Heuristic and Reinforcement Learning Algorithms for Dynamic Service Placement on Mobile Edge Cloud

Dhruv Garg\textsuperscript{*}

Georgia Institute of Technology
Atlanta, USA
dgarg39@gatech.edu

Nanjangud Narendra

Ericsson Research
Bangalore, India
nanjangud.narendra@ericsson.com

Selome Tesfatsion

Ericsson Research
Stockholm, Sweden
selome.kostentinos.tesfatsion@ericsson.com

ABSTRACT

Edge computing hosts applications close to the end users and enables low-latency real-time applications. Modern applications in turn have adopted the microservices architecture which composes applications as loosely coupled smaller components, or services. This complements edge computing infrastructure that are often resource constrained and may not handle monolithic applications. Instead, edge servers can independently deploy application service components, although at the cost of communication overheads. Dynamic system load in mobile network cause like latency, jitter, and packet loss to fluctuate frequently. Consistently meeting application service level objectives while also optimizing application deployment (placement and migration of services) cost and communication overheads in mobile edge cloud environment is non-trivial. In this paper we propose and evaluate three dynamic placement strategies, two heuristic (greedy approximation based on set cover, and integer programming based optimization) and one learning-based algorithm. Their goal is to satisfy the application constraints, minimize infrastructure deployment cost, while ensuring availability of services to all clients and User Equipment (UE) in the network coverage area. The algorithms can be extended to any network topology and microservice based edge computing applications. For the experiments, we use the drone swarm navigation as a representative application for edge computing use cases. Since access to real-world physical testbed for such application is difficult, we demonstrate the efficacy of our algorithms as a simulation. We also contrast these algorithms with respect to placement quality, utilization of clusters, and level of determinism. Our evaluation not only shows that the learning-based algorithm provides solutions of better quality, it also provides interesting conclusions regarding when the (more traditional) heuristic algorithms are actually better suited.

KEYWORDS

Edge computing, low-latency applications, service placement, reinforcement learning, heuristic algorithms

ACM Reference Format:

Dhruv Garg, Nanjangud Narendra, and Selome Tesfatsion. 2021. Heuristic and Reinforcement Learning Algorithms for Dynamic Service Placement on Mobile Edge Cloud. In Proceedings of ACM Conference (Conference'17). ACM, New York, NY, USA, 13 pages. https://doi.org/10.XXXX/XXXXXXX.XXXXXXX

1 INTRODUCTION

The area of cloud computing is being revolutionized by the shift towards the edge. This is also enabled by the shift of traditional monolithic cloud applications into chains of microservices (interchangeably referred to as services in this paper), which can be independently deployed to run on different edge servers. While edge computing significantly reduces the round-trip time (latency) and jitter (variation in latency), SLO attainment can vary based on dynamic demand and network conditions. The application control logic monitors the system behavior and dynamically scales up or down the services to provide optimum performance. Infrastructure deployment cost however raises the key issue of optimally placing microservice chains.

The issue becomes especially acute for applications deployed over mobile edge networks, which have their own challenges. First, edge sites are resource constrained, which means that some sites may not be capable to run certain services. Second, service replication (duplicate service instances) may be needed across edge sites. Depending on the service, this may be costly or even infeasible. However, there is trade-off since service replication might be needed to meet SLOs and excess replication adds to redundancy and monetary cost. Third, system dynamics could necessitate service migration. This includes network changes or faults in any part of the system (e.g. congestion leading to performance degradation) and UE movement between zones controlled by different edge servers (migrating services/sessions to follow users and ensure minimal latency). It is thus a multi-objective problem to meet application SLAs and user experience while minimizing cost.

In general, dynamic service placement for mobile edge cloud can be of two types: heuristic and learning-based. Heuristic methods wait to receive triggers before taking a service replication, migration or eviction decision. Learning based methods are used when there is a possibility of further optimizing the system based on learning from the history of aforementioned triggers.

In this paper, we present three algorithms for dynamic service placement in the mobile edge cloud- two heuristic (set-cover greedy optimization and integer programming), and one learning-based
While choosing a placement algorithm for their application.

The microservice architecture used in our edge-computing application use case for our experiments to test and compare the three implemented algorithms based on certain quantitative and qualitative metrics. Our results demonstrate that although the learning-based algorithm generates the most optimal placement result compared to the other two, there are situations where the heuristic algorithms do perform quite well too. Further, the results provide insights and reference for practitioners while choosing a placement algorithm for their application.

In particular, our contributions are:

(1) Design and implementation of three multi-objective optimization algorithms to deploy services on mobile edge network, while satisfying constraints. In particular, our placement algorithms are- to the best of our knowledge, *the first such algorithms to take the underlying mobile network infrastructure into account*.

(2) Analysis of the algorithm solution based on placement quality, infrastructure deployment cost, determinism of results and algorithm execution time.

(3) Comparative evaluation of algorithms vis-a-vis each other.

The rest of this paper is organized as follows. In Section 2, we discuss the drone swarm navigation use case, which also forms the basis for experiments. The heuristic and learning based algorithms are described in Section 3. Results of the experimental evaluation of the three algorithms, and the comparative analysis, are presented in Section 4. Comparisons and directions from the related work are presented in Section 5. Finally, the paper concludes with suggestions for future work in Section 6.

2 APPLICATION USE CASE

The microservice architecture used in our edge-computing application use case is based on the autonomous drone swarm coordination scenario from DeathStarBench [11]. This is an open-source benchmark suite built with microservices that is representative of large end-to-end services, and is modular and extensible. DeathStarBench includes a social network, a media service, an e-commerce site, a banking system, and IoT applications for coordination control of UAV swarms.

Most of the top-of-the-line commercially available drones (DJI Mavic Air 2, Yuneec H520, Kespry Drone 2.0, Autel Evo, Skydio 2) are powered by 2500-5200 milli-ampere hour (mAh) lithium batteries and have flight times between 25-35 minutes. Any additional computational tasks (e.g. video analytics or object detection) placed on the drones will further reduce the flight time. Thus, we assume that existing drones do not have sufficient battery power or onboard hardware for such computationally heavy tasks, and need to be offloaded to edge servers. The drones are only responsible for perceiving the environment, sending the captured data to servers and implementing the navigation instructions from the controller.

Applications consuming the drone video feeds for further analytics tasks leverage the computational capabilities available at the edge and cloud servers.

The drone swarm mobilization application’s microservice chains, taken from [11] and their constituent services are depicted in Fig. 1.

Applications consuming the drone video feeds for further analytics tasks leverage the computational capabilities available at the edge and cloud servers.

The drone swarm mobilization application’s microservice chains, taken from [11] and their constituent services are depicted in Fig. 1. It performs motion planning and coordinated routing for a fleet of drones. It uses location, speed, orientation data, and performs object recognition and obstacle avoidance using the camera feeds. These tasks are compute and storage intensive, and data feeds from the drones are continually processed at the edge servers. The application is also time-critical since it deals with the real-time navigation of currently in flight drones. Given the limited battery power and compute resources on the drone, a viable solution is to transmit and process data at the edge site servers. Due to the drone’s proximity with the edge sites, the mobile network can provide a reliable, single hop, high-bandwidth and low-latency communication channel between the edge servers and the drones. While most of the processing is performed at the edge servers, a cloud server can optionally be used for persisting copies of data or constructing the initial route per drone.

Since the application relies heavily on edge servers, its processing throughput will also be limited by their hardware resources. To optimize deployment costs, and meet the strict quality of service (QoS), tasks must be intelligently assigned on servers using the placement algorithm. The complexity of deploying the drone coordination application is further increased by the fact that the navigation services must be accessible to the drones irrespective of their movement across coverage area of individual edge sites. In such scenarios, inefficient placement strategies could lead to over-provisioning of microservices across edge servers and low utilization of server capacity. This is costly and also limits the capacity of edge servers to execute more tasks in parallel. Furthermore, it has been seen that certain placement strategies [16, 25, 27] come up with optimal static placements for a given system state, but are unable to dynamic situations. In case of network, UE mobility or
failure events, they make ad-hoc runtime decisions which increase the deployment cost.

These challenges of resource management in an edge-centric heterogeneous deployment, strict QoS, requirement of service access across a coverage area, and moving UE (drones) make this application a fit use-case to test our placement algorithms.

3 DYNAMIC PLACEMENT ALGORITHMS

3.1 Overview

In this section, we present the three dynamic placement algorithms for edge computing applications based on microservices architecture. These algorithms are generalizable to any network topology and application scenario. Due to time constraints, we test the algorithms for the drone swarm coordination application (described in section 2) alone, but the algorithms have been designed and implemented to remain application agnostic. The first algorithm is a traditional heuristic weighted set-cover approach [13]. The second is also a heuristic algorithm based on a mixed integer linear programming approach as taken by [2]. Finally, the third algorithm is a reinforcement learning approach developed using the well-known OpenAI Gym environment [6] and uses the Proximal Policy Optimization (PPO) [24] algorithm.

3.1.1 Input and output of the algorithms. **Input:** Two sets of inputs are used; service description graph containing information about the service (e.g., microservice chains, set of their constituent services, resource requirements, QoS requirements communication overhead, data locality constraints, microservice collocation constraints, service requirement constraints) and a network graph describing the mobile network topology (e.g. connections between the UE and Base Station, links between the BS and the user plane anchor point (e.g., the User Plane Function in 5G, latency links etc. For better understanding, a table of data attributes of the configuration file are given below. TODO: Add a table of the required inputs, with attributes marked as required or not. For instance, Figure 2 illustrates three edge sites that are deployed at the edge of a 5G network. Each edge site is connected to an User Plane Function (UPF) via N6 interface. The UPF is the network function providing IP Anchor Point for the UEs in 5G network. One UPF can be connected to multiple BSs via N3 interface. And at a given time, the UE is connected to one BS. However, it is possible that one UE can access multiple edge sites through different UPFs. Each edge site has network connections with other edge sites. The network link latency of all links in Figure 2 can be determined and be made accessible to other entities in the mobile network and the edge infrastructure. In addition to the server sites, base station, and UPFs, the network topology file contain information such as node neighbors, total capacity and unit cost of resource of edge sites, and delays, source, and destination of communication link. The monitored available capacity of edge sites are also fed to the algorithms.

**Output:** The output for each algorithm is a set of replicas (or instances) of each microservice and their placement on edge servers in the cluster.

3.1.2 Objective Function. The primary goal of the algorithms is to ensure that availability of all application services at all user equipment (UEs) e.g. in our case drones that can access various

![Figure 2: An example network connectivity of mobile network and edge sites](image)

**Table 1: Technical terms involved in the network connectivity graph**

| Term | Definition |
|------|------------|
| User Plane Function (UPF) | A software component that supports features and capabilities to facilitate user plane operation. Examples include: packet routing and forwarding, interconnection to the Data Network, policy enforcement and data buffering |
| User Equipment (UE) | Any device used directly by an end-user to communicate |
| Base Station (BS) | Transmitter that relays wireless signals - typically between User Equipment and Edge Sites - using radio frequencies |
| Edge site | An intermediate cluster of one or more servers typically connected to a Base Station, other Edge Sites and one or more cloud servers |
| N3 interface | Interface between the radio access network and the UPF |
| N6 interface | Interface between UPF and any external service (or internal) networks or service platforms |

| Symbol | Definition |
|--------|------------|
| Graph G(N, L) | Network graph consisting of N network components and L links |
| Network components (N) | Consists of B = b1-bn BSs, U = u1-un UEs, E = e1-en edge sites |
| Network links (L) | Consists of L = L1-ln network links between nodes in N |
| Microservices (S) | Set of microservices in the application, S = s1-sn |
| Chains (C) | Larger service chains C = c1-cn made by the combination of microservices S |
| Latency requirements (R) | Application given SLOs for latency limits R = r1-rn of each chain in C |
| Placement (Ψ) | Mapping between e in E and s in S |

**Table 2: Definitions for symbols used in the algorithms**
application microservices from within the coverage area. In addition to ensuring service access, the algorithms aim to minimize the deployment cost while also meeting the application requirements defined in the scenario. The algorithms, supported costs and constraints are described below.

### 3.2 WSSP: Weighted Set-cover based Service Placement

The rationale behind weighted set-cover is to find a cover with the smallest number of sets from a collection of sets over a universe whose union equals the universe and whose total weights are minimum [13]. Our WSSP tries to minimize the weighted sum of costs such as deployment, processing, communication, and storage costs while fulfilling requirements of service performance and capacity constraints of sites in the distributed edge. As output it produces the smallest number of edge sites and their mapping to individual microservices that meets the latency limits for all UEs accessing the service. The algorithm, Algorithm 3.1, begins by sorting the microservice chains in ascending order by latency requirement (line 1). It then adds edge sites to the list of potential edge sites P for placement if latency requirement of a chain $c_i$ can be satisfied from any base station (BS) (helps transmit information between the UE and the edge site) to the edge sites and edge sites have enough capacity to admit all microservices that belong to $c_i$ (lines 2-7). If a site cannot host all microservices, the chain $c_i$ is split and potential subset of edge sites for the split microservices from $c_i$ are selected (line 9). The details of Split-and-assign can be found in Algorithm 3.2. Next, costs/weights to the potential edge sites are calculated using a scoring function (line 10). As the optimization problem has multiple objectives, a scalarization method [18] is used to transform the problem into a single objective optimization problem. The scalarized score function is expressed as:

$$F_{score} = (a_1F_{Dep}) + (a_2F_{CPU}) + (a_3F_{storage}) + (a_4F_{comm}) + (a_5F_{update})$$

(1)

where $F_{score}$ is a weighted sum of the normalized deployment ($F_{Dep}$), CPU ($F_{CPU}$), storage ($F_{storage}$), communication ($F_{comm}$), and update ($F_{update}$) costs. $a_1, a_2, a_3, a_4,$ and $a_5$ are the weights for the importance of deployment, CPU, storage, communication and update costs, respectively and $a_1 + a_2 + a_3 + a_4 + a_5 = 1$. The parameter values need to be chosen in order to achieve the desired trade-offs. Finally, the algorithm finds the minimal number of edge sites for $c_i$ that can satisfy the resource and latency constraints for any BS using Find_minimal_sites algorithm (refer Algorithm 3.3).

To find a split list of microservices of a chain $c_i$ and associated selected edge sites, Algorithm 3.2 uses a metric as a criteria for splitting (line 1). Depending on the objective, the metric can be based on the demand of individual microservices and available capacity of the site to be selected, dependency among microservices, and collocation constraints or a combination of them. Here we have used a simple approach that splits microservices based on microservice demand and available capacity of edge clusters. For the cluster that is selected to host the first part of the split microservices, a set of neighboring sites are sorted by their latency related to it in ascending order (line 2-4). If latency and resource requirements can be met, the remaining part of $c_i$ is assigned to a site that is being evaluated (lines 6-8). If resource requirement cannot be met then the same algorithm is called again with the remaining part of $c_i$ and the neighbour site as inputs (line 10).

Algorithm 3.1: WSSP Algorithm

| Input: | Sets of G, S, C, R as defined in table 2 |
| Output: | Placement mapping $\Psi$ |

1. Sort C in ascending order by latency requirements
2. for $c_i \in C$ do
3. Let P represent potential edges sites for placement.
4. for $e_j \in E$ do
5. Select $e_j$ if $r_i$ can be satisfied from any b $\in B$ to $e_j$
6. if $e_j$ has enough capacity for $c_i$ then
7. $P \leftarrow P \cup e_j$
8. else
9. $P \leftarrow P \cup \text{split-and-assign}(c_i, r_i, e_j)$
10. Assign weights W to P (using score function using Eq 1)
11. Find-minimal-sites(W,P)
12. end
13. end
14. end

Algorithm 3.2: Split-and-assign($c_i, r_i, e_j$)

| Input: | Chain $c_i$, latency requirement of $c_i r_i$, edge site $e_j$ |
| Output: | Split list of microservices $s_i$, selected edge sites $e_i$ |

1. Split $c_i$ based on metric
2. Assign the first part of $c_i$ to $e_j$
3. $C_r \leftarrow c_i, \text{remaining\_part}$
4. neighboringSet$_{e_j}$ $\leftarrow$ Find a set of $e_j$’s neighbor sites, and sort by latency in ascending order
5. for $k \in \text{neighboringSet}_{e_j}$ do
6. if $r_i$ can be met then
7. if resource requirement can be met then
8. assign $C_r$ to k
9. else
10. split-and-assign($C_r, k$)
11. end
12. else
13. Continue
14. end
15. end

Algorithm 3.3 is based on the weighted set-cover algorithm [13]. It finds the minimum number of sites for microservice placement to provide microservices within the required latency limit for all UEs. The basic idea is to select sites with minimum ratio of weights ($W_i$) and number of added sites ($S_i$) (see line 4-5).

### 3.3 MISP: Mixed-Integer linear programming based Service Placement

The MISP placement algorithm is executed in three stages. Initially, the algorithm evaluates cost of deploying microservices to servers
based on compute resources and constraints. The resulting matrix is solved using the mixed integer linear programming solver to return an initial placement mapping each microservice to one server. Next, microservice communication costs are taken into account. MISP tries to minimize these costs by deploying communication-heavy microservices closer to their dependent microservices. These updates can reduce the overall placement cost. In the third stage, MISP checks for SLO attainment and if all UEs are able to access the services. In case some UEs report one/more services exceeding the SLO, MISP deploys more microservice instances close to these UEs.

Our algorithm improves upon the MCAPP-IM [2] algorithm in four ways. First, based on microbenchmarks, we found that Hungarian-matching took three to four orders of magnitude greater time to solve larger systems of equations i.e. larger network and service chains. Hence, we replaced the hungarian algorithm as the solver with MISP. Second, the MCAPP-IM algorithm assumed that only one microservice would be assigned to any server. This is not realistic and server utilization needs to be maximized. Third, MISP supports a number of real-world application constraints which include service constraint (microservices placed on certain servers due to hardware requirements), collocation constraint (tightly coupled microservices that need to be placed on the same server) and data locality constraints (sensitive user data that must be stored on servers within a defined region). The MISP algorithm is described in Algorithm 3.4.

The initial microservice-server placement is done based on deployment cost alone (lines 1-3, Algorithm 3.4). For this, application and topology data is read and costs for \( y \) (compute), \( \delta \) (microservice-user communication), and \( \rho \) (microservice migration) are computed. These values help formulate the cost matrix, i.e. \( \text{TM} \) in line 2. \( \text{TM} \) is a \( kn \times n \) matrix where \( k \) is the number of microservices and \( n \) is the number of servers. Each row in \( \text{TM} \) represents a microservice, and columns represent its deployment cost on different servers \( e \). This factors in heterogeneous edge compute where the deployment costs can vary based on the server hardware. \( \tau \) adds or ignores \( \rho \) (microservice relocation) costs based on whether a microservice can be migrated from server \( e_i \) to \( e_j \). The \( \text{TM} \) also enforces the application’s service \( (s \cdot c) \) and data locality \( (l \cdot c) \) constraints. It does so by determining microservice-server placements which are not allowed by the application and assigns those costs to a large numeric constant, \( \kappa \). The solver (which takes \( \tau \) as input) naturally eliminates solutions with such server-microservice combinations as they would significantly increase the placement cost.

The \( \text{TM} \) and edge site capacity \( (c_{ap}) \) constraints are given as input to \text{mip Solver} which uses Google’s assignment solver. Based on these inputs, we construct constraints for the MIP solver to generate the initial placement solution, mapping one microservice to exactly one server while minimizing the overall cost. This gives us the initial microservice placement solution \( (\Psi_{mip}) \), based on the compute and relocation costs, application constraints and the edge server resource limits.

Second, a heuristic search \text{LSearch} (Algorithm 3.5) is used to reduce the inter-microservice communication costs in \( \Psi_{mip} \) solution, obtained in (line 3, Algorithm 3.4). It computes \( \Lambda \), the communication costs incurred by each of the microservices based on their interaction with other microservices \( (s_{ij}) \) and current server placement in \( \Psi_{mip} \). \text{LSearch} evaluates combinations to move heavily communicating microservices closer to each other since this could reduce the communication cost and the overall deployment cost. However, we note that this migration of microservices is only allowed where it is permitted based on defined application constraints. Costs for the different combinations \( (\Psi_{LSearch}) \) is compared with \( \Psi_{mip, cost} \). If \( \Psi_{LSearch} \) is lower, the placement solution is updated, else, the previous solution is retained.

---

**Algorithm 3.3:** Find-minimal-sites(W,P)

**Input:** A collection of subset of the universe BS covered by potential edge sites, \( S = \{ S_1, S_2, ..., S_m \} \). Weights of elements of \( S \), \( W = \{ W_1, W_2, ..., W_m \} \)

**Output:** Minimum number of cost effective edges sites that cover all elements of BS

1. Let I represent set of elements included so far.
2. Initialize I = []
3. while I is the same as BS do
   4. Choose \( S_i \in S \) minimizing the ratio of the weight \( W_i \) and number of newly added elements, i.e., \( W_i / |S_i - I| \).
   5. Add elements of above picked \( S_i \) to I, i.e., \( I = I \cup S_i \)
4. end
5. Return I

---

**Algorithm 3.4:** MISP Algorithm

**Input:** Sets of G, S, C, R as defined in table 2, For \( t > 0 \), solution \( \Psi_{t-1} \) is also given

**Output:** Placement solution \( \Psi_t \)

1. Read application file to extract computation (\( y \)), constraints and communication costs (\( \delta \))
2. Populate \( \tau \), containing cost for deploying the microservices across edge sites
3. Obtain \( \Psi_{mip} \) using \text{mip Solver} which places all microservices on exactly one server
4. Calculate the initial cost \( \Psi_{mip, cost} \) using \text{calcCost}
5. New placement \( \Psi_t \) using \text{LSearch} to reduce communication overhead
6. Get final \( \Psi_t \) after calculating \text{serviceLatency} for each UE

---

**Algorithm 3.5:** LSearch()

**Input:** Initial placement \( \Psi_{mip} \) and its cost from \text{mip Solver}

**Output:** \( \Psi_{LSearch} \): Placement after \text{LSearch}

1. Compute communication cost using \( \Psi_{mip} \) placement and microservice dependencies
2. Find microservices with highest communication overheads
3. Try more placement combinations to reduce communication cost of identified microservices
4. Update placement solution if new placement combination reduces overall cost

---

1https://developers.google.com/optimization/assignment/overview
Third, MISP ensures availability of application services are provided within latency limits, to all the UEs in the coverage area. 

Algorithm 3.6 invokes calcLatency to calculate service chain latency for all the services provided at each UE. As mentioned previously, service latency for a UE is the sum of all network links in the graph, connecting the UE to the nearest servers hosting the microservices required for that service. Duplication of microservices on servers close to the UE is triggered if a service chain is not provided within the latency limit (line 14, Algorithm 3.6). To provide maximum services with minimum deployment cost, the microservice duplication is done in ascending order of compute load (line 13, Algorithm 3.6), and at the server connected to the UE with lowest latency link. To reiterate, the new deployment of a microservice $s$ on server $e$ is dependent on the application constraints. After each microservice duplication, service latency to UEs is re-computed. The duplication step is repeated until SLO is met or there are more combinations to try. At this stage, we have the final placement solution $\Psi_f$.

Algorithm 3.6: serviceLatency()

| Input: $C, E, B, r, \Psi_f, G, \text{colloc}$ |
| Output: $\Psi$: Final placement after latency check and duplication |
| 1 Iterate over base stations $B$ |
| 2 For each edge server calculate the shortest path $(G, b, e)$ network latency to the base station |
| 3 Iterate over service chains for each base station $B$ |
| 4 Use calcLatency to determine total network latency to provide chain $c$ to BS $b$ |
| 5 If current latency $r$ lesser than permissible limit, continue |
| 6 Else, use duplicate to greedily create new instances of a microservice $s$ on a nearby server |
| 7 Update $\Psi_f$ with newer $\Psi_f'$ after every placement change |

3.4 RLSP: Reinforcement learning based Service Placement

Reinforcement Learning (RL) [19] is a machine learning technique where agents learn a policy from trial and error, in an interactive environment. The concepts of history and state are central to it. The history describes all the interactions that have taken place between the environment and the agent, and its abstract representation is called state. Key concepts of RL include delayed rewards (short term and long term rewards), importance of time (state at time $t$ + 1 is dependent on the state at time $t$) and that an agent’s action affects its next input, next action and its future path. The RL agent learns about the environment by repeatedly taking actions and modifying states based on the reward obtained. The goal of the RL agent is to maximize the overall reward. We designed and implemented a custom OpenAI Gym [6] environment, called RLSP (Algorithm 3.7), to model the service placement problem as a Reinforcement Learning (RL) problem.

The algorithm includes an RL agent which has (i) action_space and actions, (ii) observation_space, and a reward function. In the init method (Algorithm 3.8), the application and topology data are read, and the cost matrix i.e. TMatrix ($r$) is computed. Here, we also define the shape and type of the environment’s action_space and observation_space. An action by RLSP includes three attributes: (i) action type (deploy, evict or hold a microservice), (ii) microservice index $[0, s_j - 1]$ where $s_j$ is the number of microservices in the application, and (iii) server on which the action has to be taken $[0, e_n - 1]$, where $e_n$ is the number of servers. Thus, the action space (actions taken by the agent) is a list of three values where each value is a discrete integer within the defined range. The observation space (environment information) gives a structure for the environment to return information on (i) the number of service chains $[0, C_k]$ ($C_k$ is the number of service chains in the application) accessible by each of the $b$ BS (base stations), and (ii) the number of microservices $[0, s_j]$ deployed on each of the $e$ servers. Thus the observation space is a list of size $[B + E]$. Since both action_space and observation_space take discrete integer values in a defined range, they are of MultiDiscrete type. Besides this, init also initializes variables for reward, counter, serverMicro, matrix of microservices deployed on servers, and userServicesAccess (matrix of service chains accessible from base stations).

Algorithm 3.7: RLSP: An RL agent for Service Placement

| Input: Observation obs representing the current environment state and the reward value |
| Output: An action to perform on the environment to maximize reward while satisfying service latency, application and resource constraints |

1 _init_(self) |
2 Define environment variables, action_space and observation_space |
3 _next_observation(self) |
4 Loads the next frame and provides environment information to the RL agent |
5 step(self, action) |
6 Performs an action and obtains reward and observable information from the environment |
7 _take_action(self, action) |
8 Performs an action on the environment |
9 reset(self) |
10 Reset values in all variables and matrices |
11 render(self) |
12 Display current environment state, reward and counter |

Algorithm 3.8: _init_(self)

| Input: $\Psi$: Given placement solution |
| Output: $\Psi$cost: Placement cost |
| 1 Initialize reward, rewardOrPenalty, counter, max_steps and create matrices serverMicro and userServicesAccess |
| 2 Evaluate compute ($\gamma$) and constraint costs from application, topology files |
| 3 Calculate matrix $r$ using calcT |

4 Define action_space and observation_space
The RL agent could also take invalid actions. These include: trying to hold or evict a microservice which is not currently running on a server, or trying to deploy a microservice on a server where it is already running. In such cases, a small negative reward is given to the agent and the serverMicroservices matrix is left unchanged. This is also the stage where application constraints are enforced by checking the cost matrix $r$. Here, the actions taken by the agent, in violation of the application constraints (data locality, service or collocation) are penalized. Upon receiving a penalty, it is expected that RLSP would learn and avoid such actions in the future.

After performing the action, step calls next_observation (Algorithm 3.9). Here, the next input state of the environment is provided to the agent. This function also gets the updated environment state from serverMicroservices, which is given as input to the checkUserLatency function to calculate the service chain latency at various UEs. After performing the latency check, serviceUserAccess matrix is returned, which contains information on which service chains are accessible from which base stations. The latency check returns the serviceUserAccess matrix, which contains information on the service chains accessible from the various base stations.

The step function uses the two matrices to find the step’s net reward. Using the serviceUserAccess matrix, a total positive reward is computed based on the total service chains accessible from the different UEs. As the number of service chains provisioned increase, the positive component of the reward will also increase. Another factor included in the reward function is the deployment (compute and communication) cost. It is computed using the serverMicroservices matrix and it is subtracted from the reward. This is because the placement strategy aims to provide maximum services through minimum microservice instances, i.e., lower deployment cost. For effective learning of the RLSP, this net reward is further scaled using a discount factor. The discount factor diminishes the value of the reward in the initial time-steps, and keeps increasing its weight as the number of time-steps approach the $\max_steps$ value. This ensures that the RL agent is not significantly influenced by large short term positive or negative rewards. Instead, it works towards a large cumulative reward in the long term. At this stage, the step function returns the next observation and current reward to RLSP.

Thus, in order to maximize the cumulative reward, the RLSP will try to gain more positive reward by ensuring greater accessible service chains and taking valid actions. It will also try to minimize the negative reward from deployment costs and invalid actions. The implementation also has helper functions of reset and render (Algorithm 3.7) to re-initialize all variables after the current iteration ends, and to display the current system state.

Finally, we train and test the RLSP. To train our RL agent, we use the Stable Baselines3 [22] library. This library provides reliable PyTorch implementations of state-of-the-art reinforcement learning algorithms, and a framework to train, test and save RL agents built on the OpenGym environment. In particular, we use the Proximal Policy Optimization (PPO) [24] algorithm to train our agent. The PPO algorithm combines benefits of multiple workers from A2C (Actor-Critic Algorithm), and of trust region from TRPO (Trust Region Policy Optimization). The PPO algorithm uses a Trust Region based objective function which is also compatible with Stochastic Gradient Descent. In particular, PPO provides an ease of

---

### Algorithm 3.9: _next_observation()

**Input:** self

**Output:** obs

1. Load the next input frame of application, network data
2. `userServAccess ← serviceLatency (C, E, B, Ψ)`
3. Populate obs of `observation_space` type using `userServAccess` and `servMicro`.

---

### Algorithm 3.10: _take_action()

**Input:** self, action

**Output:** status, rewardOr Penalty

1. `act_type, act_micro, act_serve ← action`
2. Initialize `rewardOr Penalty ← 0`
3. if `action is valid based on servMicro` then
   4. `rewardOr Penalty ← small positive reward`
   5. Update `servMicro`
4. else
   5. `rewardOr Penalty ← small/large negative penalty`

---

### Algorithm 3.11: step()

**Input:** self, action

**Output:** obs, reward, done

1. `counter ← counter + 1`
2. `status, rewardOr Penalty ← _take_action(self, action)`
3. `obs ← _next_observation()`
4. Get `servAvailable from obs`
5. `cost ← Computed from calcCost(Ψ) and δ`
6. Get new reward based on `servAvailable, cost, max_steps` and `delay_modifier`
7. if `counter >= max_steps` then
   8. `done ← true`
9. else
   10. `done ← false`

---
4 EVALUATION

4.1 Experimental setup

4.1.1 Network topology. We assume a hypothetical telecommunication network setup spanning four geographically distributed edge servers (E1 - E4), also called edge sites, which are identified by a unique id and lat, long location. The topology used for the simulation experiments was similar to Figure reffig:mntopology. Their information specified is assumed to include the memory, disk storage, compute capacities, and also the presence of specialized hardware such as Graphics Processing Units (GPU), if any. These resource capacities set an upper bound for the algorithm to only deploy microservices on a server as long as resources are available. The presence of specialized hardware on a server also helps us meet the constraints for microservices that utilize it. For the convenience of our algorithms, we also define a unit cost of storage and compute on a server based on the resources it provides. This plays a role in heterogeneous compute environments where different edge servers provide varying compute power, resulting in different deployment costs. (It is to be noted that the compute/storage capacities of the various edge sites are assumed to be heterogeneous.)

4.1.2 Application and workloads. The experiments to test the dynamic service placement algorithms use the drone swarm mobilization application, introduced in Section 2. This application has 13 service chains which utilize a total of 23 microservices. The service chains were modeled based on the system design given in Figure 1 and are described in Table 3.

Each service chain in the application has a configurable latency limit \( r_q \). The QoS (primarily, latency) requirements of the 13 chains are broadly categorized into ultra-low or strict (0.2 ms) latency, moderate or medium (0.4 ms) latency, and relaxed or high (0.6 ms) latency. These are typical values for so-called Ultra Reliable Low Latency Communication (URLLC), especially for a critical use case such as drone swarms [4]. These 13 chains, in different ratios, are used to create the Ultra-Low (W1), Moderate (W2) and Relaxed (W3) application workloads. Such workloads enable us to assess the placement quality of algorithms when applied to diverse applications with varying QoS requirements. The details in Table 4 show the count and proportion of ultra-low, medium and relaxed service chains in workloads W1, W2, and W3, respectively. The number of ultra-low latency service chains increase their percentage from 30% to 55% and further to 70% as we move from W1 to W2, and finally to W3. These workloads are also modeled to have 20% microservices with service requirement constraints (e.g. specialized hardware); 10% microservices with collocation constraint (e.g. tightly coupled microservices which run on the same server) and 20% microservices with data locality constraint (e.g. sensitive data to be placed only on edge servers in a region).

4.1.3 Hardware used. We use simulation to conduct experiments on an Intel Core i5-8350U @ 1.70 GHz machine with 16 GB RAM. One of the reasons to do simulations is because of the limited access to actual compute and networking testbeds to deploy microservices based application, capture network metrics and apply our 3 algorithms. Secondly, there are very few reinforcement learning frameworks for both training and testing RL agents. If at all a physical environment is created to train an agent for a policy (e.g. robotics), they are very application specific. This is also because the actuation metrics for the RL agents are different. Owing to these limitations, and to have a level comparison between the two heuristic and reinforcement learning approaches, we use simulation.

Comparative evaluation: The three algorithms WSSP, MISP and RLS are implemented to provide solutions to workloads W1, W2 and W3. Their solutions are evaluated and compared on four key parameters: placement quality, application deployment, consistency in results and algorithm execution time.

4.2 Placement Quality

A placement algorithm’s solution quality can be measured by the extent to which it satisfies Quality of Service (QoS) requirements, application constraints, and access to microservices across the coverage area.

4.2.1 Costs and constraints. The three algorithms successfully produced solutions that satisfied the QoS requirements in workloads W1, W2, W3. The placement fulfilled the service, collocation, and data locality constraints defined for microservices in the drone application. The algorithms also employed mechanisms to reduce the overall placement costs. The only exception here is that WSSP...

| Workload | Relaxed chains | Moderate chains | Ultra-low chains |
|----------|----------------|-----------------|------------------|
| W1       | 2 (15%)        | 7 (55%)         | 4 (30%)          |
| W2       | 2 (15%)        | 4 (30%)         | 7 (55%)          |
| W3       | 2 (15%)        | 2 (15%)         | 9 (70%)          |

Table 4: Workloads and their service chain compositions
did not take into account collocation and data locality constraints, which it does not support. This results in WSSP deploying some microservices on edge server E1, which are constrained to run only on edge server E3, as shown by MISP and RLSP in Figures 5a, 5b and 5c.

4.2.2 Services provided in the coverage area. The placement solutions from the three algorithms provided 100% service coverage, i.e., microservices deployed at assigned edge servers were able to provision all services within their latency limits, for all UEs in the network coverage area. MISP performs duplication of microservices on edge servers close to the UE till the QoS latency is met. This check is performed at all UEs, so that 100% service coverage is ensured. WSSP does not directly perform duplication of microservices until the latency is met. It has knowledge of all the UEs where latency limit can be satisfied, and it then selects and deploys services on the smallest number of edge servers which provide full coverage. RLSP on the other hand gets a positive reward for each service chain that it provides to any UE, within the latency limits. Hence it takes into account the costs/constraints and deploys microservices on edge servers to provide 100% service coverage, thereby maximizing its reward.

4.2.3 QoS for UE. The 100% service coverage for the all solutions was verified during experimentation. The distributions for latencies at UEs for two representative moderate and ultra-low service chains are shown in Figure 3. These two categories of service chains constitute up to 85% of the three workloads. In the moderate QoS chain, we see that sufficient deployment of microservice and the latency links between UE and the edge servers ensures that QoS requirements are 35% to 82.5% within the 0.4ms latency limit (Figure 3a). For the ultra-low latency service chain, this ranges between the min/avg/tail latencies and 0.2ms limit narrows to 2% to 65% (Figure 3b), as the MISP and WSSP algorithms make new deployments to ensure that QoS requirements are met. Meanwhile, RLSP manages a lower latency compared to MISP and WSSP due to substantial microservice deployment on E4, which is close to many UEs, thereby reducing the end-to-end latency at the UEs.

4.2.4 Variability of the deployment with evolving conditions. Several placement decisions are made by the algorithms to adapt to the changing application requirements and dynamic network conditions. This requires a finite non-zero time and could cause QoS violations while the placement algorithm makes decisions on deploying, migrating or evicting microservices. Figure 5 shows the microservices deployed on edge servers by the algorithms for the three different workloads. It can be seen that WSSP and MISP follow a similar trend of retaining the existing microservices on E1 while deploying new microservice instances on E4 due to the stricter QoS requirements as we move from W1 to W3. On the other hand, RLSP’s placement solution remains the same. It means that in case the drone application service requirements are changed at run-time, WSSP and MISP algorithms will have a short time-span where some service chains will be temporarily inaccessible, while RLSP would have fewer or no disruptions. This is evident from Figure 4 which shows the service access in the coverage area as the drone application transitions from W1 to W2 at timestamp 100, and from W2 to W3 at timestamp 200. The WSSP and MISP solutions make new deployments for W2 and W3 which makes 7% service chains unavailable across multiple UE. This is restored after new deployments are completed at E4. The RLSP solution takes nearly 40 iterations to converge on its first solution. However, its solution is stable and requires no new deployments as future workload transitions made by the application.

This demonstrates the advantage of RLSP to use past experience to reach placement solutions, i.e., the RL agent has sufficient prior knowledge of the system which helps it reach a better quality placement solution. The agent realizes that for stricter latencies, additional microservices need to be deployed, and is able to do so in advance. Thus, this behavior provides a stable deployment which will make relatively fewer changes as the application requirements change with time.

4.3 Application deployment

The application deployment criteria considers the number of microservices deployed, the cost incurred, the number of edge servers used and their load.

4.3.1 Microservices deployed and deployment cost. One of the aims of the placement algorithms is to reduce the deployment cost, while satisfying the application QoS and constraints. By optimizing the placement solution, the server load and the cost borne by infrastructure providers will be reduced. It must be noted here that WSSP has a relaxation since it does not support the service hardware requirements, microservice collocation, data locality constraints. For example, MISP and RLSP solutions deploy certain microservices on E3 as they are bound by constraints. This can be seen in Figure 5 which shows the microservice counts per server for all the workloads and algorithms.

The WSSP solution deploys a total 34, 38 and 42 microservices for the three workloads. MISP has a similar deployment but places 36, 40 and 44 microservices across the edge servers. The RLSP algorithm provides the minimum cost solution since it deploys only 23 microservices to provide 100% service coverage for all three workloads. Thus, compared to WSSP and MISP, the RLSP solution deploys 32-45% and 36-48% fewer microservices respectively. This would lead to significant reduction in infrastructure cost.

The difference in the number of microservices deployed across algorithms is because of the way they arrive at the final solution. WSSP makes one-time decisions considering all factors, and does not re-visit the result. MISP on the other hand deploy sufficient microservices to provide 100% service chains across the coverage area, but it does not have a microservice eviction policy. Thus, both do not further try to minimize deployment cost while still meeting the application QoS requirements. The RLSP algorithm on the other hand, has an evict microservice action which enables it to solve with minimum deployment cost. In the training process it was observed that RLSP first formed a complete but less optimized policy i.e. deployed more microservices, but provided 100% service coverage. On further training, it also used its evict action to evict some microservices. This helped it converge to a final policy with less microservice deployment and 100% coverage.

4.3.2 Edge Servers contribution and utilization. All the three algorithms take into account the edge server site capacities and deploy
microservices on edge servers as long as sufficient resources are available. In addition to the compute and storage resources, the algorithms also take into account the network link latency during communication from edge servers to UEs. Deploying microservices on edge servers having resources but large link-latency will not help the algorithms to provide a solution which satisfies the application QoS requirements. Thus, the algorithms take into account both the factors of resources and link latency while making placement decisions.

Figure 6 shows each server’s contribution, as a percentage, in the microservice placement solutions produced by the three algorithms. It can be seen that most of the microservices are deployed among edge servers $E_1$ and $E_4$ since they are closer to many UEs. This provides ease of meeting QoS requirements through lower latency network links. The plot shows that server $E_1$ contributes to 50-67% of all microservices deployed by WSSP and MISP across the three workloads. Server $E_4$ is also close to several UEs and some moderate and relaxed services were provisioned through $E_1$ for workload $W_1$. However, as the QoS requirements became stricter in moderate $W_2$ and ultra-low $W_3$ workloads, WSSP and MISP deploy additional microservices on server $E_4$ (as shown in Figure 5). Across the three workloads, server $E_4$ contributes to 31-45% of the microservices deployed by WSSP and MISP. The RLSP algorithm finds the most optimal placement by placing 92% of its total deployed microservices on $E_4$. It also ensures that the deployment is within the server’s resource limits.

In terms of server utilization, it is seen that edge servers $E_2$ and $E_3$ are not used at all by WSSP. They are utilized in a limited capacity by MISP and RLSP algorithms which place a few microservices on these edge servers to meet the application’s constraints. Utilization of edge servers $E_1$ and $E_4$ is similar for WSSP and MISP algorithms across workloads. On the other hand, the RLSP algorithm utilizes server $E_4$ to a considerable extent in all three workloads.

4.4 Consistency of results
To validate the deterministic nature of the algorithms, the experiments for each workload were re-run multiple times. The heuristic based approaches of WSSP and MISP - as implemented for our paper - are deterministic as they produced the same placement solution each time that they are given the same network conditions, application constraints and QoS requirements. The same cannot be said about RLSP. In contrast to WSSP and MISP algorithms which compute the solution through heuristic algorithms and mathematical solvers, the RLSP solution is obtained over multiple time-steps where it takes an action decision at each time-step, as described in Algorithm 3.7. The RL agent in the RLSP algorithm takes actions based on its learning from the training stage and the given network deployment and application requirements. Based on its actions, the environment keeps generating the reward and a new state for the next time-step. Since these actions, reward values and environment states vary across timestamps, the path the RLSP algorithm takes to obtain the final converged placement solution is non-deterministic. In other words, although RLSP arrives
at 100% service accessibility result after several steps, the placement solution and the cumulative reward vary across runs. That being said, the placement solution for each run achieves all the placement objectives of service coverage, application constraints and QoS requirements.

### 4.5 Algorithm execution time

The time taken by the three algorithms to arrive at their final placement solutions is referred to as the algorithm execution time, shown in Figure 7. We see that WSSP is nearly 13.8% faster than MISP since the former supports fewer constraints leading to saved computation time. The RLSP algorithm, as stated earlier, arrives at the solution after multiple time-steps. For the service placement problem, RLSP algorithm took nearly 40 iterations to converge at the final placement solution, and it spent about 0.002 seconds at each step to to predict the next action. Thus, it spent about 0.08 seconds to find the placement solution for either of the three workloads.

### 4.6 Comparative Analysis

Our comparative evaluation showed that all three implemented algorithms were able to meet the latency requirements, but in different ways. The WSSP algorithm was not able to meet some constraints since they were not designed into the algorithm. Overall, the RLSP algorithm was able to provide better results, although it takes more time since it is based on reinforcement learning which converges after multiple iterations. An extensively trained RL agent is likely to experience much lesser variations in its solution (due to varying network conditions) compared to the other two algorithms. This provides stability in application deployment. It also deploys fewer microservices, leading to lower deployment costs, since it has an explicitly defined eviction policy. Although all three algorithms do provide good server utilization,
it was seen that as the latency requirements got tighter, the RLSP algorithms showed better server utilization. However, due to its non-deterministic nature, it is not guaranteed to produce the same result after every run, although every result it produced did meet all the QoS requirements.

In conclusion, the three algorithms can be seen as a spectrum of dynamic placement solutions, ranging from the faster (WSSP) to slow (MISP) and slower (RLSP), although this is offset by solution quality (from constraint satisfaction viewpoint). Hence users can select the one most appropriate for their respective needs.

5 RELATED WORK

5.1 Edge computing

Dziyauddin et al. [7] provides an overview of the architecture for vehicular edge computing, including a review of computation offloading and content caching and delivery approaches for vehicular edge computing. Another work [8] investigates ultra-reliable and low-latency communication (URLLC) in fog networks, including a task distribution scheme with proactive caching of computing tasks at edge nodes. Efforts have also been made to build tools to test scheduling policies for edge computing [3].

5.2 Dynamic service placement

Overall, service placement for low latency and intensive applications bring conflicting cost functions including computation, communication and migration which need to be optimized, which we have demonstrated in this paper.

Virtual Reality (VR) and Augmented Reality (AR) applications have already adopted the edge computing paradigm to overcome the unpredictable latency of cloud-based processing. Wang et al. [28] places service entities for VR applications in the edge environment. Similarly, in VR group gaming, cloudlets are used [30]; while an algorithm for service placement for video analytics is presented in [10].

Salah et al. [23] present an overview of the several optimization metrics - mainly latency, cost and resource utilization - explored by recent works in service placement for fog and edge computing. Wang et al. [29] consider VR applications and formulate four cost types - activation, placement, proximity and collocation (resource contention among the services placed on the same edge cloud). We focus on the placement costs and also take into account proximity and collocation. Skarlat et al. [26], like us, focus on QoS requirements and exploit the presence of edge computing to reduce QoS violations compared to a purely cloud-based approach. They also model applications as a combination of services and take into account the application requirements and the resources and latency links between the fogs (edges). However, unlike us, they do not take into account the utilization of resources in its placement and realistic network data such as communication delays.

Similar to ours, Ayoubi et al. [1] model equations for service latency, fog utilization, communication and computation costs as their key optimization objectives. He et al. [14] categorize edge resources based on shareable (storage) and non-shareable (CPU cycles, bandwidth) resources for optimal provisioning of services and request scheduling. However, while deployment costs are considered by the other works, their placement strategies do not support application specific constraints such as data locality and specialized hardware requirements which we have considered.

Mahmud et al. [17] take a slightly different approach where their placement objective is based on user expectations and enhancement of Quality of Experience (QoE) with respect to access, service delivery and resource consumption. On similar lines, [21] also lays emphasis on user mobility while optimizing edge service performance.

Salah et al. [23] provide a clear service placement taxonomy based on four distinct aspects: control plane and coordination (centralized or distributed); placement characteristics (offline or online); nature of system (dynamic or not); mobility support (yes or no). In our work, we are mainly concerned with the second and third of these criteria.

Heuristic based: Skarlat et al. [26] design and implement a genetic algorithm to make service placement decisions. The genetic algorithm allows to investigate a large search space while providing viable solutions in polynomial time. The ITEM algorithm developed in [29] uses the graph-cut method to solve the combinatorial optimization problem based on a series of minimum s–t cut instances. A similar approach is also taken in [30]. He et al. [14] formulate service placement and request scheduling as an integer linear program aiming at serving maximum requests while minimizing the resource consumption. Similarly, Farhadi et al. [10] take a joint approach in optimizing service placement and request scheduling by combining greedy heuristic and shadow request scheduling techniques.

Learning based: Ayoubi et al. [1] propose a four-stage MADE strategy which consists of monitoring, analysis, decision making and execution phases. Ouyang et al. [20] take an adaptive user-centric approach towards service placement which also considers user-mobility and user’s preferences, and which uses an online learning based algorithm. Brandth et al. [5] uses a model-free Q-learning based approach for service migration. Finally, Gao et al. [12] delve into preserving application QoS in the mobile edge computing framework by also laying emphasis on the network factors, since both access network and edge nodes are vulnerable to congestion.

None of the above works provide any sort of detailed comparison among specific algorithms across the spectrum encompassing heuristics (simpler and more complicated) and learning-based algorithms to provide practitioners a possible range of solutions with various tradeoffs.

6 CONCLUSION AND FUTURE WORK

In this paper, we have addressed the well-known dynamic service placement problem. In contrast to earlier works that usually present only a single solution to the problem from a single perspective (albeit with variations), our paper proposes and compares three diverse algorithms to resolve this problem. These algorithms cover the spectrum from simpler heuristics to more involved learning-based approach. In so doing, we have presented the algorithms, as well as a detailed comparative evaluation on a realistic drone swarm application. While the results of our comparative evaluation demonstrate that the learning-based approach does provide better results, we have also shown that the simpler heuristics are not to be ruled out, especially when a trade-off such as execution time is
considered. Hence the work in our paper can be considered as - to
the best of our knowledge - the first such detailed comparative study
of dynamic placement algorithms along with recommendations to
practitioners on the best scenarios in which each of them can be
used.

Another novel aspect of our paper has been to incorporate the
intricacies of the underlying mobile network infrastructure while
developing our algorithms, something that is also - to the best of our
knowledge - the first such detailed treatment of this problem. We
hope that this will give impetus to further research on dynamic ser-
vice placement approaches that incorporate the underlying mobile
network infrastructure.

Hence our future work will involve not only the above incor-
poration of mobile network infrastructure into dynamic service
placement approaches, it will also involve the following: (1) de-
velopment and evaluation of components that actually implement
microservice migration across edge servers (perhaps either on a
network emulator or a real 5G testbed); (2) enrichment of microser-
vice migration via integration of service mesh technology [15] for
dynamic management of microservice traffic, especially to demon-
strate microservice migration on a large-scale; and (3) investigation
of how our algorithms can be deployed and evaluated on real-life
eamples (perhaps such as those described in [11]) on a large scale.

ACKNOWLEDGMENTS

We wish to thank Ajay Kattepur for assistance on RLSP, as well as
our respective colleagues in Ericsson Research for their feedback.

REFERENCES

[1] Masoumeh Ayoubi, Mohammadreza Ramezanpour, and Reihaneeh Khorsand. 2020.
An autonomous IoT service placement methodology in fog computing. Software:
Practice and Experience (2020).

[2] Tayebeh Bahreini and Daniel Grou. 2017. Efficient placement of multi-
component applications in edge computing systems. In Proceedings of the Second
ACM/IEEE Symposium on Edge Computing. 1–11.

[3] Awadh Basukar, Anderson Da Silva, Adrien Lebre, Clement Mommessin, Pierre
Neyron, Tanig Nogko, Yuic Cordeil, Denis Trystram, and Alexandre van Kem-
pen. 2020. Investigating Placement Challenges in Edge Infrastructures through a
Common Simulator (extended version). Ph.D. Dissertation. INRIA.

[4] Irem Bor-Yaliniz, Mohamed Salem, Gamini Senerath, and Halim Yanikomeroglu.
2019. A learning-based approach for dynamic service placement problem in Fog
and Edge Computing. ACM Conference on Cloud Computing Technology and Science
(CloudCom). IEEE, 468–476.

[5] Florian Brandtherm, Lin Wang, and Max Mühlhäuser. 2019. A learning-based
framework for optimizing service migration in mobile edge clouds. In Proceedings
of the 2nd International Workshop on Mobile Edge Systems, Analytics and Networking.
12–17.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schul-
man, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. arXiv preprint
arXiv:1606.01540 (2016).

[7] Rudradhata Akmah Dzyauddin, Dasiy Niyato, Nguyen Cong Luong, Mohd
Azri Mohd Izhah, Marwan Hadhiri, and Sabwani Daud. 2019. Computation
offloading and content caching delivery in vehicular edge computing: A survey. arXiv
Preprint arXiv:1912.07863 (2019).

[8] Mohammed S Elbamby, Mehd0 Bennis, Walid Saad, Matti Latva-Aho, and
Choon Seon Hong. 2018. Proactive edge computing in fog networks with latency
and reliability guarantees. EURASIP Journal on Wireless Communications and
Networking 2018, 1 (2018), 1–13.

[9] Chen Fan and Li Li. 2020. Service Migration in Mobile Edge Computing Based on
Reinforcement Learning. Journal of Physics: Conference Series 1584 (Jul 2020),
012058. https://doi.org/10.1088/1742-6596/1584/1/012058

[10] Vajjheh Farhadi, Fidan Mehmct, Ting He, Tom La Porta, Hana Kharamfoush,
Shiqiang Wang, and Kevin S Chan. 2019. Service placement and request schedul-
ing for data-intensive applications in edge clouds. In IEEE INFOCOM 2019-IEEE
Conference on Computer Communications. IEEE, 1279–1287.

[11] Yu Gan, Yanqi Zhang, Dalun Cheng, Anikatha Shetty, Priyal Rathi, Nayan Kataraki,
Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An
open-source benchmark suite for microservices and their hardware-software
implications for cloud & edge systems. In Proceedings of the Twenty-Fourth Inter-
national Conference on Architectural Support for Programming Languages and
Operating Systems. 3–18.

[12] Bin Gao, Zhi Zhou, Fangming Liu, and Fei Xu. 2019. Winning at the starting
line: Joint network selection and service placement for mobile edge computing.
In IEEE INFOCOM 2019-IEEE Conference on Computer Communications. IEEE,
1459–1467.

[13] Refael Hassin and Asaf Levin. 2005. A better-than-greedy approximation al-
gorithm for the minimum set cover problem. SIAM J. Comput. 35, 1 (2005), 189–200.

[14] Ting He, Hana Kharamfoush, Shiqiang Wang, Tom La Porta, and Sebastian Stein.
2018. It’s hard to share: Joint service placement and request scheduling in edge
clouds with sharable and non-sharable resources. In 2018 IEEE 38th International
Conference on Distributed Computing Systems (ICDCS). IEEE, 365–375.

[15] Wubin Li, Yves Lemieux, Jing Gao, ZhuoFeng Zhao, and Yanbo Han. 2019. Service
mesh: Challenges, state of the art, and future research opportunities. In 2019 IEEE
International Conference on Service-Oriented System Engineering (SOSE). IEEE,
122–125.

[16] Fabio Lopez-Pires, Benjamin Barin, Leonardo Benitez, Satl Alzumeh, and Au-
gusto Amarilla. 2018. Virtual machine placement for elastic infrastructures in
overbooked cloud computing datacenters under uncertainty. Future Generation
Computer Systems 79 (2018), 830–848.

[17] Redowan Mahmud, Satish Narayana Sirrama, Kotagiri Ramamohanarao, and
Rajkumar Buyya. 2019. Quality of Experience (QoE)-aware placement of applica-
tions in Fog computing environments. J. Parallel and Distrib. Comput. 132 (2019),
190–203.

[18] R Timothy Marler and Jasbir S Arora. 2004. Survey of multi-objective optimization
methods for engineering. Structural and multidisciplinary optimization 26, 6
(2004), 369–395.

[19] TM Mitchell. 1997. Machine Learning, McGraw-Hill Higher Education. New York
(1997).

[20] Tao Ouyang, Rui Li, Xu Chen, Zhi Zhou, and Xin Tang. 2019. Adaptive user-
managed service placement for mobile edge computing: An online learning
approach. In IEEE INFOCOM 2019-IEEE Conference on Computer Communications.
IEEE, 1468–1476.

[21] Tao Ouyang, Zhi Zhou, and Xu Chen. 2018. Follow me at the edge: Mobility-
aware dynamic service placement for mobile edge computing. IEEE Journal on
Selected Areas in Communications 36, 10 (2018), 2333–2345.

[22] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto,
and Noah Dorman. 2019. Stable Baselines3. https://github.com/DLR-RM/stable-
baselines3.

[23] Farah Ait Salah, Frédéric Desprez, and Adrien Lebre. 2020. An overview of service
placement problem in Fog and Edge Computing. ACM Computing Surveys (CSUR)
53, 3 (2020), 1–35.

[24] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov.
2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347
(2017).

[25] Mina Sedaghat, Francisco Hernandez-Rodriguez, and Erik Elmroth. 2016. Decen-
tralized cloud datacenter reconciliation through emergent and topology-aware
behavior. Future Generation Computer Systems 56 (2016), 51–63.

[26] Olena Skarlat, Matteo Nardelli, Stefan Schulthe, Michael Borkowski, and Philipp
Leitner. 2017. Optimized IoT service placement in the fog. Service Oriented
Computing and Applications 11, 4 (2017), 427–443.

[27] Petter Sverd, Wubin Li, Eddie Wadbro, Johan Tordsson, and Erik Elmroth. 2015.
Continuous datacenter consolidation. In 2015 IEEE 37th International Conference on
Cloud Computing Technology and Science (CloudCom). IEEE, 387–396.

[28] Xili Wan, Xinxue Guan, Tianjing Wang, Guangwei Bai, and Baek-Yong Choi. 2018.
Application deployment using Microservice and Docker containers: Framework
and optimization. Journal of Network and Computer Applications 119 (2018),
97–109.

[29] Lin Wang, Lei Jiao, Ting He, Jun Li, and Max Mühlhäuser. 2018. Service entity
placement for social virtual reality applications in edge computing. In IEEE
INFOCOM 2018-IEEE Conference on Computer Communications. IEEE, 468–476.

[30] Yuan Zhang, Lei Jiao, Jinyao Yan, and Xiaojun Lin. 2019. Dynamic service
placement for virtual reality group gaming on mobile edge clouds. IEEE
Journal on Selected Areas in Communications 37, 8 (2019), 1881–1897.