Proactive Microservice Placement and Migration for Mobile Edge Computing

Kaustabha Ray, Ansuman Banerjee
Advanced Computing and Microelectronics Unit
Indian Statistical Institute
kaustabharay@gmail.com, ansuman@isical.ac.in

Nanjangud C. Narendra
Ericsson Research
Bangalore, India
nanjangud.narendra@ericsson.com

2020 IEEE/ACM Symposium on Edge Computing (SEC) | 978-1-7281-5943-0/20/$31.00 ©2020 IEEE | DOI: 10.1109/SEC50012.2020.00010

***Abstract*—In recent times, Mobile Edge Computing (MEC) has emerged as a new paradigm allowing low-latency access to services deployed on edge nodes offering computation, storage and communication facilities. Vendors deploy their services on MEC servers to improve performance and mitigate network latencies often encountered in accessing cloud services. A service placement policy determines which services are deployed on which MEC servers. A number of mechanisms exist in literature to determine the optimal placement of services considering different performance metrics. However, for applications designed as microservice workflow architectures, service placement schemes need to be re-examined through a different lens owing to the inherent interdependencies which exist between microservices. Indeed, the dynamic environment, with stochastic user movement and service invocations, along with a large placement configuration space makes microservice placement in MEC a challenging task. Additionally, owing to user mobility, a placement scheme may need to be recalibrated, triggering service migrations to maintain the advantages offered by MEC. Existing microservice placement and migration schemes consider on-demand strategies. In this work, we take a different route and propose a Reinforcement Learning based proactive mechanism for microservice placement and migration. We use the San Francisco Taxi dataset to validate our approach. Experimental results show the effectiveness of our approach in comparison to other state-of-the-art methods.**

***Index Terms*—Mobile Edge Computing, Service Placement, Service Migration, Reinforcement Learning**

I. Introduction

With rapid proliferation of mobile and Internet-of-Things (IoT) devices, the number and sophistication of software services and applications in the IoT space has increased dramatically [1]. Such services often require high processing power and have stringent latency requirements. To supplement such scenarios, devices are typically complemented with cloud services to enhance the Quality-of-Service (QoS) of the user application [2]. However, such a mechanism does not always necessarily conform to QoS requirements of real-time services such as facial recognition, online gaming, video streaming and processing [1] [3]. Mobile Edge Computing (MEC) is a new service provisioning paradigm showing much promise in recent times. The central idea of MEC is to have service providers deploy their services on MEC servers located near mobile base stations. User service invocations are typically routed to, and served from nearby MEC servers on their route as they move around, with improved latency and turnaround times, thereby mitigating high latencies of Cloud Computing.

A service placement policy determines which services are deployed on which MEC servers. Additionally, since users are typically mobile, a key element in service placement is migration where individual service instances may need to be shifted to a different server considering a user's mobility pattern. Indeed, we have a number of possibilities for service placement and migration, considering the different factors (e.g. mobility, server load, latency) that need to be considered. In recent years, several placement and migration policies taking into consideration different scenarios and optimization metrics for application service provisioning in the MEC context have been proposed in literature [4]–[8]. However, most of these solutions need to be re-examined today through a different lens, considering the recent paradigm shift in the application provisioning model, from a monolithic service architecture to the micro-service deployment model, that is being increasingly adopted across the service industry by service providers like Amazon, Netflix [9]. In the service parlance, an application is considered a monolith, wherein all the services comprising the application are packed into a single unit. In the micro-service architecture, the application is split into independent interacting microservices with inter micro-service dependencies. Prior work on service placement in MEC [4]–[8], [10], [11] has predominantly catered to monolithic service applications, and thus need to be revisited to consider and account for the distinctive characteristics of microservices when making placement and migration decisions [12]. This is the main context in MEC that we attempt to address in this work.

Microservice based applications are represented as Directed Acyclic Graphs (DAGs) where vertices denote individual microservices and edges represent microservice interactions which depict the order in which the constituent microservices are executed. Such applications are typically containerized [13], with each microservice hosted in an isolated container. To utilize a microservice, the corresponding container has to be initialized on a server with a non-trivial initialization time. We assume stateful migration of microservices where relocating containers between servers incurs data movement latencies. The problem of service placement and migration in the microservice context in MEC is much more complex than the one for their monolithic counterparts, considering the inter-dependencies that need to be accounted for, while deploying a solution. Indeed, in recent literature, only a small handful of

978-1-7281-5943-0/20/$31.00 ©2020 IEEE
DOI 10.1109/SEC50012.2020.00010

recent proposals [12], [14], [15], to the best of our knowledge, have focused on the microservice model in the MEC context. However, these approaches neither take into consideration non-trivial latencies involved with microservice containers nor user mobility patterns. We consider both in this work in addition to the factors considered by those existing methods.

Traditional placement and migration policies focus on *reactive* placement, i.e., microservice placement after detecting service invocation. Our main proposal in this paper is to develop a *proactive* microservice placement and migration approach by prefetching microservices considering the workflow dependency structure in the application microservice DAG which aids to abate service deployment latencies. Proactively prefetching microservices is a complex task due to: i) the large configuration space of the mobility of devices coupled with microservice interdependencies; ii) the unpredictability of edge servers as as an operating environment due to the dynamic and on demand nature; and iii) the stochasticity of how users initiate service requests while having a myriad of mobility and service invocation patterns.

In this paper, we present our approach for proactive microservice placement and migration (movement) of an already placed microservice in the MEC setup. We focus on applications whose workflows are defined by a linear sequence of microservices. To the best of our knowledge, this is the first work that exploits the microservice dependency task graph structure to prefetch and pre-provision microservices to better meet QoS requirements in MEC. We use a Markov Decision Process (MDP) with rewards to model proactive microservice placement and migration. Further, since the rewards corresponding to each state of the MDP are unknown, we use Reinforcement Learning (RL) to demonstrate how to learn the unknown rewards to effectively deploy and migrate services. In particular, to make effective use of the MDP, we use the Dyna-Q [16] algorithm, which is a combination of model free and model based RL. Additionally, to cater to different traffic patterns, we substantiate Dyna-Q by a heuristic to adapt to varying traffic loads. We further present experimental results of our algorithm in practical scenarios driven by real-world mobility traces of taxis in San Francisco [8] and timing characteristics obtained from the DeathStarBench microservice benchmark suite [9]. Our analysis reveals an average 28% improvement in latency obtained using our proactive approach over the traditional reactive one, for some state-of-the-art MEC microservice benchmark models.

The rest of this paper is organized as follows. Section II presents an example to be used in the rest of this paper. Section III describes the problem formulation. Section IV describes our RL-based approach. Section V describes our implementation along with experimental details. Related work is discussed in Section VI, and finally, Section VII concludes the paper with suggestions for future work.

II. Motivating Example

In this section, we present a motivating example to explain the problem context addressed in this paper. Consider two

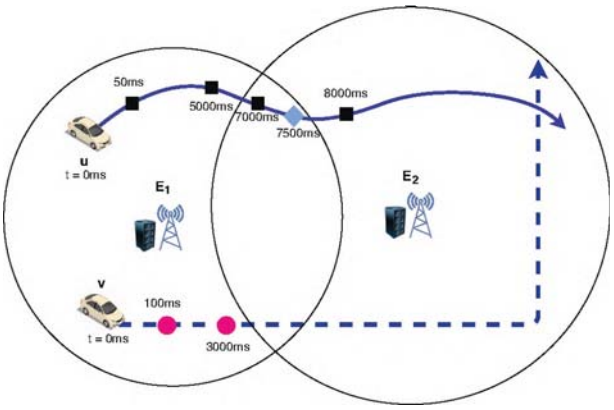


Fig. 1: Microservices Invocations by Vehicular Users

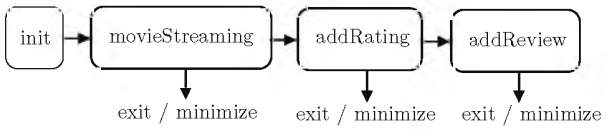


Fig. 2: Movie Streaming Application Microservices

vehicles u and v following the trajectories shown in Figure 1. The passengers of the vehicles access several applications using their smartphones, with each application modeled as an almost linear microservice workflow (with special exit / minimise nodes but no branching in control flow). We select a movie streaming application as a representative use case. A user can either access the microservices in the linear sequence or choose to exit / minimise the application. In the minimized state, microservice containers corresponding to the application are retained on the server while all containers are removed from the server if a user exits an application. The workflow of the application, depicted in terms of its constituent interdependent microservices, is shown in Figure 2. The microservices, hosted in containers, are deployed by service providers on edge servers which have service areas associated with them as depicted by circles around servers E_1 and E_2 in Figure 1.

We now explain the microservice container based provisioning model assumed in this work. When a user invokes a microservice, the container corresponding to the microservice has to be deployed on an edge server if the container is not already present. Additionally, the corresponding service registry has to be updated on a container orchestration system to reflect the deployment state of the microservices. On the other hand, if the container corresponding to the microservice already exists on the edge server, a new task is spawned out of the existing container. The tasks of deploying containers and creating new tasks incur non-negligible latencies. Prefetched microservices, if not used, have no state migration cost. On the other hand, a state-aware migration has to be performed when an user actively using a microservice moves out of the service area of the server where it is hosted and the local computation

Time t	User Action	Server-Service State
$0ms$		No Services Deployed
$50ms$	$u \rightarrow$ movieStreaming	initialize movieStreaming
$75ms$		$E_1 \rightarrow$ movieStreaming
$100ms$	$v \rightarrow$ movieStreaming	$E_1 \rightarrow$ movieStreaming initialize new task for v
$110ms$		$E_1 \rightarrow$ movieStreaming, v_{task}
$3000ms$	v exits movieStreaming	$E_1 \rightarrow$ movieStreaming
$5000ms$	$u \rightarrow$ addRating	initialize addRating
$5025ms$		$E_1 \rightarrow$ addRating
$7000ms$	u minimizes addRating	$E_1 \rightarrow$ addRating
$8000ms$	$u \rightarrow$ addReview	initialize addReview
$8025ms$		$E_2 \rightarrow$ addReview

TABLE I: On-demand Placement of Microservices

state has to be sent to the server from where he is served next. For the sake of simplicity, in the following discussion, we assume it takes $25ms$ to initialize a container, $10ms$ to create a new task in an already existing container and $30ms$ to perform a state-aware migration of a container from one server to another. It may be noted that the timing values used here are just representative ones used for illustrating our problem context. We work with real world microservice timings in our experiments (Section V). In accordance to our objective of proactively placing microservices, we explain in the following subsections how prefetching and proactive deployment can help mitigate some of these latencies, compared to an on-demand service placement scheme wherein service containers are provisioned only after the corresponding microservice is invoked and the container deployed for the first time.

A. On-Demand Microservice Placement

In an on-demand placement scheme, the microservices are deployed only when a user invokes the service. Microservice invocations are depicted by black rectangles and red circles on the trajectories of u and v respectively in Figure 1. At time $t = 50ms$, u invokes the "movieStreaming" service. Since E_1, the nearest server, does not yet host the "movieStreaming" service, the corresponding container is deployed (maybe downloaded from the cloud or nearby servers), initialized and the registry updated. We assume the process takes a total time of $25ms$. At $t = 100ms$, v invokes the "movieStreaming" service. However, in this instance, the corresponding container being already deployed on E_1, only a new task is created incurring an assumed initialization time of $10ms$. At time $t = 5000ms$, u invokes the "addRating" service and incurs an assumed initialization time of $25ms$. At $t = 7000ms$, u minimizes the application on his mobile. Let us assume the "addRating" service is not utilized henceforth. At $t = 8000ms$, the user relaunches the application but instead uses the "addReview" service which requires an initialization latency of $25ms$ at E_2. Thus, the total initialization latency incurred in an on-demand scheme is $25 + 25 + 25 = 75ms$, which adds to the overall latency experience. Table I shows the sequence of events.

B. Proactive Microservice Prefetching and Migration

To mitigate the latencies incurred when deploying services, we propose to proactively prefetch the services, considering the microservice dependency structure. Such an approach allows microservices which are expected to be utilized in the near future to be prefetched and deployed on the MEC server while simultaneously catering to the previously invoked service. To cater to mobility, proactively migrating already deployed services also needs to be examined as we explain later.

Time t	User Action	Server-Service State
$0s$		No Services Deployed
$50ms$	$u \rightarrow$ movieStreaming	initialize movieStreaming
$75ms$		$E_1 \rightarrow$ movieStreaming
$100ms$		$E_1 \rightarrow$ movieStreaming, addRating
$100ms$	$v \rightarrow$ movieStreaming	$E_1 \rightarrow$ movieStreaming, addRating initialize new task for v
$110ms$		$E_1 \rightarrow$ movieStreaming, addRating, addReview, v_{task}
$135ms$		$E_1 \rightarrow$ movieStreaming, addRating, addReview, v_{task}
$3000ms$	v exits movieStreaming	$E_1 \rightarrow$ movieStreaming, addRating, addReview
$5000ms$	$u \rightarrow$ addRating	$E_1 \rightarrow$ addRating, addReview
$7000ms$	u minimizes addRating	$E_1 \rightarrow$ addRating, addReview
$8000ms$	$u \rightarrow$ addReview	state-aware migrate addReview
$8010ms$		$E_2 \rightarrow$ addReview

TABLE II: Proactive Placement of Microservices

Proactively Prefetching Microservices

Consider the following deployment strategy. Initially, when u invokes the "movieStreaming" service, both "addRating" and "addReview" are prefetched to server E_1. Thus, at $t = 135ms$, all three service containers have been initialized on E_1. At $t = 5000ms$, when u invokes the "addRating" service, it no longer incurs the initialization latency of $25ms$. At $t = 8000ms$, u invokes the "addReview" service, however, it is no longer in the coverage area of E_1. Thus, "addReview" which was initialized at E_1, needs to be migrated to E_2. However, since "addReview" was not used, it can be re-initialized incurring a total latency $25+25 = 50ms$. Interleaving service prefetching and execution thus leads to a reduction in initialization latencies. However, the additional latency of $25ms$ incurred while re-initializing the "addReview" service was due to the mobility of u from E_1's service zone to that of E_2. As such, a service placement scheme has to be revisited owing to user mobility. Table II summarizes the timeline of events using prefetching.

Proactive Prefetching and Migration

Let us assume that at $t = 7000ms$, instead of minimizing the application, u continues utilizing the "addRating" service till $t = 8000ms$. Since u traverses service zones while utilizing a service, the service has to be re-deployed once u is in $E2$'s service area. In such scenarios, for the "addRating" service, a state-aware migration has to be performed from E_1 to E_2. Additionally, since the "addReview" service had been proactively deployed on E_1, it has to be migrated as well. Let us assume the state-aware migration is initialized at $t = 7500ms$ depicted by the light blue diamond on u's trajectory and is completed at $t = 7555ms$ since it takes $30ms$ each to migrate the "addRating" service and $25ms$

Time t	User Action	Server-Service State
$0s$		No Services Deployed
$50ms$	$u \rightarrow$ movieStreaming	initialize movieStreaming
$75ms$		$E_1 \rightarrow$ movieStreaming
$100ms$		$E_1 \rightarrow$ movieStreaming, addRating
$100ms$	$v \rightarrow$ movieStreaming	$E_1 \rightarrow$ movieStreaming, addRating initialize new task for v
$110ms$		$E_1 \rightarrow$ movieStreaming, addRating, addReview, v_{task}
$135ms$		$E_1 \rightarrow$ movieStreaming, addRating, addReview, v_{task}
$3000ms$	v exits movieStreaming	$E_1 \rightarrow$ movieStreaming, addRating, addReview
$5000ms$	$u \rightarrow$ addRating	$E_1 \rightarrow$ addRating, addReview
$7500ms$	$u \rightarrow$ addRating	migrate addRating, addReview
$7555ms$	$u \rightarrow$ addRating	$E_2 \rightarrow$ addRating, addReview
$8000ms$	$u \rightarrow$ addReview	$E_2 \rightarrow$ addReview

TABLE III: Proactive Placement + Migration of Microservices

to re-initialize "addReview". The events are summarized in Table III. The total initialization latency experienced by u in this case is $25ms$. However, additional *interleaved* migration latencies for "addReview" and "addRating" are incurred which is *not* perceived by the user. Note that, since the migration is performed proactively, at $t = 8000ms$, u did not have to wait to use the "addReview" microservice. The total speed-up obtained over the on-Demand placement scheme is thus 66%. For real benchmarks, depending on the sizes of the containers corresponding to the microservices and their deployment times, container initialization times can often be in the order of seconds or more, unlike milliseconds as assumed here in the representative use case. For such cases, the speed-up achieved by us can significantly impact user-perceived latencies.

The example above shows the trade-off in latency overhead using proactive prefetching versus reactive on-demand provisioning with data and execution state transmission of the microservices. The challenge is in determining for a given microservice, how many successor microservices to deploy proactively, and more importantly, the target edge servers to deploy them as the user moves and accesses these enroute. An overtly conservative strategy may always proactively prefetch the containers of all successor microservices, whenever any microservice is deployed. However, this may at times turn out to be wasteful in terms of resources needlessly blocked on the edge server by the prefetched containers, if these microservices are actually not invoked at that location. On the other extreme, a fully reactive policy does not help as well since such a policy would lead to latency hits in each microservice invocation. The challenge is in being able to predict the user service invocation pattern as a function of his mobility, such that better prefetching can be carried out. Our objective here, is to learn and synthesize the optimal proactive prefetch, deployment and migration schedule, given a microservice workflow.

III. Formal Model

In this section, we formally describe the proactive placement and migration problem and formulate an MDP model.

A. Problem Definition

The MEC system comprises a set of edge servers denoted as $E = \{E_1, E_2, ..., E_p\}$, where each E_i is associated with a service radius r_i. We have a set of users $U = \{u_1, u_2, \dots u_q\}$ and a set of applications $A = \{A_1, A_2, \dots A_r\}$. An application $a \in A$ comprises a linear workflow of microservices $S = \{s_1, s_2, ..., s_n\}$ with special exit nodes, denoting the order of invocation of microservices. We use a model similar to [11] where we do not consider a back-end cloud, instead consider only a set of edge servers. A location is defined as the latitude and longitude coordinates of the entity under consideration. Servers have fixed locations while users are free to move and their coordinates vary over time. We consider a discrete time model, as in [8]. Let us consider a user $u \in U$ where $u(t)$ denote the user's current location at time slot t. We denote the set of active microservices associated with $u(t)$ as $h(t)$ and the corresponding location as $l(t)$. We assume that the set $h(t)$ can only be co-located at a single $l(t)$, i.e., all microservices in $h(t)$ will be placed by our scheme on a single edge server, as discussed later. We also assume that all latencies for deploying containers, instantiating tasks and migrating containers are strictly additive. As discussed later, the policy agent designed by our RL approach governs the placement and migration of microservices on the edge servers. At each time-slot, it observes $u(t), h(t)$ and $l(t)$, and decides on placing/migrating the relevant services $h(t+1)$, so that the user experiences the best latency values. At the beginning of each time slot, our policy agent can choose from one of the following options:

- Proactively Placing Microservices : At any location $u(t)$, when u invokes a microservice $s_i \in S$ whose successor microservices are $\{s_{i+1}, s_{i+2}, ..., s_n\}$, the agent selects the nearest server $E_i \in E$ to deploy s_i along with j successor microservices, i.e., $\{s_{i+1}, s_{i+2}, ..., s_{i+j}\}$, where $0 \leq j \leq n - i$, and updates $h(t+1) = h(t) \cup s_i \cup s_{i+1} \cup s_{i+2} \cup ... \cup s_{i+j}$. It incurs a deployment cost $c(r)$, where c is a non-decreasing function of r, the resource requirement of the microservices to be deployed. We relate $c(r)$ with the MDP reward function as explained in Section IV.
- Microservice Migration : When u moves away from the service zone of the server on which $h(t)$ was deployed, the agent performs a state-aware migration to re-deploy the microservice currently in use to the edge server nearest to the user's new location. Other microservices which are not currently in use are re-initialized. In this case, $h(t) = h(t+1)$, but $l(t) \neq l(t+1)$. Performing such a state-aware migration for active services incurs a cost $m(r)$, where m is a non-decreasing function of r whereas re-initialization incurs cost $c(r)$.

For the sake of simplicity and ease of illustration, we first present the problem model for a single user accessing a single application. We relax these requirements later in Section IV where we build on this to cater to multiple users accessing multiple services simultaneously. In the following, we use an

MDP to formally model prefetching and migration.

B. State Representation of MDP

Formally we define our policy agent for proactive placement and migration as an MDP, as follows:

Definition III.1. *A MDP is a tuple ($\mathcal{S}$, T, P, A, R_a) where :*

- *$\mathcal{S}$ is a finite set of states*
- *T is the transition function from $s \in \mathcal{S}$ to a subset $\{s_1, s_2, \dots s_n\} \in \mathcal{S}$ denoting all successor states of s.*
- *$P : S \times S \rightarrow [0, 1]$ denotes the probability distribution on transitions such that for all states $s : \sum_{s' \in S} P(s, s') = 1$*
- *A is the set of actions, i.e, from a state s, upon an action $a \in$ A(s), the successor state s' is determined by T(s)*
- *$R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s' , due to action $a \in A$.*

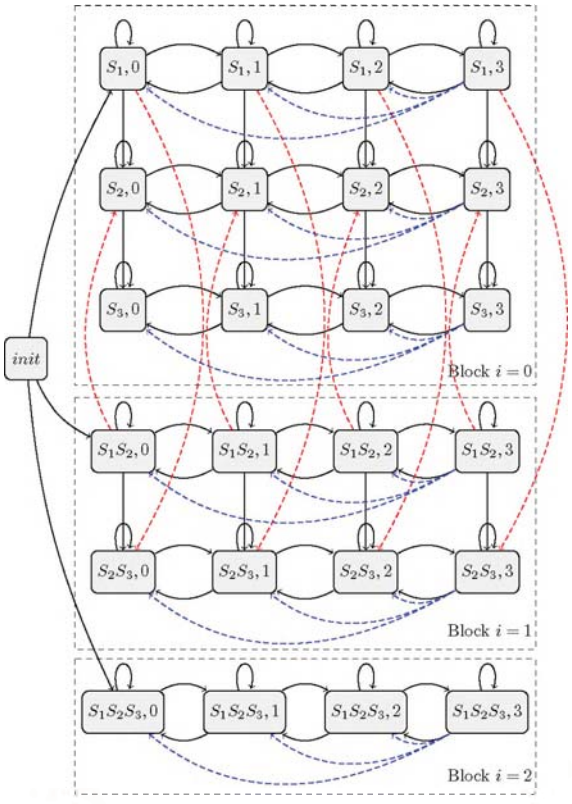


Fig. 3: Proactive Service Placement and Migration MDP

Each state of the MDP for a user u is represented by a vector $[service, distance]$. In each state, $service$ represents $h(t)$, while $distance$ represents the distance between the location $u(t)$ of u and the location of $h(t)$, i.e., $l(t)$. We represent the distance as an abstract measure similar to [8]. In the MDP, $distance$ is a mapping from a concrete measure such as the Euclidean or Manhattan distance to the abstract measure. The server E_i corresponding to $l(t)$ is associated with a maximum service area demarcated by the radius r_i from its location. Hence, there is an upper bound on the $distance$ representation in the MDP. The upper bound denotes the distance between the location of the server and a coordinate located on the circumference induced by the service radius r_i of the server E_i. Further, since each server can have a different service radius, the upper bound distance is normalized in the range $[0, k]$ where k is a user defined parameter. However, since such an interval is continuous, the interval $[0, k]$ is discretized at intervals of 1. Hence, all possible values of distances are $0, 1, \dots, k$. The concrete distance from $u(t)$ to $l(t)$ is thus mapped to the discretized interval distance set as follows: distance between $u(t)$ and $l(t)$ in the continuous interval $[0, 1)$ is mapped to $k = 0$, distance in the continuous interval $[1, 2)$ is mapped to $k = 1$ and so forth, where $[0, 1)$ denotes the continuous interval inclusive of the lower bound 0 and exclusive of 1. As such, $distance = k$ denotes the scenarios when the distance between $u(t)$ and $l(t)$ exceeds k. The $[service, distance]$ vector thus, uniquely identifies the microservices which have been proactively prefetched and the distance between a user u and the server $E_i \in E$ where the prefetched microservices are hosted. The MDP structure embodies all possibilities for prefetching discussed earlier, considering a given microservice workflow. In the following, we use our example application with 3 main constituent microservices to illustrate the MDP construction for user u.

Example III.1. *Figure 3 depicts the MDP for the Movie Streaming Application accessed by user u in Section II which comprises 3 microservices "movieStreaming", "addRating" and "addReview", represented as S_1, S_2 and S_3 respectively. The "movieStreaming" microservice is initialized on server $E_1 \in E$ upon invocation of the application by u. We consider the Euclidean distance measure as an illustration. The MDP assumes $k = 3$. Let us suppose the Euclidean distance between the location of u and E_1 evaluates to a value between 0 and 1. The state $[S_1, 0]$ denotes such a scenario. Thus, the distance identifier of a state vector abstractly represents a concrete distance interval. Similarly, $[S_1, 1]$ is the scenario when the distance between u and E_1 is between 1 and 2. Since $k = 3$, the state $[S_1, 3]$ denotes the scenario when u moves to a location when the distance between u and E_1 exceeds 3. Such states only correspond to a single service being deployed at a discrete time-point. The state $[(S_1, S_2, S_3), 0]$, on the other hand, exhibits all three microservices being deployed on a server at a distance between 0 and 1 from u.* ■

C. Proactively Prefetching Microservices

The MDP can be viewed as comprising several blocks. Each block corresponds to prefetching i $(0 \leq i < n)$ services corresponding to the linear workflow, where n is the number of microservices in the application. The case $i = 0$ corresponds to reactive deployment, where only upon invocation, the respective service is initialized. The case $i = n - 1$, on the other hand, corresponds to an over conservative strategy where all services comprising the workflow are deployed upon

application initialization. When the value of i ranges between 1 and $n-1$, i consecutive services are prefetched.

Example III.2. *The MDP constructed in Figure 3 comprises 3 blocks, each depicted with a dashed rectangle. The first block corresponds to reactive microservice deployment while the remaining blocks represent prefetched deployment. The state $[(S_1, S_2), 0]$, within the block $i = 1$ denotes the scenario where microservices S_1 and S_2 are prefetched while the state $[(S_1, S_2, S_3), 0]$ within the block $i = 2$ denotes the scenario where the microservices S_1 , S_2 and S_3 are prefetched.* ■

D. Transition Representation of MDP

Transitions from the $init$ state denote the number of microservices to initially prefetch with one transition to each block.

Example III.3. *The $init$ state has outgoing transitions to $[S_1, 0]$, $[(S_1, S_2), 0]$ and $[(S_1, S_2, S_3), 0]$ denoting proactive prefetching of $0, 1$ and 2 microservices respectively.* ■

Other transitions occur when the state of u changes and can be broadly classified into two types: transitions within a block and transitions between blocks.

Intra-Block Transitions: Transitions within a block occur only when a user moves from one location to another or when there is a transfer of control from one microservice to its subsequent microservice in the application workflow.

Example III.4. *Let us suppose the initial Euclidean distance between u and E_1 was between 0 and 1. Such a scenario is denoted in the MDP by the state $[S_1, 0]$. Along the course of u's path, let us suppose the Euclidean distance measure at some time-point exceeds 1. This change in u's location is represented by the transition to $[S_1, 1]$. Continuing along its trajectory, as long as the Euclidean distance between u and E_1 lies between 0 and 1, it remains in the state $[S_1, 1]$ denoted by the self transition. The transition from $[S_1, 0]$ to $[S_2, 0]$ denotes the transfer of flow of control in the microservices workflow from S_1 to S_2 while the distance between u and the server where S_2 is deployed remains within 1. Note that however, transitions denoting changes to both distance and service invocation trajectory can not happen. For example, there is no transition from $[S_1, 0]$ to $[S_2, 1]$. This is because prefetching services is carried out with respect to the current relative locations of the server and the user. Thus, a simultaneous change in both is only accounted for by first updating the location followed by the service invocation.* ■

Inter-Block Transitions : Transitions between blocks represent the possibility of proactively prefetching variable number of microservices. Consider a state S of the MDP where the $services$ component of the state vector comprises $(S_m, \ldots, S_n)$. In the event of the transfer of flow of control of microservices from S_m to S_{m+1}, outgoing transitions from S portray choices in the number of proactively prefetched microservices by transitions to all states in the MDP whose identifier begins with S_{m+1}. Such transitions are represented in Figure 3 by red curved dashed lines.

Example III.5. *The outgoing transition from state $[S_1, 0]$ to state $[(S_2, S_3), 0]$ of block $i = 1$ denotes the situation when u experiences a flow of control transfer from S_1 to S_2, and both S_2 and its successor S_3 are prefetched to the server. Note that the other choice of deploying S_2 only is already covered in block $i = 0$. Thus, the transition from S_1, S_2 in block $i = 1$ to S_2 block $i = 0$ demarcates the scenario when u experiences the same flow of control from S_1 to S_2, but the agent decides not to proactively fetch any other service. However, note that transitions such as those from $[S_1, 0]$ to $[(S_1, S_2, S_3), 0]$ are not possible since the latter demarcates prefetching all three services upon invocation of S_1 while the former denotes S_1's deployment with no other service prefetched. Also note that transitions such as those from $[(S_2, S_3), 0]$ to $[S_3, 0]$ are not possible. In state $[(S_2, S_3), 0]$, both S_2 and S_3 have already been prefetched. Such a transition would only depict a flow of control from S_2 to S_3 which does not necessitate a further prefetching decision.* ■

E. Migration of Microservices

Migrations occur when u moves from one service zone to another. In such a scenario, if the new service zone has only one server associated with it, the microservices are migrated to that server. Otherwise, if multiple choices of servers are available, the nearest server is selected. We consider capacity constraints greedily as explained later in Section IV-B. When u crosses the boundaries of service zones, on migration of services, its movement is represented in the MDP by transitions from states whose distance vector component is k to $\{0, 1, \ldots, k-1\}$. We assume that the MEC servers are distributed such that each area is in the coverage of at least one MEC server, hence a target server always exists.

Example III.6. *Let us assume that along u's trajectory, at some time-point, the Euclidean distance between u and E_1 is between 2 and 3 denoted by the state $[S_1, 2]$. When u traverses further away from the server, exceeding E_1's service radius, the state of the MDP is updated to $[S_1, 3]$. S_1 is then migrated to the nearest server say E_2. In such a scenario, the Euclidean distance between E_2 and u is re-calculated and the new corresponding abstract distance is represented in the MDP by blue dashed transitions to $[S_1, 0], [S_1, 1], [S_1, 2]$ according to the re-calculated distance and mapped appropriately using the abstract distance representation.* ■

Additionally, since users can exit the application at any stage, we add an extra state, $exit$ to the MDP demarcating that the user has exited the application. From all states excluding the $init$ state, transitions are drawn to this state signifying the event of an application closure. The $exit$ state and the corresponding transitions are not shown in Figure 3 for brevity. Further, we do not require any explicit encoding to denote the minimized state of an application since prefetched services are retained on MEC servers while minimized and evicted only upon application exit.

The MDP built above embodies the underlying solution space for our problem context, accounting for all prefetch and deployment possibilities. While on one hand, the states represent the different service user deployments, the transitions represent the corresponding possibilities, induced by user movement and possible service invocations. Once the MDP is built, we now proceed to determine the rewards associated with our MDP, based on the movement and service invocation patterns of users. This is helpful for deciding the proactive prefetching strategy, i.e. for which microservice, how many successors to prefetch at which location and deploy on which edge server. We formulate this problem as a Reinforcement Learning (RL) problem where the agent explores interactions with the environment to learn the rewards for the best strategy.

IV. Reinforcement Learning Solution

Our RL formulation is along the lines of a standard RL framework [16] where an agent continuously interacts with the environment, receives and interprets rewards and attempts to synthesize the optimal strategy. We use the Dyna-Q [16] RL algorithm, a combination of model based and Q-Learning, a model free RL paradigm. The Dyna-Q Algorithm is summarized in Algorithm 1.

Algorithm 1: Dyna-Q

```
1  Initialize Q(s,a) and Model(s,a), ∀s ∈ S, ∀a ∈ A(s)
2  while true do
3      s ← observe the application state
4      a ← ε-greedy(s,q)
5      Observe the next state s' and the reward obtained
6      Update Q(s,a) using Equation 1
7      Model (s,a) ← r,s'
8      for i = 0...n do
9          s ← random state previously observed
10         a ← random action previously taken in s
11         r,s' ← Model(s,a)
12         Update Q(s,a) using Equation 1
```

Q-learning essentially estimates the optimal Q-function, Q, by its sample averages. In this paper, we consider the simple ϵ-greedy action selection method: at any decision step i, with probability ϵ, Q-learning chooses a random action to improve its knowledge of the application, whereas, with probability $1-\epsilon$, it chooses the action greedily by exploiting its knowledge about the application, i.e., a = $\text{argmax}_a\ Q(s,a)$. Most of the time, the ϵ-greedy policy selects the best known action for a particular state, while it favors the exploration of sub-optimal actions with low probability. At the end of each time slot i, $Q(s,a)$ is updated as follows :

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma\ \text{argmax}_a Q(s',a) - Q(s,a)\right] \quad (1)$$

where α is the learning rate assigned to the agent. The equation updates the Q-value of state s by determining the action corresponding to the highest Q-value among all successor states ($\text{argmax}_a Q(s',a)$), which is discounted by γ and updated according to the reward r observed from the environment. Differently from Q-learning, Dyna-Q aims to speed up the learning process by simulating the system interaction with the environment. At run-time, Dyna-Q observes the application state and selects an adaptation action using the estimates of $Q(s,a)$, as Q-learning does. At the end of the time step i, DynaQ exploits a sampled model of the system, $Model(s,a)$, where $Model(s,a)$ refers to our MDP model as discussed in Section III-B, to simulate the interaction between the application and the environment (lines 8 - 12). Dyna-Q updates $Model(s,a)$ at runtime, by storing the next state s' and reward r for the explored state-action pair (s,a) at line 7. Dyna-Q updates the Q-function akin to Q-learning using (1) and resorting on the state-action pairs previously observed.

In our context, in a real-world scenario, multiple users access multiple applications simultaneously. Each application A_i is thus associated with its own MDP M_i as described in Section III-B. Each M_i has a corresponding reward r_i whose initial value is set to 0. When a user u_j invokes an application A_i, the corresponding MDP M_i is assigned to the user. This is used with u_j and M_i to execute the required prefetching / migration actions. When the agent executes an action, it receives rewards from the environment. Thus, rewards are assigned whenever there exists a transition to denote state change. The reward function denoted by R is a weighted combination of resources consumed by prefetched microservices actually utilized and prefetched microservices not invoked by the user.

$$R = \sum_{\mu \in \mu_{used}} [\mu * c(\mu_{resources})] - \sum_{\mu \in \mu_{unused}} [\mu * c(\mu_{resources})] \quad (2)$$

μ_{used} and μ_{unused} are sets of indicator variables representing the set of prefetched services which have been invoked and not invoked respectively while $c(\mu_{resources})$ represents the resource costs of microservice μ according to cost function $c(r)$. This reward is calculated whenever the user invokes a microservice in the application workflow following which the Q-values are updated. Additionally, the Dyna-Q algorithm simulates interactions (Lines 8-12) to represent real-world interactions. A positive reward is assigned for services which are prefetched and utilized by the user while a negative reward is assigned to unused services. Such a reward function embodies migration decisions as well. Negative rewards signal the agent to lean towards prefetching a lower number of services thereby reducing migration costs while advocating a reduction in number of unnecessary migrations. As such, for rewards corresponding to migrations, the same reward function is used, with the migration cost function $m(r)$ instead of $c(r)$.

Although the formulation allows us to characterize user service invocation pattern as a function of mobility using a distance based MDP which serves to be space efficient, it neither effectively quantifies the prefetch policy influenced by the network load characteristics nor cater to capacity constraints of servers. To characterize traffic load distribution and capacity constraints, we propose a heuristic in the following discussion.

Algorithm 2: Proactive Prefetching and Migration

```
   Input : low, medium, l, servers
1  Initialize K-D Tree with servers
2  foreach timepoint t do
3      foreach user u do
4          curr_loc ← updated location of u
5          server_l ← location of server assigned to u
6          d ← distance between server_l and curr_loc
7          if d ≥ server_l.coverage then
8              newserver ← query K-D tree for location
                 of nearest server to u
9              server_l ← newserver
10             nd ← distance between newserver and u
11             update MDP state according to nd and
                 migrate microservices greedily
12         else
13             nd ← new distance between server_l and u
14             update MDP state according to nd
15         if current application/service status is different
             from previous time-slot then
16             if u has invoked A_i then
17                 ld ← load in l-hop neighbourhood of u
18                 if ld ≤ low then
19                     u.M ← M_i^low
20                 else if ld ≤ med then
21                     u.M ← M_i^med
22                 else
23                     u.M ← M_i^high
24             a ← action selected with Dyna-Q for u
25             if a exceeds server capacities then
26                 s_i, s_j, ... s_n ← services corresponding
                     to a
27                 foreach s ∈ s_i, s_j, ... s_n in order do
28                     allocate s
29                     if allocate s is NULL then
30                         s_i, s_j, ... s_k ← services
                             allocated successfully
31             set MDP to state corresponding to
                 s_i, s_j, ... s_k successfully allocated
32             update Q with reward for action a using
                 Equation 2 in MDP corresponding to u
33             upon application exit update corresponding
                 MDP for A_i with highest rewards
```

A. Catering to traffic variation

The number of users invoking different applications may actually vary over time. In such a scenario, the number of users can play a crucial role in determining how the RL agent is trained. This is because, if an agent receives a high reward for an action in a low traffic load environment, i.e., when the number of application users is low, it may end up receiving a low reward for the same action in a high traffic environment since a single application proactively prefetching multiple microservices can lead to clogging of resources for other users depending on the stochastic arrival of microservice invocations. Such variance in rewards can confuse the RL agent [17]. To overcome this difficulty, instead of assigning a single MDP M_i to A_i, we associate three MDPs M_i^{low}, M_i^{med} and M_i^{high} to each A_i denoting low, medium and high traffic loads respectively. When u_j invokes A_i, depending on the current load distribution in the k-hop neighbourhood of u_j, the corresponding MDP is assigned to u_j. Such a mechanism described in Algorithm 2 allows us to separately characterize traffic workloads and allows the RL agents to effectively use the exploitation phases to learn traffic-aware policies. Note that the MDP and DynaQ-Tabulation can be stored on disk of the orchestrating entity and instantiated upon the invocation of an application corresponding to a user.

B. Catering to Capacity Constraints

The MDP discussed in Section III-B, only comprises information pertaining to the location of users and services currently in operation. We do not encode information pertaining to capacity of servers or request-server bindings in the MDP. In a realistic setting, MEC servers do not possess infinite elasticity bounds, and therefore, capacity constraints need to be adhered to. We address such capacity constraints heuristically without the requirement of having to encode any additional information in the MDP states. When the MDP agent selects an action corresponding to deploying $s_1, s_2, \ldots s_n$ microservices proactively, if all such services can be deployed without violating capacity constraints, the agent proceeds to deploying all such prefetched microservices. On the other hand, with capacity constraints not withholding, the algorithm greedily deploys the containers pertaining to the prefetched services in order of the application microservices workflow until the capacity constraints are exhausted. It then updates the MDP state corresponding to the services deployed greedily. Allocation upon migration is carried out in a similar manner.

Algorithm 2 initializes a K-D Tree [18] with the location of the MEC servers (Line 1). This allows efficient queries to locate the nearest server for users. It proceeds to update the locations of the users in the current slot (Line 4) and identifies users which have moved out of the service zones of the servers to which they were assigned. For users which have indeed moved out, the nearest server in the current location is identified by querying into the K-D tree (Line 8). Accordingly, the MDP is updated to the value of the normalized distance from the currently assigned server and the updated location of the user (Lines 7-14). It proceeds to check the service usage status of the users in the current time slot (Line 15) and calls the Dyna-Q algorithm. Depending on the action selected by the Dyna-Q algorithm and the subsequent greedy placement (Lines 25-30), the state of the MDP and the Q-values are updated with the generated rewards (Line 31-32). In the scenarios that the prefetched microservices can not be accommodated on the server, the algorithm greedily selects the set of microservices in the workflow sequence of the application which can be allocated, as explained above. Once it exhausts the server capacities (Line 29), it proceeds to set

the MDP state to reflect the microservices which have been successfully allocated greedily (Line 31). Once a user exits the application, the MDP is updated accordingly (Line 32).

V. Experimental Evaluation

We perform extensive experiments to show the efficacy of our approach, and compare its performance against a) on-demand placement, and b) MCAPP-IM [12], an algorithm for placement of applications with multiple components. In the following, we describe our experimental setup and the results.

A. Experimental Setup

To the best of our knowledge, there are no real-world MEC implementation workload traces that are publicly available and sufficiently representative of our problem context. Therefore, for our experiments, we generate synthetic workloads using some real-world taxi traces and microservices from a public domain microservice benchmark suite. In the following, we describe how we generate the problem instances for our simulation experiments and describe the experimental setup.

1) MEC Server Locations and User Trajectories: We consider a discrete time slotted system in which the locations of users in the network may change from one time slot to another. We perform simulation with real-world mobility traces of taxis in San Francisco [8], available publicly, collected over different time points during the day where different numbers of taxis operate at different times of the day. We model each taxi as a user invoking service requests along its trajectory. For MEC server location, we use the 'Existing Commercial Wireless Telecommunication Services Facilities in the San Francisco' dataset [19], which is also available publicly, as MEC server locations. The coverage area of each MEC server is randomly generated (while ensuring full coverage of the city area under study) as in [1], [20]. We assume k as 3 for our experiments as well unless specified otherwise. The generated coverage areas of all servers are normalized in $[0, 3]$ when translated to the MDP representation. However, this dataset only comprises locations of towers confined mostly to a large segment of the San Francisco area. The San Francisco taxi dataset, however, comprises taxi trajectories distributed over the entire city of San Francisco. From the dataset, we extract for each taxi, the coordinates confined to the region of MEC server locations. As a result of such filtering, some of the taxi trajectories include abrupt changes in coordinates when such taxis in the original dataset move out of the region considered in the MEC server dataset. We treat such scenarios as users exiting application usage when we assign service invocations to users. As a result, such abrupt changes do not impact the flow of microservices in the workflow in the generated dataset. In Figure 4, we use red circles to depict MEC server locations and blue circles to show the trajectory of a sample taxi.

2) Service Invocations: We use microservices from the 'Media Microservices' application of the 'DeathStarBench' benchmark suite [9]. This suite presents microservice based applications in domains such as social media, e-commerce, banking, movies and drone swarms and has been used in a range of studies such as hardware and networking implications of microservices and performance debugging of cloud microservices [21]. We use the size of the container of each microservice as its representative resource requirement. We fetch the corresponding containers from Docker Hub [22] initially. We then note the starting times of each containerized microservice by invoking a fresh docker container start after stopping all running containers. We use these times as the deployment time of containers on MEC servers. Further, we generate running times in the range $(1/3 \times \text{starting-time} \pm \lambda)$ to simulate representative times of creating tasks from already existing containers. We use these values as representative times since the DeathStarBench benchmark uses a composition driven approach where multiple microservices are used to execute a task and as such, each task comprises a composite worflow unlike our model where we assume a single container being associated with a task. $1/3$ is chosen since creation of tasks involves lower computation times as compared to deploying dedicated containers and λ is assigned a random value between 0 and 0.05 to simulate random runtime deviations. To simulate migration times, we add these task creation times to the container deployment time. These values are used as $c(r)$ and $m(r)$ for the reward function. To each taxi trajectory obtained from the dataset, we assign such service invocations randomly at different discrete time slots. We assign the invocations considering the distance between the location of the taxi in the previous time slot and the current time slot. If the distance exceeds a threshold value, we treat such a slot as a fresh invocation of an application, since we assume that the applications exit when the user leaves the area or closes the application. Additionally, while extracting the trajectories from the original taxi dataset, since we only use a portion of the San Francisco area, there are several time slots, where the location of the taxis in the subsequent discrete time slot goes outside the considered San Francisco area and are thus not a part of the extracted dataset. We treat such change in coordinates also as fresh invocations of an application. In the event that the distance does not exceed this threshold city

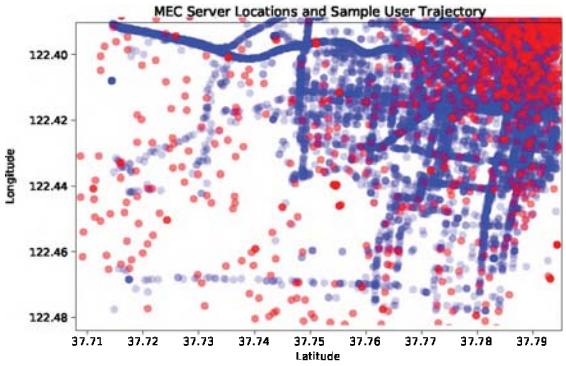
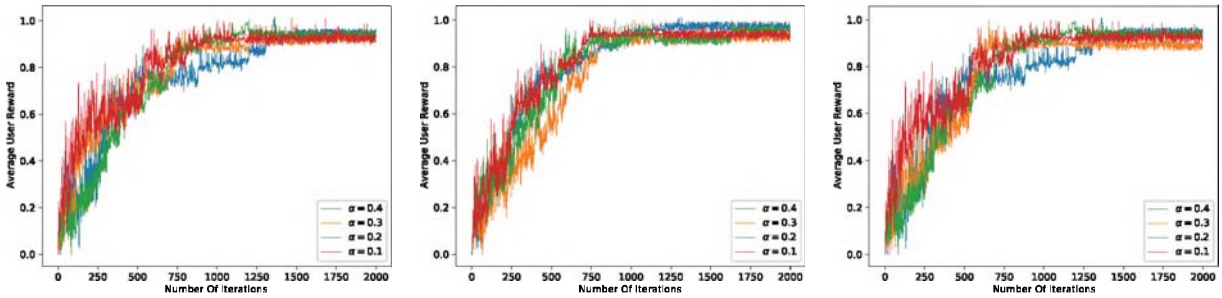
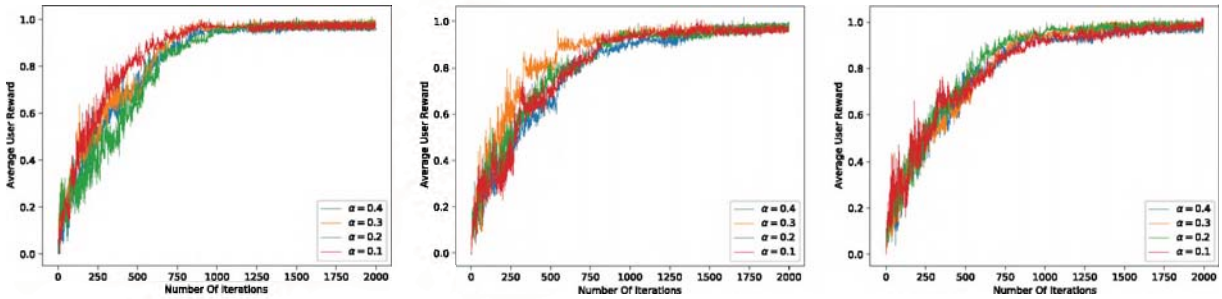


Fig. 4: Extracted MEC Server Locations and Taxi Trajectory

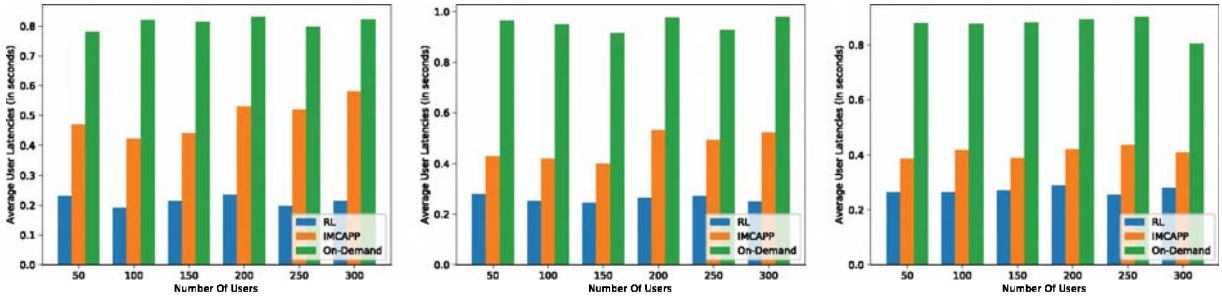


(a) Number of Users = 50 (b) Number of Users = 100 (c) Number of Users = 150



(d) Number of Users = 200 (e) Number of Users = 250 (f) Number of Users = 300

Fig. 5: Average Reward Accumulation with Variable Number of Users



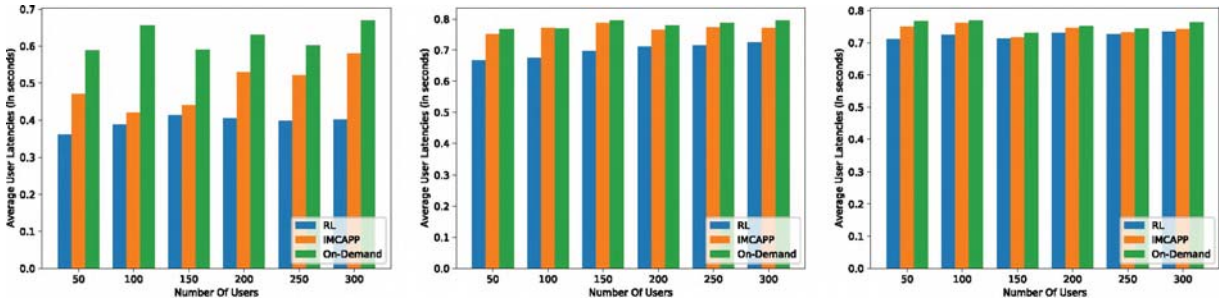
(a) $n = 4$ (b) $n = 8$ (c) $n = 12$

Fig. 6: Average User Latency for Variable Number of Application Microservices

area, we randomly generate invocation/minimization decisions along the linear workflow structure of the application. Upon invocation of the last microservice in the workflow, a fresh invocation of a random application is considered.

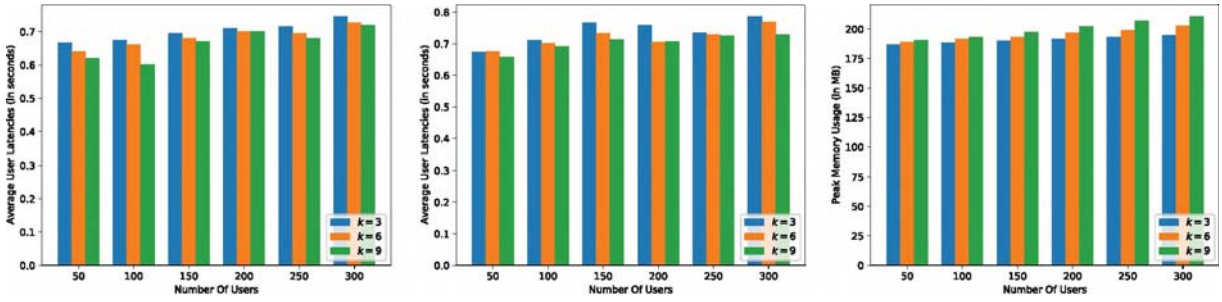
B. Results and Discussion

We compare the performance of our approach with that of On-Demand service provisioning to demonstrate the impact of proactively prefetching services on user experienced latency. Additionally, we compare with the MCAPP-IM algorithm [12]. The MCAPP-IM algorithm formulates the problem as an online bipartite matching problem supplemented with a greedy local search technique between application components and edge servers. However, they do not consider proactively deploying any of the application components. MCAPP-IM runs at each time-slot and the result of the matching thus obtained is the service-server binding. The MCAPP-IM algorithm does not consider coverage area zones of MEC servers. All experiments are performed in Python 3.7 with the K-D tree implementation of the SciPy library on a machine with an Intel Core i5 8250U processor and 16 GB of RAM. We set the value of the exploration parameter ϵ of Dyna-Q to 0.2.



(a) Server Resources = 130% (b) Server Resources = 100% (c) Server Resources = 65%

Fig. 7: Average User latencies for Variable Traffic Distributions



(a) Varying k for $n = 4$ (b) Varying k for $n = 8$ (c) Memory Usage for Varying k

Fig. 8: Effect of Varying k on Latency and Memory Usage

We vary the total number of users from 50 to 300 at an interval of 50. For each scenario, we additionally vary the learning rate α as $\{0.1, 0.2, 0.3, 0.4\}$. In Figure 5, we plot the average reward of the policy agent against the number of iterations where each iteration refers to one discrete time-slot where the average reward value is normalized between 0 and 1. As can be inferred from the figure, different learning rates produce a variation in the rewards accumulated by our agent. With a higher learning rate, a wider variation in reward accumulation is observed in general, since a higher learning rate corresponds to a greater weightage in Q-value updation in each iteration. It is however interesting to note that for a high number of users, specifically in Figure 5f, the deviations obtained are rather minor as compared to others. This is due to the fact that in a high traffic environment, the MEC servers are resource constrained and hence lean towards more reactive deployments, justifying our design objective.

In Figure 6, we plot the average user latencies as we vary the number of microservices in the application workflows. We consider representative applications involving $4, 8$ and 12 microservices (denoted as n in the figure), from the 'DeathStarBench' benchmark suite [9]. We measure the performance of the algorithms as we vary applications with the aforementioned number of microservices. There is no definite increase/decrease pattern with latencies as the number of users are increased. This is expected since with adequate availability of server resources, a higher number of users does not lead to greater contention. Proactively deploying microservices leads to an overall benefit of the latency perceived by the user in all scenarios as observed. Since MCAPP-IM does not involve server coverage areas, it incurs a lower latency as compared to the On-Demand scheme. However, our RL based algorithm being specifically catered to proactive deployment, is able to outperform MCAPP-IM in terms of average latency experienced by users.

We next analyze the effect of the traffic load on the performance of the various algorithms. We vary the number of available MEC servers while keeping the number of users fixed to simulate availability of server resources. We consider scenarios where we fix the total server resources at $130\%, 100\%$ and 65% of the requisite total resource consumption of the users. A server resource percentage of 100% denotes the scenario in which the resource availability of the server can cater to exactly the number of users fixed. In Figure 7, we plot the average latencies of the three algorithms in each scenario. With high availability of server resources, our algorithm obtains lower average user latencies as compared to MCAPP-IM. The average improvement over MCAPP-IM is around 44%.

However, as the resource availability of servers is decreased, with high traffic loads, the benefits of proactive deployment are far lower, at an average of 11%, as observed in Figures 7a, 7b and 7c. This is because in such a resource constrained environment, the agent favors lesser proactive prefetching of microservices. Such a scenario, for an application which comprises 3 microservices, as in the example in Section II, would correspond to the MDP in Section III-B executing most of the transitions in Block $i = 0$ prefetching a small number of microservices. This supports our intuition and justifies our design objectives as well.

We now analyze the effect of k on the performance of our RL approach. k is a user defined parameter which specifies the normalization constant for coverage areas of each server. We experiment with values of $k = \{3, 6, 9\}$. As the value of k is increased, the size of the MDP increases. As a result, the memory consumption increases which is validated in Figure 8c. Additionally, we vary k in the same range for another application which comprises 8 microservices instead of 4. Figure 8b depicts that there is no definite increase/decrease pattern with respect to varying n and the number of users as observed previously when we studied the impact of n on latencies. It is however interesting to note that, as the value of k is increased, there is an improvement in latency for several scenarios in Figures 8a and 8b. Increasing the value of k, allows us to represent the discretized coverage areas of servers more precisely and hence, the agent can make decisions more accurately. Thus, since k acts as a parameter which controls the degree of representation of coverage areas of servers, varying k can have an impact on the overall latency incurred. However, larger values of k incur a greater cost of representation in memory thus presenting a trade-off. Further, in Figure 8a, only a marginal improvement is observed when k is varied for the scenario when the number of users is 200. Such scenarios can indeed happen if the dataset does not incorporate scenarios spanning the entire action space of the agent thereby rendering some actions unexplored.

VI. Related Work

Service placement and migration have received a lot of attention recently in context of monolithic services [4], [6]. Our approach is related to prior research, however in the context of microservice architectures [9] in an MEC environment. We discuss below related approaches and our novelty.

- *Computation Offloading:* Offloading in Mobile Cloud Computing [23]–[25] and Mobile Edge Computing [3], [26]–[30] have both been studied extensively concerning what/when/how to offload workloads from handheld devices to the cloud or edge [31]. Wang et al. [32] deal with minimizing each MEC server's energy consumption while satisfying QoS requirements. The authors [33] consider scenarios where migrations are allowed between edge servers while keeping the objective the same as in [32]. However, offloading decisions concern moving workloads irrespective of whether the requisite services are available on the servers or not [31]. Service Placement, on the other hand, concerns decisions of deploying services on servers which is crucial in MEC due to stringent latency requirements and limited server capacities [31].
- *Service Placement:* Service Placement has received attention in a myriad of formulations including the static [4], [34] and dynamic [6], [7], [35]–[37] service context. In [4], the authors derive an approximation by incorporating rewards which are awarded when user requirements are honoured. In [7], the authors formulate a time-slotted model and develop a polynomial approximation by jointly optimizing service placement and request scheduling, i.e., which user requests are to be routed to deployed services. The works in [5] [31] additionally consider data transfer and availability for making placement decisions. In [38] and [39], the authors also consider base-stations collaborating with each other. In [36], the authors consider multi-network scenarios as well and optimize service placement by incorporating network communication costs. The authors in [37], consider a Virtual-Reality based application and study service placement strategies optimizing for the same. The authors in [37], consider placement of a Virtual-Reality application demonstrating gains in an application specific scenario. Similarly, [35] considers service placement in Software Defined Networks. All of the above works however, explore monolithic service structures. In contrast, we consider a microservice architecture with dependencies.
- *Service Migration:* While service placement embodies determining servers to deploy services, MEC complicates the dynamics with user mobility. Owing to mobility, static service placements may no longer yield QoS benefits as users move between locations. Service migration dynamically moves services to cater to user mobility [40]. The authors in [8] formulate service placement and migration using an MDP model. They develop heuristics for multi-user and multi-service model. However, they assume a monolithic service being used throughout the entire duration of the users' invocation which is not necessarily true for microservices architecture. In [20], a mobility-aware monolithic service migration strategy is proposed taking a direction vector approach. In [10], [41] and [42] the authors propose deep reinforcement learning methods for service migration but only consider migrations for a single user. In [43] a microservice reinforcement learning based approach is considered. However, none of these approaches consider the microservice dependency structure while migrating services, that is akin to our work.
- *Service Allocation/Routing:* While service placement concerns decisions of deploying services on servers, service allocation / routing [44], [45], [45], [46] deals with determining the assignment of service requests from users to already deployed services on MEC servers. Thus, a service allocation policy assumes an initial service placement strategy which determines the services to deploy on MEC servers upon which it proceeds to route service requests to such services. Allocation policies typically

consider metrics such as QoE / QoS, energy, etc. A number of allocation policies have been proposed in recent literature considering various metrics such as number of users allocated, QoE/QoS maximization, energy optimization, optimizing the number of re-allocations as users move about and so on. Authors in [46] propose optimal and approximate approaches for the network resource allocation problem in MEC. In [45], a game theoretic approach is formulated for the user allocation problem. In [44], the authors present an Integer Linear Programming based approach for maximizing the average number of users allocated to MEC servers while minimizing the number of MEC servers on which a service provider would have to deploy the applications. They formulate the problem as a variable bin packing problem. In [1], instead of static QoS values, the authors consider dynamic QoS parameters. Pasteris et al. [34] propose Linear Programming approaches and also provide a fast approximation for solving the service placement problem. However these approaches consider static service allocation scenarios. In [20], the authors consider dynamic scenarios where edge devices are mobile and reallocations are possible between MEC servers. In [47], the authors demonstrate the inter-relationship between service placement and service allocation and propose a joint optimization approach to tackle the same. They demonstrate the benefits of jointly optimizing service placement and allocation through designing approximation algorithms. However, all these approaches consider monolithic applications ignoring the recent adoption of the microservices architecture.

- *Microservice Placement and Migration:* While the above works consider monolithic services, with the adoption of microservice architecture, service placement and migration needs to be reconsidered through a different lens owing to the dependencies which exist between microservices [9]. Recently, microservice placement has received attention in [12] [14] [15]. While the authors in [14] and [15] consider placing such applications in an edge computing environment, we consider MEC systems where user dynamics complicate placement options [12]. Existing work neither considers proactively deploying microservices nor user mobility patterns to determine microservice placement. The authors in [12] deliberate placement and migration strategies for MEC with multi component applications but do not consider the microservice dependency graph.
- *Service Placement with Prediction:* In [48], the authors consider placement decisions while considering future predicted costs. The authors in [49] consider geo-distributed application deployment in a Fog Computing setup by incrementally deploying services. They consider proactively migrating services. The authors in [50] propose a Deep Learning based approach to predict resource utilization of Virtual Network Functions and propose approaches to incorporate Service Function Chains. They deliberate how service chains can effectively represent an indicator for resource utilization. Taking the machine learning route, in [51], a reinforcement learning based migration approach is proposed. However, none of these proactively prefetch and migrate services together.

Most of the work related to microservice-server binding is centered around determination of which microservices to allocate to available MEC servers. Our work on the other hand, deals with determining for a given microservice, how many successor microservices to deploy proactively, by learning the users' service invocation patterns in the presence of variable traffic workloads and more importantly, the target edge servers to deploy them. We thus aim to effectively learn user microservice invocation patterns as an enabler to determine which microservices to prefetch as opposed to prior work which delve into on-demand provisioning of service requests.

VII. Conclusion and Future Directions

In this paper, we propose a learning based mechanism to proactively deploy microservices on edge servers considering microservice graph application structures. For the sake of simplicity, we consider a linear workflow microservice, examples of which are abundant in practice. Even for such simple workflow structures, the proactive placement strategy and its benefits have not been addressed in literature, to the best of our knowledge. The linear structure helps us contain the possibilities we need to examine in the solution space, and helps us build the foundation of our learning based solution framework. Experimental results on real datasets are encouraging, and demonstrate the amount of latency improvements that our scheme leads to. We believe that our findings will lead to more avenues of future research in the MEC context, that can be combined for further enriching experiences for the end user for more general workflows.

In particular, for future work, we will be investigating encoding of capacity constraints in our MDP formulation which caters to applications whose microservice dependency structure is represented by a DAG. In addition, we will also be considering incorporating constraints such as CPU, bandwidth and I/O processing. Also, in this paper, we have modeled our cost functions as fixed mappings to resources. A possible future direction is to incorporate cost functions which consider runtime usage of resources to aid the agent make better decisions considering the heterogeneous system implications of the Edge Computing environment. Another possible direction is to consider the backpressure that could arise from situations involving aggressive prefetching of microservices. Additionally, a proactive placement strategy incorporating multi-hop service placement provides further future avenues.

References

[1] P. Lai, Q. He, G. Cui, X. Xia, M. Abdelrazek, F. Chen, J. G. Hosking, J. C. Grundy, and Y. Yang, "Edge user allocation with dynamic quality of service," in *ICSOC 2019*, pp. 86–101.

[2] Y. Cai, F. R. Yu, and S. Bu, "Cloud computing meets mobile wireless communications in next generation cellular networks," *IEEE Network*, vol. 28, no. 6, pp. 54–59, 2014.

[3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE IoT journal*, vol. 3, no. 5, pp. 637–646, 2016.

[4] M. Herbster, S. Pasteris, W. Shiqiang, and T. He, "Service placement with provable guarantees in heterogeneous edge computing systems," in *IEEE INFOCOM*, vol. 2019, IEEE, 2019.

[5] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas, "Joint service placement and request routing in multi-cell mobile edge computing networks," in *IEEE INFOCOM*, pp. 10–18, IEEE, 2019.

[6] T. Ouyang, R. Li, X. Chen, Z. Zhou, and X. Tang, "Adaptive user-managed service placement for mobile edge computing: An online learning approach," in *IEEE INFOCOM*, pp. 1468–1476, IEEE, 2019.

[7] V. Farhadi, F. Mehmeti, T. He, T. La Porta, H. Khamfroush, S. Wang, and K. S. Chan, "Service placement and request scheduling for data-intensive applications in edge clouds," in *IEEE INFOCOM*, pp. 1279–1287, IEEE, 2019.

[8] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge computing based on markov decision process," *IEEE/ACM ToN*, vol. 27, pp. 1272–1288, June 2019.

[9] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ASPLOS*, pp. 3–18, 2019.

[10] C. Zhang and Z. Zheng, "Task migration for mobile edge computing using deep reinforcement learning," *FGCS*, vol. 96, pp. 111–118, 2019.

[11] H. Badri, T. Bahreini, D. Grosu, and K. Yang, "Energy-aware application placement in mobile edge computing: A stochastic optimization approach," *IEEE TPDS*, vol. 31, no. 4, pp. 909–922, 2020.

[12] T. Bahreini and D. Grosu, "Efficient placement of multi-component applications in edge computing systems," in *Proceedings of the Second ACM/IEEE SEC*, pp. 1–11, 2017.

[13] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[14] S. Wang, M. Zafer, and K. K. Leung, "Online placement of multi-component applications in edge computing environments," *IEEE Access*, vol. 5, pp. 2514–2533, 2017.

[15] G. Tato, M. Bertier, E. Rivière, and C. Tedeschi, "Split and migrate: Resource-driven placement and discovery of microservices at the edge," in *OPODIS*, vol. 153, pp. 9:1–9:16, 2019.

[16] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. The MIT Press, 2018.

[17] H. Mao, S. B. Venkatakrishnan, M. Schwarzkopf, and M. Alizadeh, "Variance reduction for reinforcement learning in input-driven environments," in *ICLR*, 2019.

[18] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf, "Computational geometry," in *Computational geometry*, pp. 1–17, Springer, 1997.

[19] "Existing commercial wireless telecommunication services facilities in san francisco: Datasf: City and county of san francisco," May 2020.

[20] Q. Peng, Y. Xia, Z. Feng, J. Lee, C. Wu, X. Luo, W. Zheng, H. Liu, Y. Qin, and P. Chen, "Mobility-aware and migration-enabled online edge user allocation in mobile edge computing," in *IEEE ICWS*, pp. 91–98, IEEE, 2019.

[21] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 19–33, 2019.

[22] "Docker hub : https://hub.docker.com/."

[23] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *FGCS*, vol. 25, no. 6, pp. 599–616, 2009.

[24] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *FGCS*, vol. 29, no. 1, pp. 84–106, 2013.

[25] H. Yao, C. Bai, M. Xiong, D. Zeng, and Z. Fu, "Heterogeneous cloudlet deployment and user-cloudlet association toward cost effective fog computing," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 16, p. e3975, 2017.

[26] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," 2017.

[27] W. Chen, D. Wang, and K. Li, "Multi-user multi-task computation offloading in green mobile edge cloud computing," *IEEE TSC*, vol. 12, no. 5, pp. 726–738, 2019.

[28] Z. Chen, Z. Chen, and Y. Jia, "Integrated task caching, computation offloading and resource allocation for mobile edge computing," in *GLOBECOM*, pp. 1–6, 2019.

[29] C. Ding, J. Wang, M. Cheng, C. Chang, J. Wang, and M. Lin, "Joint beamforming and computation offloading for multi-user mobile-edge computing," in *GLOBECOM*, pp. 1–6, 2019.

[30] Z. Hao, Y. Sun, Q. Li, and Y. Zhang, "Delay - energy efficient computation offloading and resources allocation in heterogeneous network," in *GLOBECOM*, pp. 1–6, 2019.

[31] J. Xu, L. Chen, and P. Zhou, "Joint service caching and task offloading for mobile edge computing in dense networks," in *IEEE INFOCOM*, pp. 207–215, IEEE, 2018.

[32] F. Wang, J. Xu, X. Wang, and S. Cui, "Joint offloading and computing optimization in wireless powered mobile-edge computing systems," *IEEE TWC*, vol. 17, no. 3, pp. 1784–1797, 2017.

[33] L. Chen, S. Zhou, and J. Xu, "Computation peer offloading for energy-constrained mobile edge computing in small-cell networks," *IEEE/ACM ToN*, vol. 26, no. 4, pp. 1619–1632, 2018.

[34] S. Pasteris, S. Wang, M. Herbster, and T. He, "Service placement with provable guarantees in heterogeneous edge computing systems," in *IEEE INFOCOM*, pp. 514–522, IEEE, 2019.

[35] A. Tomassilli, F. Giroire, N. Huin, and S. Pérennes, "Provably efficient algorithms for placement of service function chains with ordering constraints," in *INFOCOM 2018*, pp. 774–782, 2018.

[36] B. Gao, Z. Zhou, F. Liu, and F. Xu, "Winning at the starting line: Joint network selection and service placement for mobile edge computing," in *INFOCOM 2019,*, pp. 1459–1467, 2019.

[37] L. Wang, L. Jiao, T. He, J. Li, and M. Mühlhäuser, "Service entity placement for social virtual reality applications in edge computing," in *INFOCOM 2018,*, pp. 468–476, 2018.

[38] N. Yu, Q. Xie, Q. Wang, H. Du, H. Huang, and X. Jia, "Collaborative service placement for mobile edge computing applications," in *IEEE GLOBECOM*, pp. 1–6, IEEE, 2018.

[39] L. Chen, C. Shen, P. Zhou, and J. Xu, "Collaborative service placement for edge computing in dense small cell networks," *IEEE TMC*, pp. 1–1, 2019.

[40] T. Taleb, A. Ksentini, and P. A. Frangoudis, "Follow-me cloud: When cloud services follow mobile users," *IEEE TCC*, vol. 7, pp. 369–382, April 2019.

[41] S. Cao, Y. Wang, and C. Xu, "Service migrations in the cloud for mobile accesses: A reinforcement learning approach," in *IEEE NAS*, pp. 1–10, IEEE, 2017.

[42] Z. Gao, Q. Jiao, K. Xiao, Q. Wang, Z. Mo, and Y. Yang, "Deep reinforcement learning based service migration strategy for edge computing," in *IEEE SOSE*, pp. 116–1165, 2019.

[43] S. Wang, Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. S. Shen, "Delay-aware microservice coordination in mobile edge computing: A reinforcement learning approach," *IEEE TMC*, pp. 1–1, 2019.

[44] P. Lai, Q. He, M. Abdelrazek, F. Chen, J. Hosking, J. Grundy, and Y. Yang, "Optimal edge user allocation in edge computing with variable sized vector bin packing," in *ICSOC 2018*, pp. 230–245.

[45] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM ToN*, vol. 24, no. 5, pp. 2795–2808, 2015.

[46] C. You, K. Huang, H. Chae, and B.-H. Kim, "Energy-efficient resource allocation for mobile-edge computation offloading," *IEEE TWC*, vol. 16, no. 3, pp. 1397–1411, 2016.

[47] K. Poularakis, J. Llorca, A. M. Tulino, I. Taylor, and L. Tassiulas, "Service placement and request routing in mec networks with storage, computation, and communication constraints," *IEEE/ACM Transactions on Networking*, vol. 28, no. 3, pp. 1047–1060, 2020.

[48] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. K. Leung, "Dynamic service placement for mobile micro-clouds with predicted future costs," *IEEE TPDS*, vol. 28, no. 4, pp. 1002–1016, 2016.

[49] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder, "Incremental deployment and migration of geo-distributed situation awareness applications in the fog," in *ACM DEBS*, pp. 258–269, 2016.

[50] H. Kim, S. Jeong, D. Lee, H. Choi, J. Yoo, and J. W. Hong, "A deep learning approach to vnf resource prediction using correlation between vnfs," in *IEEE Conference on Network Softwarization (NetSoft)*, pp. 444–449, 2019.

[51] F. e. a. Brandherm, "A learning-based framework for optimizing service migration in mobile edge clouds," in *EdgeSys*, pp. 12–17, 2019.