_3, User_3 moves again and by t_4 a new instance MS6-q2-2 is created while MS6-q2-1's CPU allocation is reduced from 0.3 to 0.2 cores. Finally, at t_5 User_3 returns to its previous gNB, removing MS6-q2-2, and increasing MS6-q2-1's CPU allocation again. Demonstrating STEP's flexibility, our experiment showed that on average 35% of shareable MSs were shared with sharing reaching upto 70%.

Fig. 5 (right) depicts the time evolution of MSs hosted on each edge server, with IDs corresponding to Fig. 2 (right). Despite Edge-1 and Edge-2 hosting more MSs due to higher user density in their coverage areas, STEP exhibits significant robustness to such uneven user distribution. Indeed, the allocated CPU is uniform across all servers, thus guaranteeing load balancing. At last, Fig. 5 (right-most) shows the time evolution of the Apps deployment costs ((2a)), showing STEP's cost is on average 17% and 16% lower than App-level and Least-loaded (resp.) baselines, consistent with the small-scale results.

VII. Conclusions

We tackled the management of composite edge Apps with stateful and stateless MSs by formulating the NP-hard problem to minimize deployment costs. Our low complexity solution, STEP, exploits a multi-layer DNTG to model deployment choices and builds an expanded graph that meets all constraints. STEP matches the optimum and reduces CPU usage by 42% against the baselines. Our testbed validated STEP's effectiveness, achieving sub-second execution time and a 16–17% reduction in deployment costs against the baselines.

References

[1] S. Luo *et al.*, "An in-depth study of microservice call graph and runtime performance," *IEEE TPDS*, 2022.

[2] S. Luo *et al.*, "Erms: efficient resource management for shared microservices with sla guarantees," in *ACM ASPLOS*, 2023.

[3] M. R. Hossen *et al.*, "Practical efficient microservice autoscaling with qos assurance," in *ACM HPDC*, 2022.

[4] T. Bahreini *et al.*, "Efficient algorithms for multi-component application placement in mobile edge computing," *IEEE TCC*, 2020.

[5] K. Ray *et al.*, "Proactive microservice placement and migration for mobile edge computing," in *IEEE/ACM SEC*, 2020.

[6] Z. Chen *et al.*, "Mobility-aware seamless service migration and resource allocation in multi-edge iov systems," *IEEE TMC*, 2025.

[7] Y. Liu *et al.*, "E^2MS: An efficient and economical microservice migration strategy for smart manufacturing," *IEEE TSC*, 2024.

[8] B. Xiang *et al.*, "A dataset for mobile edge computing network topologies," *Data in Brief*, 2021.

[9] 3GPP, "User equipment (ue) radio access capabilities." https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3193, 2017.

[10] A. Calagna *et al.*, "Processing-aware Migration Model for Stateful Edge Microservices," in *IEEE ICC*, 2023.

[11] C. Chekuri *et al.*, "On multidimensional packing problems," *SIAM Journal on Computing*, 2004.

[12] G. Xue *et al.*, "Finding a path subject to many additive qos constraints," *IEEE/ACM TON*, 2007.

[13] F. A. Salaht *et al.*, "An overview of service placement problem in fog and edge computing," *ACM CSUR*, 2020.