DRL-based Service Migration for MEC Cloud-Native 5G and beyond Networks

Theodoros Tsourdinis*‡, Nikos Makris*†, Serge Fdida‡ and Thanasis Korakis*†
*Dept. of Electrical and Computer Engineering, University of Thessaly, Greece
‡Sorbonne Université, CNRS, LIP6, Paris, France
†Centre for Research and Technology Hellas, CERTH, Greece
Email: ttsourdinis@uth.gr, nimakris@uth.gr, serge.fdida@sorbonne-universite.fr, korakis@uth.gr

2023 IEEE 9th International Conference on Network Softwarization (NetSoft) | 979-8-3503-9980-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/NetSoft57336.2023.10175417

***Abstract*—Multi-access Edge Computing (MEC) has been considered one of the most prominent enablers for low-latency access to services provided over the telecommunications network. Nevertheless, client mobility, as well as external factors which impact the communication channel can severely deteriorate the eventual user-perceived latency times. Such processes can be averted by migrating the provided services to other edges, while the end-user changes their base station association as they move within the serviced region. In this work, we start from an entirely virtualized cloud-native 5G network based on the OpenAirInterface platform and develop our architecture for providing seamless live migration of edge services. On top of this infrastructure, we employ a Deep Reinforcement Learning (DRL) approach that is able to proactively relocate services to new edges, subject to the user's multi-cell latency measurements and the workload status of the servers. We evaluate our scheme in a testbed setup by emulating mobility using realistic mobility patterns and workloads from real-world clusters. Our results denote that our scheme is capable sustain low-latency values for the end users, based on their mobility within the serviced region.**

***Index Terms*—Multi-access Edge Computing, Beyond 5G, Cloud-Native network, AI/ML, OpenAirInterface, Kubernetes**

I. Introduction

Multi-access Edge Computing (MEC) has been considered as a key technology for minimizing service access delay, by appropriately moving the services closer to the end-users, aka the network edge. The deployment of such edge functionality has been boosted by the definition of the 5G Service Based Architecture (SBA) for the 5G Core Network (5GCN) [1] that allows the breakout of user traffic generated and exchanged within the telecommunications network through the User Plane Function (UPF). Within the 5G architecture, UPF can be placed at the network edge, and when combined with services being deployed close to it, can enable low latency time for reaching these services, avoiding transmissions of traffic over long distances to the operator datacenters where usually the 5GCN resides.

Given that such MEC functionality is inherently supported within the existing and upcoming next-generation telecommunication networks, it is crucial that their service model complies with client mobility. As such networks provide ubiquitous access, clients can freely move around and get service through different cells. For such cases, MEC and associated services need to inherently adapt to the new locations of the users, towards maintaining low service access latency for their users. This, in turn, means that the provided services need to be migrated to new locations throughout their operation, towards ensuring seamless low-latency access.

Network Functions Virtualization (NFV) is a key technology for supporting the aforementioned functionality. By decoupling the provided services from the underlying hardware, both functions of the operator (e.g. 5GCN UPF) as well as offered services (e.g. Voice over IP, Video on Demand, etc.) can be executed as microservices, even on resource-constrained equipment on edge. This further allows their seamless (in several cases) migration to new hosts, in order to support client mobility.

Furthermore, localization in 5G networks is becoming an increasingly important aspect to consider [2]. The Location Management Function (LMF) allows for the efficient tracking and management of user locations within the network, providing real-time information about the coverage needs. This information can be used to optimize handover and service migration decisions, ensuring that services are always placed in the most appropriate location to support client mobility. Similar to this, the ETSI MEC working group has specified interfaces and functions for retrieving statistics from the radio network, with the aim to optimize the edge service deployments [3].

Additionally, incorporating Machine Learning (ML) and Artificial Intelligence (AI) approaches can significantly improve the service migration process. AI-based algorithms can be used to predict user mobility patterns and the load of edge servers, allowing proactive service migration decisions. This can help to ensure a seamless user experience, providing low-latency, high-quality services to users, even as services are migrated to new locations.

In this work, we use the state-of-the-art service orchestrator Kubernetes (k8s) in order to deploy the telecommunications network (5G RAN and CN) and accompanying services. On this setup, we develop the needed functionality towards enabling live migration capabilities for the edge services. On top, towards enabling accurate migration decisions, we employ DRL architectures that are able to learn the optimal migration decision policy based on the user's localization information and the edge workload observations. Our contributions are summarized as follows:

- To provide a seamless MEC experience to moving users of the network by exploiting our developed edge infrastructure.
- To enable continuous and uninterrupted low-latency access to services deployed on the network edge.

- To model the service migration environment as a Markov Decision Process (MDP) problem, and to design a reward function that incorporates migration cost penalties to guide the decision-making process.
- To implement DRL algorithms on top of our environments and to compare their performance.
- To select the optimal target location for the edge services by taking advantage of the user's localization, utilizing a DRL agent.
- To evaluate and integrate the developed approach in a real 5G edge setup, using realistic mobility patterns and real-world edge workload dataset.

The rest of the paper is organized as follows. Section II shows related works found in recent literature, and how our work differentiates from them. Section III shows our architecture and our contributions towards supporting such functionality. In Section IV we present our experimental findings, and in Section V we conclude this work.

II. Related Work

Multi-access Edge Computing (MEC) is considered to be the main enabler for low-latency service access in 5G communications. Through the definition of the 5GCN in a disaggregated manner and executing it using the Service Based Architecture [1], MEC can be truly realized in a low-cost manner, allowing service providers to take advantage of the network edges for providing selected services with low latency. Such applications are of particular interest to the IoT community, as for certain use cases low latency access and edge selection can be beneficial for the services offered over the top. In [4], authors discuss the role of MEC in 5G and IoT, and demonstrate how IoT applications can benefit from a MEC-enabled 5G network with a use case that utilizes MEC to achieve edge intelligence in IoT scenarios. Authors in [5] exploit the Virtual Machine (VM) technology in order to provide migration capabilities in such IoT edge scenarios, while at the same time reducing the loading time of the VM-based application by mangling the transferred files from each edge host. In [6], the authors model the problem of MEC location selection in an IoT environment as a multiattribute decision-making problem, based on SDN and NFV. In this work, the authors are able to reduce the server response time and improve the quality of the user service experience. Specifically to the 5G network model, authors in [7] present a 5G network architecture together with its network management capabilities, complementing MEC with the connectivity service. The authors address different classes of use cases and applications and evaluate their approach in a testbed setup. Subject to client mobility, modeling the best wireless channel association and service placement within the network is not a trivial task [8], especially when trying to meet a minimum Service Level Agreement (SLA) on latency with the end-user. In [9], authors argue on the applicability of MEC to a vehicular environment where services are replicated across different hosts and prove that their approach can prune the end-to-end communication latency. In [10], authors try to develop MEC solutions coupled with user mobility, for the fast relocation of service instances to guarantee the desired Quality of Experience. The authors use containers for hosting the services and develop a framework where proactive service replication for stateless applications is exploited to drastically reduce the time of service migration. In [11] and [12], authors explore the Checkpoint/Restore In Userspace (CRIU) technology to migrate containerized services to different hosts subject to client mobility. Although CRIU provides the ability to migrate stateful applications as well, it fails to address different types of protocols supported in the telecommunications network environment, such as the SCTP protocol for the N1/N2 interface between the Access and Mobility Management Function (AMF) and the gNB. In [13], authors explore the methodologies for handovers and service migrations employing probabilistic and prediction algorithms, using real-world datasets, and evaluating the implemented models. Similarly, in [14] the authors employ statistical and machine learning models to forecast the edge evolution, in order to get the migration decisions. Although these approaches are valid, classical machine learning and deep learning algorithms don't cope with the dynamic nature of edge environments. Moreover, in order for these models to be effective huge datasets are needed. In such dynamic environments, the use of reinforcement learning (RL) may be necessary in order to effectively adapt to changing conditions and make real-time migration decisions. Additionally, RL-based approaches have the added benefit of being able to consider the long-term consequences of migration decisions, rather than simply predicting the next best action. In [15] the authors propose a DRL approach for service migration in (MEC)-enabled vehicular networks in a simulation environment, observing communication delay and migration costs and evaluating the learning ability of the agent. The work reduces the end-to-end latency and migration costs. However, the solution is tested only in a simulation and there is no system architecture or an explanation of the integration of their approach in real-world infrastructures. On the contrary, in work [16], authors employ DRL for determining the bandwidth for service migrations in 5G Networks. They employ the DDPG algorithm in a continuous action space defined as the bandwidth for the corresponding migrations. They evaluate their algorithm in real-edge infrastructure utilizing CRIU technology to migrate the services. Although their solution targets 5G Networks, there is no integration of their approach into a 5G network with the respective interfaces.

In this work, we progress beyond existing literature by using a cloud-native RAN and Core Network, deployed by using a blend of micro-services and VMs, based on the OpenAir-Interface (OAI) [17] platform. The selection of the different types of virtualization depends on the services (network/edge services) as detailed further below, consisting of either Virtual Network Functions (VNFs) or Containerized Network Functions (CNFs). We blend the approaches of the CRIU-based microservices and VM-based service provisioning, towards

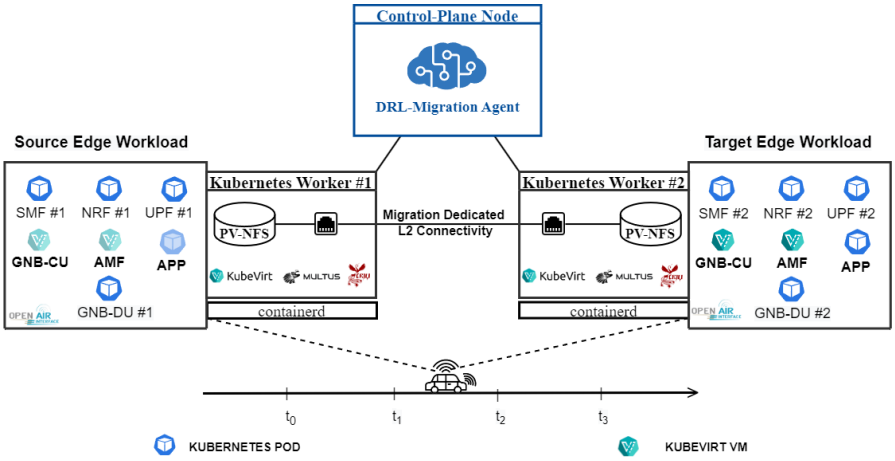


Fig. 1: The deployment of the live-migration capable 5G Edge Infrastructure on Kubernetes

reaping the benefits of both worlds in the k8s environment. By taking advantage of the multi-cell RTT feature standardized by the 3rd Generation Partnership Project (3GPP) and the workload cluster measurements, we model our infrastructure as an MDP problem. We define the states and the actions and we design a reward function that targets optimal decisions and incorporates migration cost-aware penalties. On top, we implemented a service migration Deep Q-Network (DQN) and Deep State-Action-Reward-State-Action (Deep SARSA) agents. To train the agents, we developed a digital-twin simulation environment identical to our real-world setup. Finally, we evaluate the agent's performance in the real edge infrastructure. In the next section, we detail our system architecture and key building blocks.

III. System Architecture

Our overall setup consists of a 5G Edge architecture, that is entirely based on the Kubernetes (k8s) framework, enhanced with novel capabilities for service continuity of MEC applications and maintenance of 5G Virtual Network Functions (VNFs). Fig. 1 summarizes the end-to-end service-based 5G network that uses hybrid solutions offered by the coexistence of VMs and containers. By default, there is no built-in mechanism in k8s to support the migration of stateful pods between its cluster nodes. Our architecture covers this gap using various technologies that benefit beyond 5G networks as they can contribute to the seamless experience of users regardless of their mobility. Furthermore, we enhance our architecture with a digital twin-driven DRL framework that predicts the edge conditions and takes optimal migration decisions. Below, we analyze the components of our architecture and the diverse technologies, that make up our setup. To evaluate our implementation, we utilized the NITOS testbed [18], a remotely accessible facility located at the University of Thessaly, Greece.

A. Architecture of the Edge Infrastructure

Our cluster consists of three k8s workers and one control-plane node. The control-plane node is responsible for monitoring the health of the other nodes as well as the proper operation of the VNFs and services. The remaining nodes host – by default – pods, but also VMs due to the KubeVirt framework that we deployed in our cluster. A pod is the minimal object deployment for a microservice within the k8s environment; it consists of at least one/more containerized services, that are intercommunicating with each other. KubeVirt [19] is an add-on that extends k8s capabilities by delivering VMs as container workloads. The significant addition of KubeVirt to the k8s ecosystem brings an ideal environment for edge solutions as it covers the gap of live migration of services in the k8s by taking advantage of VM live migration. Moreover, it enables us to manage the lifecycle of VMs in the same manner as for the pods via control plane commands.

However, containers can be live migrated too, mainly through the CRIU tool [20] which can restore the checkpointed states of the container to the destination node with the help of the runC container runtime. An important effort to integrate CRIU into k8s was accomplished through the PodMigration-Operator [21], [22], which can migrate a stateful pod across the k8s nodes. Nevertheless, it fails to seamlessly maintain IP/TCP connections since the pod's IP changes on the target host, even if a k8s service assigned with a static IP, routes the traffic to the pods.

We managed to maintain IP/TCP connections without interruptions, by attaching to the pods a secondary interface with the help of the Multus Container Network Interface (CNI). We created static, migration-dedicated MacVLAN interfaces that bind to a host-bridged interface consisting of physical and VLAN interfaces. The pods attach this MacVLAN interface through the ContainerNetworkDefinition with static IPs which are persistent during the live migration. The VMs attach their own static IPs on the same bridged interfaces, providing Layer 2 connectivity between them. With this approach, we were able to incorporate the altered PodMigration-Operator into our architecture.

We apply diskless live migration to both of our virtualized technologies. This allows us to transfer only the memory state of the containers/VMs, which results in avoiding disk being copied, thus lower migration times. This is achievable as all

nodes share a Network File System (NFS), where the NFS server is the control-plane node and the clients are the worker nodes. This NFS system includes the dump files containing the state pods and the images of the VMs respectively. To this end, the process of live migration in pods can be achieved with the following steps:

1) CRIU snapshots the state of the container on the target pod.
2) The snapshot dump file is exported to the NFS server.
3) A new-cloned pod is created on the target node that restores the source's pod state via the dump file.
4) The target pod is running and the source pod can be removed.

On the other hand, the KubeVirt VMs are importing their disk images through PersistentVolumeClaim (PVC) which is managed by Data Volumes from the Containerized-Data-Importer (CDI) which is a persistent storage management add-on for k8s. In order to perform diskless migrations, these PVCs are distributed to the NFS via the NFS subdir-external provisioner, i.e., an automatic provisioner that supports dynamic provisioning to pods/VMs using the already-existing NFS server. Subsequently, the disk image is always available to the target nodes and only the memory is copied from source to destination.

It is worth mentioning that in both types of migration, the Pre-Copy technique is employed since it has less downtime [23]. By comparing VMs and pods during the live migration, we conclude that pod seems like an ideal solution to deploy the edge services, as it has the least migration time. However, we choose to keep both technologies in our architecture because:

- VM live migrations are more stable and smoother (lower latency spikes) than the pod ones.
- CRIU doesn't support SCTP socket maintenance during live migrations, unlike KubeVirt VMs. This leads to the failure of live migration of 5G CNFs, as almost all of CNFs communicate over the SCTP protocol.

Our final architecture tools and technologies are gathered in table I.

TABLE I: Experimental Setup of the Edge Infrastructure

System	Description
Nodes	1- Control Plane Node & 3-Worker Nodes
CPU	Intel-Core i7-3770 @ 3.40 GHz
RAM	32GB
K8s Version	1.19.0
Container Runtime	Containerd
KubeVirt Version	0.45.0
CDI Version	1.43.0
CRIU Version	3.14.0
RUNC Version	1.0.2-dev
K8s NFS Provisioner	NFS Subdir External Provisioner
5G-Core NFs	OAI Multi-Slice Core Network
5G-RAN	OAI RF-Simulator & UERANSIM
5G-UE	OAI NR-UE
5G-SLICE	URLLC

B. Management & Deployment of Network Functions

For the telecom network, we rely on a multi-slice 5G Core Network provided by the OpenAirInterface (OAI) platform [17], which consists of the following containerized CNFs: 1) Network Repository Function (NRF), 2) Unified Data Repository (UDR), 3) Unified Data Management (UDM), 4) Authentication Server Function (AUSF) 5) Network Slice Selection Function (NSSF), 6) Access and Mobility Management Function (AMF), 7) Session Management Function (SMF), 8) User Plane Function (UPF). In this deployment, there are two Network Slice Selection Assistance Information (S-NSSAIs) configured, therefore two slices: 1) Ultra Reliable Low Latency Communications (URLLC) 2) Massive IoT (MIoT). However, we mainly focus on the URLLC slice. The NSSF, UDR, UDM, AUSF, and AMF are common to all slices, while UPF, SMF, and NRF are unique for each slice.

Likewise, for the Radio Access Network (RAN), we employ two different RAN simulators: ueransim and rfsimulator corresponding to our two different slices. Specifically, we utilized the disaggregated architecture from the rfsimulator including the CU and DU components, as the result of the gNodeB disaggregation into CU/DU. For the User Equipment (UE), we employed the OAI 5G-NR UE.

Since SCTP socket maintenance is not supported during the live migration of the pods, we decided to nest some of the containerized NFs inside the KubeVirt VMs, in order to be able to migrate them across the edge nodes. Some of them are UPF, SMF, AMF, and GNB-CU. Their selection was made because most of them are stateful functions and are of significant interest for live migration due to their importance in the control plane proper operation/maintenance and in the Quality of Service (QoS) that the UPF provides [24]. In opposition, edge services are better to run in pods, so that they can be quickly migrated, as a decrease in QoS in the user plane has a direct impact on end users, while the change in performance of the control plane doesn't directly affect the end user's experience.

C. Architecture of the DRL Migration Environment

As known, in case of a pod failure, the k8s control-plane node launches a new container in another node to replace the failed one. However, this can cause quite a few problems in an Edge 5G Network. Initially, QoS ceases to exist as there is no service availability. Even worse, crucial CNFs that are essential to the operation of a core network can stop working. In addition, when a network is characterized by its slice, as in the case of URLLC, the new pod that is deployed should be migrated not only to the healthiest node but also to the node that gives the lowest latency with the end-user. To determine the best candidate node we need to observe either the position of the UE or the latency measurements of the neighbor cells/servers along with the load of each node.

In 5G NR, LMF uses various techniques to measure the location of UE, including using Global Navigation Satellite System (GNSS) signals or using signals from the network itself. However, such techniques face accuracy errors and

require good network time synchronization. The 16th release of 3GPP includes support for multi-cell Round Trip Time (RTT) measurements as a new feature in the LMF. Specifically, the UE sends Sound Reference Signal (SRS) requests and receives Position Reference Signal (PRS) responses from multiple Base Stations. We decide to observe the multi-cell RTT measurements for our migration decisions since this method is robust against network time synchronization errors [25]. Additionally, RTT is a more suitable metric for our solution, as it indicates the responsiveness of each cell which sometimes is independent of the UE position (e.g huge cell capacity). Therefore, we end up relying on two metrics for migration decisions. The average *RTT* between the UE and the edge servers/cells, and the *load* of each edge server. We define the average RTT, R_i between the edge nodes and the UEs that the LMF monitors by Eq. 1; where x_i are the average RTT values of the last N transmissions per UE-node pair. We also define the total *load* L_i of each edge node in the cluster as the uniform degree along multiple dimensions as described by Eq. 2. The variables cpu_i and mem_i are the average utilization of *cpu* and *memory* respectively for the corresponding edge server.

$$R_i = (\frac{1}{N}) \sum_{i=i}^{N} x_i \tag{1}$$

$$L_i = \frac{1}{1 - cpu_i} \cdot \frac{1}{1 - mem_i} \tag{2}$$

Service migration is a challenging problem due to the dynamic nature of the environment and the complex interactions between the UE, the servers, and the network. Traditional approaches to service migration, such as rule-based or heuristic-based methods, may not be able to adapt to changing conditions or handle complex dynamics effectively. Model-free and policy-based Reinforcement Learning (RL) is well-suited for dynamic environments where the conditions may change over time, such as in a service migration environment where the UE is moving and the loads on the servers may vary. By using RL, the agent can learn an optimal policy for minimizing the RTT between UE and the servers, and for balancing the loads on the servers. This can help to improve the overall performance of the system and provide a better experience for the UE. Traditional RL algorithms such as Q-Learning, use a Q-table to store each state and the corresponding values of all actions (Q-value). However, this cannot scale if the state space expands, since the Q-table will also become larger, resulting in inefficient learning. Deep Reinforcement Learning is particularly effective at adapting to these changing conditions, as it can learn from a large amount of data and can generalize to unseen situations. Moreover, DRL employs a Q-function rather than a Q-table and utilizes deep neural networks (Deep Q-Networks/DQN) that estimate the Q-function, resulting in effective and scalable learning. By taking the aforementioned into account, and by exploiting the edge migration capabilities of our architecture, we designed and implemented a DRL Service Migration framework. The architecture of our solution is illustrated in Fig. 2.

We created two identical custom environments, by utilizing the OpenAI Gym platform [26]. The first one is a simulation environment for training purposes, playing the role of a digital twin in the real environment. The second one is for evaluating our solution in a real-world environment. The only difference between them is that the real-world environment employs the real cluster and leverages the migration APIs that we developed in the section III-A. This allows us to safely and efficiently explore a wide range of possible scenarios and actions without damaging real-world systems. Both environments are modeled as a Markov decision process, with the same states *s*, actions *a*, and rewards *r*. The states can be observed by the Eq. 3.

$$s = (N_i, R_1, ...R_N, L_1, ...L_N) \tag{3}$$

In this equation, the states are represented as a tuple of the variables; N_i is the Node where the set of user's edge services (pods/VMs) are running and $_N$ is the total number of nodes. In our cluster, we have three edge nodes, thus the states can be redefined as s = (N_i, R_1, R_2, R_3, L_1, L_2, L_3). The action space *a* includes the actions: *Wait* and *Migrate to* N_i *server*. The Migrate action migrates the set of edge services to a specific N_i edge server, thus including as many migration possible actions as the number of edge nodes. The *Wait* action simply means that the agent doesn't migrate the services to any of the candidate nodes at the current time-step. The rewards *r* represent the feedback the RL agent receives after taking action in a given state. We define two local rewards: r_R and r_L.

$$r_R = \frac{R_{min}}{R_i} - \frac{R_i}{R_{max}} + \frac{\min(R_{list})}{R_i} - \frac{R_i}{\max(R_{list})} \tag{4}$$

$$r_L = \frac{L_{min}}{L_i} - \frac{L_i}{L_{max}} + \frac{\min(L_{list})}{L_i} - \frac{L_i}{\max(L_{list})} \tag{5}$$

Both rewards determine the feedback for each action taken by the agent from the point of view of *RTT* and *load* of the edge servers respectively. Each reward is calculated to a similar respective mathematical formula given by Eq. 4 and 5. In these equations, the values R_i and L_i are the current *RTT* and *load* values of the server that the agent migrated or stayed to. The ranges *(*R_{min},R_{max}*)* and *(*L_{min},L_{max}*)* are *RTT* and *load* Service Level Agreement (SLA) thresholds. Furthermore, the lists R_{list} and L_{list} contain the *RTT* and *load* values for all the candidate nodes. Both rewards are designed to reward/penalize the agent when the migrated server's *RTT* or *load* values are inside/outside SLA thresholds and when the agent selects the most optimal/mediocre server between the candidate nodes. Specifically, the first subtraction of the fractions expresses the ranking of the node in the SLA thresholds, while the second one expresses the ranking of the node among the candidate nodes. In addition, both rewards are migration-cost-aware since they penalize the agent if it migrates the services to less optimal nodes. For example, in the case of *RTT* if the measurements are the following for each

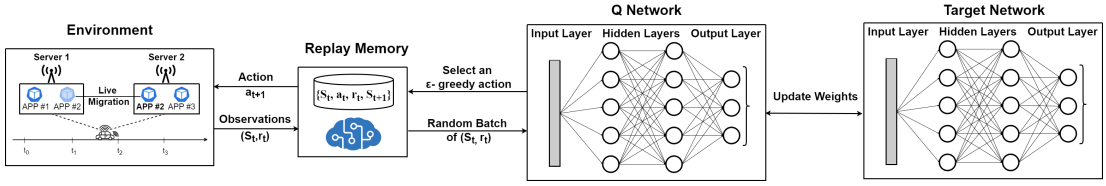


Fig. 2: Deep Reinforcement Learning Architecture for the Live Migration Environment.

node: $(3, 5, 6)_{ms}$ the corresponding reward functions will be: $(0.6, -0.2, -0.6)$ with thresholds defined as $R_{min} = 1$ and $R_{max} = 20$. Although every *RTT* value is far below the R_{max}, the reward functions are negative for the less optimal nodes, preventing the agent from migrating services to them. This results in resource and bandwidth saving, as the agent will try to migrate or stay to the most optimal node, instead to follow an always-migrate policy. However, to better assist the agent to avoid relocating services to the most unhealthy node, we introduce a migration penalty. This penalty is given by Eq. 6 and 7.

$$p_R = \begin{cases} +\frac{\text{cost}}{2}, & \text{if } R_i > R_{\max} \text{ and } R_i = \max(R_{list}) \\ 0\ , & \text{otherwise} \end{cases} \quad (6)$$

$$p_L = \begin{cases} +\frac{\text{cost}}{2}, & \text{if } L_i > L_{\max} \text{ and } L_i = \max(L_{list}) \\ 0\ , & \text{otherwise} \end{cases} \quad (7)$$

Both p_R and p_L, penalize the agent for exceeding the maximum allowed RTT/load, and for selecting the node with the worst rank among candidate nodes. Each penalty amounts to half of the migration cost. This cost is a hyperparameter and in our case, it symbolizes the maximum bandwidth that can be wasted for one service relocation and it's a constant. Finally, the global reward is the sum of the two individual rewards subtracted by the sum of the two individual penalties, as can be observed by Eq. 8. It is worth mentioning that each time the *RTT* or *load* thresholds are exceeded we terminate the episode. This approach has been taken, to guide the agent to not violate the SLA and to address the credit-assignment problem. The credit-assignment problem occurs when the agent receives the reward at the end of each episode without identifying the responsible actions.

$$r = r_R + r_L - (p_R + p_L) \quad (8)$$

To evaluate the performance of the proposed DRL solution, we employ real-world scenarios. Since multi-cell RTT measurements-datasets are not yet publicly available, we implemented a realistic mobility scenario. This scenario emulates a part of a real-world 5G commercial topology, located on State Route 111 highway, California U.S. The map topology illustrated in Fig. 3 is obtained by the Ookla 5G Map [27]. Precisely, this scenario emulates cars traveling on the given highway in both directions, with speeds varying from 80 to 104.5 km/h with the limit of the highway being 105 km/h. On this route, there are three 5G antennas, with approximately equal distance between them. We assume that the edge servers are located next to the antennas and that we monitor the Muli-Cell RTT measurements through the LMF. The RTT values are linearly proportional to the Euclidean distances between UEs and edge servers/base-stations. Also, the RTT values are affected by the radio interference as random loss, which we generate by adding Additive White Gaussian Noise (AWGN) with a fixed standard deviation per route. The rate at which the RTT changes depends mainly on car speeds and different driving profiles. To generate a large variance that could lead to efficient learning and generalization of the agent, we distribute the variety of speeds uniformly. Subsequently, the RTT values of UE/cell pair change as $\delta_{RTTi} = \frac{d_i}{v_i} + \lambda$, where the d_i is the distance between the UE and the corresponding edge server, vi is the velocity defined by $v_i = uniform(80, 104.5)$ and the random-loss $\lambda = awgn(0.5, 0, 1)$.

To emulate realistic edge workloads, we relied on Google cluster workload traces [28]. This open dataset includes resource requests and usage measurements from Google's Borg cloud clusters, for an entire month. Specifically, we utilize the average *cpu* and *memory* usage from three different machine IDs in the cluster, given by the corresponding equations: $cpu = \frac{\Sigma(U_{\text{cpu}})}{T_{\text{window}}}$ and $mem = \frac{\Sigma(U_{\text{mem}}) \cdot T_{\text{sample}}}{T_{\text{window}}}$. The variables U_{cpu} and U_{mem} are the *cpu* and *memory* usage respectively, while the T_{window} is the measurement window and T_{sample} is the length of the sample. We obtain the *cpu* and *memory* every time-step and we calculate the total *load* per edge server given by Eq. 2. In order to avoid overfitting and to have a large variance to the repetitive load scenario we apply additional Gaussian noise to memory and CPU respectively. The noise is applied each time the scenario is repeated and follows a normal distribution with a mean of 0 and a standard deviation of 1 for both *cpu* and *mem* metrics of each node.

This way we can observe all the states that we defined on Eq. 3 by emulating mobility and load scenarios that take place in a well-defined real-world topology with realistic load patterns among the edge servers.

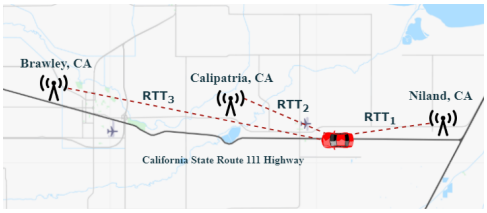


Fig. 3: Part of a real-world 5G commercial topology located near State Route 111 highway, California U.S.

To learn an optimal policy for this environment using DRL, we utilize a deep Q-network approach, where the DQN agent is trained to predict the expected reward for each action in a given state via the Q-Network. The Q-Network is a neural network and in our case, the Q-Network is a Long short-term memory (LSTM). The reason behind choosing LSTM is that it can handle variable-size inputs and outputs, thus resulting in easier retraining of the agent. It is responsible for approximating the action-value function $Q(s, a)$ and is updated at each time-step based on the current state and chosen action. In order to stabilize the learning process, we implemented also a target neural network in our system. The target network is identical to the Q-Network and is used to generate the target values for the Q-Network updates [29]. The target network is not involved in the training and it is only updated by the Q-Network periodically. This results in the reduction of the variance in the learning process and can improve the stability of the system. In addition to the Q-Network and target network, we employ a replay buffer to store past experiences and sample them during the training process. This helps to decorrelate the experiences and can also improve the stability and sample efficiency of the learning process. In more detail, the $Q(s, a)$ is updated based on experiences in the environment, which are stored in the replay buffer and sampled for learning. At each step, the Q-Network takes the current state as input and produces a vector of estimates of the action-values for each possible action. The Q-Network is then updated using gradient descent to minimize the mean squared error between the predicted and target values. The target network is periodically updated to match the weights of the Q-Network and produces the target values. Then, computes the estimated return of taking the selected action in the current state and the optimal action in the next state via y_i:

$$y_i = r + \gamma \max_{a'} Q_{target}(s', a'; \theta)$$

where r is the reward received after taking action a, s' is the next state, a' is the next action, γ is the discount factor that controls the importance of future rewards, and θ symbolizes the updated weight parameters. Finally, the Q(s,a) is updated based on the cost function $L(\theta)$ which is the squared difference between target Q and predicted Q:

$$L(\theta) = \mathbb{E}_{s,a,r,s'}\left[(y_i - Q(s, a; \theta))^2\right]$$

In addition to the DQN algorithm, we also implemented another RL algorithm called SARSA (State-Action-Reward-State-Action). In contrast with DQN, SARSA is an on-policy algorithm as the $Q(s, a)$ is updated based on the current choices of the policy. The SARSA algorithm differs from DQN in the way the target values are computed. Instead of using the maximum expected future reward, SARSA uses the reward and the expected action value of the next state to update the current action value via y_i':

$$y_i' = r + \gamma Q_{target}(s', a'; \theta)$$

In our implementation, the SARSA agent employs a similar DRL architecture as the DQN with a Q-network (MLP neural network). This kind of implementation is mentioned by the literature as Deep Sarsa [30].

To address the "exploration vs exploitation" problem, we employ the LinearAnnealedPolicy for both algorithms. In this policy, the exploration rate ϵ that controls the probability of selecting a random action is decreased linearly. This reduction rate is controlled by the exploration rate decay d which directs the rate at which ϵ decreases over time. This allows the agent to gradually shift from exploration to exploitation as it learns the optimal actions for a given state. To implement the DQN and SARSA architectures we relied on TensorFlow keras-rl2 python library.

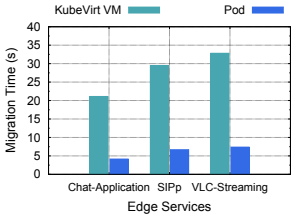
All the aforementioned hyper-parameters and the corresponding values we used for the optimal training are gathered in the table II after extensive experimentation. The common hyper-parameters of DQN and Deep Sarsa algorithms such as Q-Network, γ, α, ϵ, and d are chosen to have the same values for close comparison.

TABLE II: Deep Reinforcement Learning Parameters.

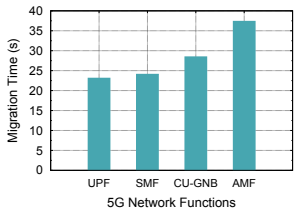
Parameter	Value
Deep Q-Network	LSTM
Deep Q-Network depth	2
Hidden layer depth	24
Optimizer	Adam
Activation	ReLU
Target Model Update	20
Replay buffer size	20000
Discount factor γ	0.99
Policy	LinearAnnealedPolicy
Learning rate α	0.001
Exploration rate ϵ	1.0
Exploration rate decay d	0.1
Number of steps	200000

IV. Evaluation

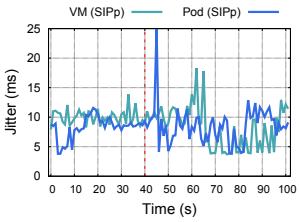
For the evaluation part of the Edge-Cloud Infrastructure, we initially compared the migration times on KubeVirt VMs and pods in various types of applications: 1) Text Application server, 2) SIPp [31] server, and 3) VLC streaming server. The SIPp application uses Session Initiation Protocol (SIP) to transfer VoIP packets. As illustrated in Fig. 4a, the migration times are considerably lower in the pods compared to the KubeVirt VMs. However, the migration times of the VMs are generally not prohibitive. Next, we focused on the migration times of VMs that are hosting various NFs including SMF, UPF, AMF, and CU. The operation of the network functions is uninterrupted and the AMF-VM has the longest migration time and this can be observed by Fig. 4b. Next, we captured the latency and the throughput that the end-user experiences during the interactions with the SIPp server while the server was migrating to other edge nodes, as a pod, and as a VM. The results are displayed in Figs. 4c and 4d, where in both plots, the vertical-dotted line denotes the time that the migration was initiated. Fig.4c shows that the VM migration had a smoother impact on the experience of UE in contrast to pod migration which completed much faster (at 47th second), but had a significantly higher spike in the observed latency.



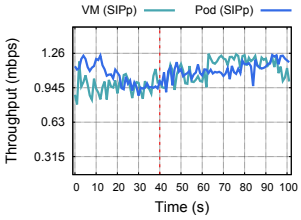
(a) Migration time on services: *VM vs Pod*



(b) Migration time on *NFs* as *VMs*



(c) End-to-End *Jitter* during migration of services: *VM vs Pod*.



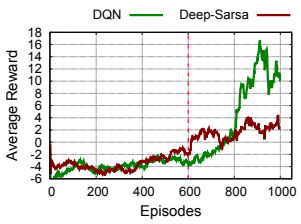
(d) End-to-End *Throughput* during migration of services: *VM vs Pod*.

Fig. 4: Live Migration measurements.

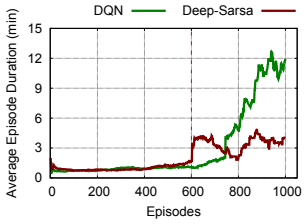
Fig.4d shows that the end-user had a seamless experience in terms of throughput in both virtualization technologies. There was only a small imperceptible drop during the migration, which was followed by a small rise. It is worth noting that the service relocated to a node that is closer to the UPF/g-NodeB. Therefore, a small drop in jitter and a small increase in throughput are subsequently observed.

Toward evaluating the DQN and Deep Sarsa agents, we trained both agents for 200.000 steps. Fig. 5a illustrates the average reward over the training episodes for the two agents. Both agents were able to learn and improve their performance over the course of the episodes. Moreover, they explored the action space effectively in the beginning and then switched to exploiting the learned policy as the episodes progressed. Specifically, before the 600th episode both agents were fully exploring the environment. After the vertical line, the agents progressively started exploiting and this is demonstrated by the increase in average reward over the episodes which by the end converged. However, the DQN agent had a better performance, as it reached higher rewards. This denotes that the DQN agent is trained efficiently, as there's a good balance between exploration and exploitation. Additionally, the DQN agent maintained the QoS at higher levels during the phase of exploiting. This is indicated by Fig. 5b, which displays the increase in the average episode duration. The QoS is increased since we terminate the episodes, each time the RTT or load thresholds are exceeded. This means that the DQN agent took better actions that met the conditions of the SLA. Although sometimes failed to not violate the SLA, due to the fact that the cluster might be overloaded. For the aforementioned reasons, the DQN agent qualifies for the taking of service migration decisions.

We evaluated the DQN agent's performance, in the real-world k8s cluster with the developed migration APIs. Specifically, by taking advantage of TensorFlow's save/load methods



(a) Average reward per episode during training.



(b) Average duration per episode during training.

Fig. 5: Agents training evaluation: DQN vs Deep Sarsa.

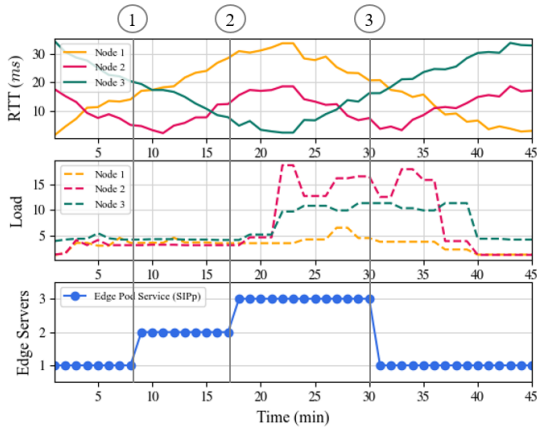


Fig. 6: DQN agent's actions during user's movement in the highway, in an overloaded edge cluster; vertical lines denote when the migrations take place on pods.

we loaded the saved agent's model weights and started testing it in our second evaluation environment. We employ the mobility scenario, by emulating a car that has the max speed on the highway for one round-trip (two-way route). We also apply an unseen heavy edge workload from one specific day of the entire measurement month by leveraging Google's Borg cloud cluster dataset. The OAI-NR-UE interacts with the SIPp server that is packaged as a pod, in order for the agent to achieve the least migration times. The footprint of this experiment is illustrated in Fig. 6. The RTT and load thresholds are $R_{max} = 20$ and $L_{max} = 12$ respectively. At the beginning of the experiment, the agent stays on the optimal Node 1. Then at the time points 1 and 2, the agent follows the UE by relocating the service near the end-user (Follow-Me approach), before the RTT threshold is exceeded, and at the same time, schedules it in healthy nodes. Before time point 3, the service is running on Node 3 but as the RTT increases the agent should relocate the service to a better candidate node. However, in that case, every node is out of SLA thresholds, so instead to migrate the service to the overloaded but with satisfactory RTT Node 2, it remains on Node 3 violating the SLA but saving resources. Finally, at time point 3, it relocates the service to Node 1, which falls within the SLA thresholds. This implies that the agent learned the policy successfully, as it proactively relocated the service to the nodes with the optimal RTT and load values and saved resources without performing unnecessary migrations.

V. Conclusion

In this work, we developed and experimentally evaluated an SLA-aware 5G edge infrastructure that offers high QoS to the end users regardless of their mobility. We developed the necessary migration capabilities in a k8s environment supporting VM and pod technologies. Our system provides continuous low-latency access to the edge services and an uninterrupted throughput experience. On top of this setup, we implemented a digital-twin environment that is identical to the real-world environment and we developed DQN and Deep Sarsa, migration agents. After the training of our agents in the simulation environment, we employed the DQN agent's model weights in our real-world infrastructure. Our results denote that the DQN agent successfully learned the policy based on multi-cell RTT measurements and the workload of edge servers. Our framework can dynamically and proactively relocate MEC apps in a k8s environment depending on the experience of the users and the condition of the edge nodes. In the future, we foresee extending our scheme to support proactive handover decisions synergistically with service migrations.

Acknowledgment

The research leading to these results has received funding from the European Horizon 2020 Programme for research, technological development and demonstration under Grant Agreement Number No 101008468 (H2020 SLICES-SC). The European Union and its agencies are not liable or otherwise responsible for the contents of this document; its content reflects the view of its authors only.

References

[1] S. Kekki *et al.*, "ETSI White Paper No. 28: MEC in 5G networks," 2018.

[2] R. Keating, D. Yoon, T. Tao, and H. Huang, "Opportunities and challenges for nr rat-dependent based positioning," in *2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall)*, 2019, pp. 1–6.

[3] F. Giust, X. Costa-Perez, and A. Reznik, "Multi-access edge computing: An overview of ETSI MEC ISG," *IEEE 5G Tech Focus*, vol. 1, no. 4, p. 4, 2017.

[4] Y. Liu, M. Peng, G. Shou, Y. Chen, and S. Chen, "Toward Edge Intelligence: Multiaccess Edge Computing for 5G and Internet of Things," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 6722–6747, 2020.

[5] W. Lu, X. Meng, and G. Guo, "Fast Service Migration Method Based on Virtual Machine Technology for MEC," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4344–4354, 2019.

[6] Z. Lv and W. Xiu, "Interaction of Edge-Cloud Computing Based on SDN and NFV for Next Generation IoT," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 5706–5712, 2020.

[7] M. Corici, P. Chakraborty, and T. Magedanz, "A Study of 5G Edge-Central Core Network Split Options," *Network*, vol. 1, no. 3, pp. 354–368, 2021. [Online]. Available: https://www.mdpi.com/2673-8732/1/3/20

[8] S. Lee, S. Lee, and M.-K. Shin, "Low Cost MEC Server Placement and Association in 5G Networks," in *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, 2019, pp. 879–882.

[9] M. Emara, M. C. Filippou, and D. Sabella, "MEC-Assisted End-to-End Latency Evaluations for C-V2X Communications," in *2018 European Conference on Networks and Communications (EuCNC)*, 2018.

[10] I. Farris, T. Taleb, H. Flinck, and A. Iera, "Providing ultra-short latency to user-centric 5G applications at the mobile network edge," *Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 4, p. e3169, 2018, e3169 ett.3169. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3169

[11] S. Ramanathan, K. Kondepu, M. Razo, M. Tacca, L. Valcarenghi, and A. Fumagalli, "Live Migration of Virtual Machine and Container Based Mobile Core Network Components: A Comprehensive Study," *IEEE Access*, vol. 9, pp. 105 082–105 100, 2021.

[12] S. Ramanathan, A. Bhattacharyya, K. Kondepu, M. Razo, M. Tacca, L. Valcarenghi, and A. Fumagalli, "Demonstration of Containerized Central Unit Live Migration in 5G Radio Access Network," in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp. 225–227.

[13] H. Abdah, J. P. Barraca, and R. L. Aguiar, "Handover prediction integrated with service migration in 5g systems," in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–7.

[14] M. Pomalo, V. T. Le, N. El Ioini, C. Pahl, and H. R. Barzegar, "Service migration in multi-domain cellular networks based on machine learning approaches," in *2020 7th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 2020, pp. 1–8.

[15] A. Abouaomar, Z. Mlika, A. Filali, S. Cherkaoui, and A. Kobbane, "A deep reinforcement learning approach for service migration in mec-enabled vehicular networks," in *2021 IEEE 46th Conference on Local Computer Networks (LCN)*, 2021, pp. 273–280.

[16] R. A. Addad, D. L. C. Dutra, T. Taleb, and H. Flinck, "AI-Based Network-Aware Service Function Chain Migration in 5G and Beyond Networks," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 472–484, 2022.

[17] N. Nikaein, M. K. Marina, S. Manickam, A. Dawson, R. Knopp, and C. Bonnet, "Openairinterface: A flexible platform for 5g research," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 5, p. 33–38, Oct. 2014. [Online]. Available: https://doi.org/10.1145/2677046.2677053

[18] N. Makris, C. Zarafetas, S. Kechagias, T. Korakis, I. Seskar, and L. Tassiulas, "Enabling open access to LTE network components; the NITOS testbed paradigm," in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2015, pp. 1–6.

[19] "KubeVirt: Virtualization extension for Kubernetes," [Online], https://github.com/kubevirt/kubevirt.

[20] "CRIU - a utility to checkpoint/restore Linux tasks," [Online], https://criu.org.

[21] J. Schrettenbrunner, "Migrating pods in kubernetes," Ph.D. dissertation, 12 2020.

[22] "Podmigration-Operator: An operator that supports Pod Migration in K8s," [Online], https://github.com/SSU-DCN/podmigration-operator.

[23] M. Hines and K. Gopalan, "Post-copy based live virtual machine migration using pre-paging and dynamic self-ballooning," 01 2009, pp. 51–60.

[24] M. Gundall, J. Stegmann, C. Huber, and H. Schotten, "Towards organic 6g networks: Virtualization and live migration of core network functions," 10 2021.

[25] S. Dwivedi, R. Shreevastav, F. Munier, J. Nygren, I. Siomina, Y. Lyazidi, D. Shrestha, G. Lindmark, P. Ernström, E. Stare, S. M. Razavi, S. Muruganathan, G. Masini, Busin, and F. Gunnarsson, "Positioning in 5g networks," 2021. [Online]. Available: https://arxiv.org/abs/2102.03361

[26] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016. [Online]. Available: https://arxiv.org/abs/1606.01540

[27] "OOKLA 5G MAP™: The interactive Ookla 5G Map tracks 5G rollouts in cities across the globe." [Online], https://www.speedtest.net/ookla-5g-map.

[28] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: the next generation," in *EuroSys'20*, Heraklion, Crete, 2020.

[29] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015. [Online]. Available: https://arxiv.org/abs/1509.06461

[30] D. Zhao, H. Wang, K. Shao, and Y. Zhu, "Deep reinforcement learning with experience replay based on sarsa," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2016, pp. 1–6.

[31] "SIPp - a SIP protocol test tool," [Online], https://github.com/SIPp/sipp.