Structure-aware reinforcement learning for node-overload protection in mobile edge computing

Anirudha Jitani, *Student Member, IEEE,* Aditya Mahajan, *Senior Member, IEEE,* Zhongwen Zhu, *Senior Member, IEEE,* Hatem Abou-zeid, *Member, IEEE,*
Emmanuel Thepie Fapi, and Hakimeh Purmehdi, *Member, IEEE*

***Abstract*—Mobile Edge Computing (MEC) involves placing computational capability and applications at the edge of the network, providing benefits such as reduced latency, reduced network congestion, and improved performance of applications. The performance and reliability of MEC degrades significantly when the edge server(s) in the cluster are overloaded. In this work, an adaptive admission control policy to prevent edge node from getting overloaded is presented. This approach is based on a recently-proposed low complexity RL (Reinforcement Learning) algorithm called SALMUT (Structure-Aware Learning for Multiple Thresholds), which exploits the structure of the optimal admission control policy in multi-class queues for an average-cost setting. We extend the framework to work for node overload-protection problem in a discounted-cost setting. The proposed solution is validated using several scenarios mimicking real-world deployments in two different settings — computer simulations and a docker testbed. Our empirical evaluations show that the total discounted cost incurred by SALMUT is similar to state-of-the-art deep RL algorithms such as PPO (Proximal Policy Optimization) and A2C (Advantage Actor Critic) but requires an order of magnitude less time to train, outputs easily interpretable policy, and can be deployed in an online manner.**

***Index Terms*—Communication systems, Mobile communications, Markov process, Adaptive control, Traffic control (communications)**

I. Introduction

In the last decade, we have seen a shift in the computing paradigm from co-located datacenters and compute servers to cloud computing. Due to the aggregation of resources, cloud computing can deliver elastic computing power and storage to customers without the overhead of setting up expensive datacenters and networking infrastructures. It has specially attracted small and medium-sized businesses who can leverage the cloud infrastructure with minimal setup costs. In recent years, the proliferation of Video-on-Demand (VoD) services, Internet-of-Things (IoT), real-time online gaming platforms, and Virtual Reality (VR) applications has lead to a strong focus on the quality of experience of the end users. The cloud paradigm is not the ideal candidate for such latency-sensitive applications owing to the delay between the end user and cloud server.

This has led to a new trend in computing called Mobile Edge Computing (MEC) [1], [2], where the compute capabilities are moved closer to the network edges. It represents an essential building block in the 5G vision of creating large distributed, pervasive, heterogeneous, and multi-domain environments. Harvesting the vast amount of the idle computation power and storage space distributed at the network edges can yield sufficient capacities for performing computation-intensive and latency-critical tasks requested by the end users. However, it is not feasible to set-up huge resourceful edge clusters along all network edges that mimic the capabilities of the cloud due to the sheer volume of resources that would be required, and which would remain underutilized most of the times. Due to the limited resources at the edge nodes and fluctuations in the user requests, an edge cluster may not be capable of meeting the resource and service requirements of all the users it is serving.

Computation offloading methods have gained popularity as they provide a simple solution to overcome the problems of edge and mobile computing. Data and computation offloading can potentially reduce the processing delay, improve energy efficiency, and even enhance security for computation-intensive applications. The critical problem in the computation offloading is to determine the amount of computational workload, and choose the MEC server from all available servers. Various aspects of MEC from the point of view of the mobile user have been investigated in the literature. For example, the questions of when to offload to a mobile server, to which mobile server to offload, and how to offload have been studied extensively. See, [3]–[7] and references therein.

However, the design questions at the server side have not been investigated as extensively.

When an edge server receives a large number of requests in a short period of time (for example due to a sporting event), the edge server can get overloaded, which can lead to service degradation or even node failure. When such service degradation occurs, edge servers are configured to offload requests to other nodes in the cluster in order to avoid the node crash. The crash of an edge node leads to the reduction of the cluster capacity, which is a disaster for the platform operator as well as the end users, who are using the services or the applications. However, performing this migration takes extra

Anirudha Jitani is with the School of Computer Science, McGill University and Montreal Institute of Learning Algorithms. Aditya Mahajan is with the Department of Electrical and Computer Engineering, McGill University, Canada.
Emails: anirudha.jitani@mail.mcgill.ca, aditya.mahajan@mcgill.ca
Zhongwen Zhu, Emmanuel Thepie Fapi, and Hakimeh Purmehdi are with Global AI Accelerator, Ericsson, Montreal, Canada. Hatem Abou-zeid is with Electrical and Software Engineering Department, University of Calgary.
Emails: zhongwen.zhu@ericsson.com, emmanuel.thepie.fapi@ericsson.com, hakimeh.purmehdi@ericsson.com, hatem.abouzeid@ucalgary.ca.
A preliminary version of this paper appeared in the proceedings of the 2021 IEEE International Conference on Communications (ICC).
This work was supported in part by MITACS Accelerate, Grant IT16364.

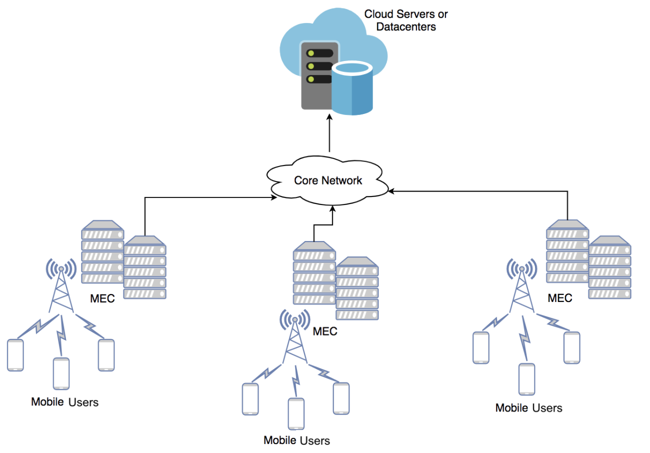


Fig. 1. A Mobile Edge Computing (MEC) system. User may be mobile and will connect to the closest edge server. The MEC servers are connected to the backend cloud server or datacenters through the core network.

time and reduces the resources availability for other services deployed in the cluster. Therefore, it is paramount to design *proactive* mechanisms that prevent a node from getting overloaded using dynamic offloading policies that can adapt to service request dynamics.

The design of an offloading policy has to take into account the time-varying channel conditions, user mobility, energy supply, computation workload and the computational capabilities of different MEC servers. The problem can be modeled as a Markov Decision Process (MDP) and solved using dynamic programming. However, solving a dynamic program requires the knowledge of the system parameters, which are not typically known and may also vary with time. In such time-varying environments, the offloading policy must adapt to the environment. Reinforcement Learning (RL) [8] is a natural choice to design such adaptive policies as they do not need a model of the environment and can learn the optimal policy based on the observed per-step cost. RL has been successfully applied for designing adaptive offloading policies in edge and fog computing in [9]–[14] to realize one or more objectives such as minimizing latency, minimizing power consumption, association of users and base stations. Although RL has achieved considerable success in the previous work, this success is generally achieved by using deep neural networks to model the policy and the value function. Such deep RL algorithms require considerable computational power and time to train, and are notoriously brittle to the choice of hyper-parameters. They may also not transfer well from simulation to the real-world, and output policies which are difficult to interpret. These features make them impractical to be deployed on the edge nodes to continuously adapt to the changing network conditions.

In this work, we study the problem of node-overload protection for a single edge node. Our main contributions are as follows:

- We present a mathematical model for designing an offloading policy for node-overload protection. The model incorporates practical considerations of server holding, processing and offloading costs. In the simplest case, when the request arrival process is time-homogeneous, we model the system as a continuous-time MDP and use the *uniformization technique* [15]–[17] to convert the continuous-time MDP to a discrete-time MDP, which can then be solved using standard dynamic programming algorithms [18].
- We show that for time-homogeneous arrival process, the value function and the optimal policy are weakly increasing in the CPU utilization.
- We design a node-overload protection scheme that uses a recently proposed low-complexity RL algorithm called Structure-Aware Learning for Multiple Thresholds (SALMUT) [19]. The original SALMUT algorithm was designed for the average cost models. We extend the algorithm to the discounted cost setup and prove that SALMUT converges almost surely to a locally optimal policy.
- We compare the performance of Deep RL algorithms with SALMUT in a variety of scenarios in a simulated testbed which are motivated by real world deployments. Our simulation experiments show that SALMUT performs close to the state-of-the-art Deep RL algorithms such as PPO [20] and A2C [21], but requires an order of magnitude less time to train and provides optimal policies which are easy to interpret. Our experiments also show that although our algorithm is designed under an idealized assumption on the arrival process, it performs well even when the ideal assumption is not satisfied.
- We developed a docker testbed where we run actual workloads and compare the performance of SALMUT with the baseline policy. Our results show that SALMUT algorithm outperforms the baseline algorithm.

A preliminary version of this paper appeared in [22], where the monotonicity results of the optimal policy (Proposition 1 and 2) were stated without proof and the modified SALMUT algorithm was presented with a slightly different derivation. However, the convergence behavior of the algorithm (Theorem 2) was not analyzed. A preliminary version of the comparison on SALMUT with state of the art RL algorithms on a computer simulation was included in [22]. However, the detailed behavioral analysis (Sec. V) and the results for the docker testbed (Sec. VI) are new.

The rest of the paper is organized as follows. We present the system model and problem formulation in Sec. II. In Sec. III, we present a dynamic programming decomposition for the case of time-homogeneous statistics of the arrival process. In Sec. IV, we present the structure aware RL algorithm (SALMUT) proposed in [19] for our model. In Sec. V, we conduct a detailed experimental study to compare the performance of SALMUT with other state-of-the-art RL algorithms using computer simulations. In Sec. VI, we compare the performance of SALMUT with baseline algorithm on the real-testbed. Finally we conclude in Sec. VII.

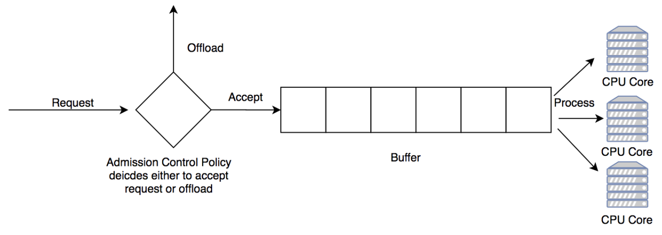

Offload
Request
Accept
Admission Control Policy deicdes either to accept request or offload
Buffer
Process
CPU Core
CPU Core
CPU Core

Fig. 2. System model of admission control in a single edge server.

TABLE I
LIST OF SYMBOLS USED

Symbol	Description
X_t	Queue length at time t
L_t	CPU load of the system at time t
A_t	Offloading action taken by agent at time t
k	Number of cores in the edge node
R	CPU resources required by a request
$P(r)$	PMF of the CPU resources required
μ	Processing time of a single core in the edge node
λ	Request arrival rate of user
h	Holding cost per unit time
$c(\ell)$	Running cost per unit time
$p(\ell)$	Penalty for offloading the packet
$\rho(x, \ell, a)$	Cost function in the continuous MDP
π	Policy of the RL agent
α	Discount rate in continuous MDP
$V^{\pi}(x, \ell)$	Performance of the policy π
$p(x', \ell' \mid x, \ell, a)$	Transition probability function
β	Discount rate in discrete MDP
$\bar{\rho}(x, \ell, a)$	Cost function in the discrete MDP
$Q(x, \ell, a)$	Q-value for state (x, ℓ) and action a
τ	Threshold vector
π_τ	Optimal Threshold Policy for SALMUT
$J(\tau)$	Performance of the SALMUT policy π_τ
$f(\tau(x), \ell)$	Probability of accepting new request
θ	Temperature of the sigmoid function
$\mu(x, \ell)$	Occupancy measure on the states starting from (x_0, ℓ_0)
b_n^1	Fast timescale learning rate
b_n^2	Slow timescale learning rate
C_{off}	Number of requests offloaded by edge node
C_{ov}	Number of times edge node enters an overloaded state

II. MODEL AND PROBLEM FORMULATION

A. System model

A simplified MEC system consists of multiple edge servers and several mobile users accessing the servers (see Fig. 1). We consider one edge server in such a system. Mobile users (connected to that server) independently generate service requests according to a Poisson process. The rate of requests and the number of users may also change with time. The edge server takes CPU resources to serve each request from mobile users. When a new request arrives, the edge server has the option to serve it or offload it to other healthy edge server in the cluster. The request is buffered in a queue before it is served. The mathematical model of the edge server and the mobile users is presented below.

1) Edge server: Let $X_t \in \{0, 1, \dots, \mathsf{X}\}$ denote the number of service requests buffered in the queue, where X denotes the size of the buffer. Let $L_t \in \{0, 1, \dots, \mathsf{L}\}$ denote the CPU load at the server where L is the capacity of the CPU. We assume that the CPU has k cores.

We assume that the requests arrive according to a (potentially time-varying) Poisson process with rate λ. If a new request arrives when the buffer is full, the request is offloaded to another server. If a new request arrives when the buffer is not full, the server has the option to either accept or offload the request.

The server can process up to a maximum of k requests from the head of the queue. Processing each request requires CPU resources for the duration in which the request is being served. The required CPU resources is a random variable $R \in \{1, \dots, \mathsf{R}\}$ with probability mass function P. The realization of R is not revealed until the server starts working on the request. The duration of service is exponentially distributed random variable with rate μ.

Let $\mathcal{A} = \{0, 1\}$ denote the action set. Here $A_t = 1$ means that the server decides to offload the request while $A_t = 0$ means that the server accepts the request.

2) Traffic model for mobile users: We consider multiple models for traffic.

- **Scenario 1:** There are N users and all users generate requests according to the same rate λ and the rate does not change over time. Thus, the rate at which requests arrive is λN.
- **Scenario 2:** In this scenario, we assume that there are N users and all users generate requests according to rate λ_{M_t}, where $M_t \in \{1, \dots, \mathsf{M}\}$ is a global state which changes over time. Thus, the rate at which requests arrive in state m, where $m \in M_t$, is $\lambda_m N$.
- **Scenario 3:** As before, there are N users. User n has a state $M_t^n \in \{1, \dots, \mathsf{M}\}$. When the user n is in state m, it generates requests according to rate λ_m. The state M_t^n changes over time. Thus, the rate at which requests arrive at the server is $\sum_{n=1}^{N} \lambda_{M_t^n}$.
- **Time-varying user set:** In each of the scenarios above, we can consider the case when the number N of users is not fixed and changes over time. We call them Scenario 4, 5, and 6 respectively.

All of these traffic models are point processes with increasing degree of complexity. We will focus on theoretical analysis on Scenario 1 but test the performance of the proposed algorithm on all 6 scenarios to highlight its performance in real-world settings.

3) Cost and the optimization framework: The system incurs three types of a cost:

- A holding cost of h per unit time when a request is buffered in the queue but is not being served.
- A running cost of $c(\ell)$ per unit time for running the CPU at a load of ℓ. For example, a running cost $c(\ell) = c_\circ \mathbb{1}\{\ell > \ell_\circ\}$ captures the setting where the system incurs a cost $c_\circ$ whenever the CPU load is greater than $\ell_\circ$.
- A penalty of $p(\ell)$ for offloading a packet at CPU load ℓ. For example, a $p(\ell) = p_\circ \mathbb{1}\{\ell < \ell_\circ\}$ captures the setting where the system incurs a penalty $p_\circ$ for off-loading a packet when the CPU load is less than $\ell_\circ$.

We combine all these costs in a cost function

$$\rho(x, \ell, a) = h[x-k]^+ + c(\ell) + p(\ell)\mathbb{1}\{a = 1\}, \tag{1}$$

where a denotes the action, $[x]^+$ is a short-hand for $\max\{x, 0\}$ and $\mathbb{1}\{\cdot\}$ is the indicator function. Note that to simplify the analysis, we assume that the server always serves $\min\{X_t, k\}$ requests. It is also assumed that $c(\ell)$ and $c(\ell)+p(\ell)$ are weakly increasing in ℓ.

Whenever a new request arrives, the server uses a memory-less policy $\pi\colon \{0, 1, \dots, \mathsf{X}\} \times \{0, 1, \dots, \mathsf{L}\} \to \{0, 1\}$ to choose an action

$$A_t = \pi_t(X_t, L_t).$$

The performance of a policy π starting from initial state (x, ℓ) is given by

$$V^\pi(x, \ell) = \mathbb{E}\left[\int_0^\infty e^{-\alpha t}\rho(X_t, L_t, A_t)dt \,\middle|\, X_0 = x, L_0 = \ell\right], \tag{2}$$

where $\alpha > 0$ is the discount rate and the expectation is with respect to the arrival process, CPU utilization, and service completions.

The objective is to minimize the performance (2) for the different traffic scenarios listed above. We are particularly interested in the scenarios where the arrival rate and potentially other components of the model such as the resource distribution are not known to the system designer and change during the operation of the system.

B. Solution framework

When the model parameters (λ, N, μ, P, k) are known and time-homogeneous, the optimal policy π can be computed using dynamic programming. However, in a real system, these parameters may not be known, so we are interested in developing a RL algorithm which can learn the optimal policy based on the observed per-step cost.

In principle, when the model parameters are known, Scenarios 2 and 3 can also be solved using dynamic programming. However, the state of such dynamic programs will include the state M_t of the system (for Scenario 2) or the states $(M_t^n)_{n=1}^N$ of all users (for Scenario 3). Typically, these states change at a slow time-scale. So, we will consider reinforcement learning algorithms which do not explicitly keep track of the states of the user and verify that the algorithm can adapt quickly whenever the arrival rates change.

III. Dynamic programming to identify optimal admission control policy

When the arrival process is time-homogeneous, the process $\{X_t, L_t\}_{t\ge 0}$ is a finite-state continuous-time MDP controlled through $\{A_t\}_{t\ge 0}$. To specify the controlled transition probability of this MDP, we consider the following two cases.

First, if there is a new arrival at time t, then

$$\mathbb{P}(X_t = x', L_t = \ell' \mid X_{t^-} = x, L_{t^-} = \ell, A_t = a) = \begin{cases} P(\ell' - \ell), & \text{if } x' = x+1 \text{ and } a = 0 \\ 1, & \text{if } x' = x,\ \ell' = \ell, \text{ and } a = 1 \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

We denote this transition function by $q_+(x', \ell' | x, \ell, a)$. Note that the first term $P(\ell' - \ell)$ denotes the probability that the accepted request required $(\ell' - \ell)$ CPU resources.

Second, if there is a departure at time t,

$$\mathbb{P}(X_t = x', L_t = \ell' \mid X_{t^-} = x, L_{t^-} = \ell) = \begin{cases} P(\ell - \ell'), & \text{if } x' = [x-1]^+ \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

We denote this transition function by $q_-(x', \ell' | x, \ell)$. Note that there is no decision to be taken at the completion of a request, so the above transition does not depend on the action.

We combine (3) and (4) into a single controlled transition probability function from state (x, ℓ) to state (x', ℓ') given by

$$p(x', \ell' \mid x, \ell, a) = \frac{\lambda}{\lambda + \min\{x,k\}\mu} q_+(x', \ell' \mid x, \ell, a) + \frac{\min\{x,k\}\mu}{\lambda + \min\{x,k\}\mu} q_-(x', \ell' \mid x, \ell). \tag{5}$$

Let $\nu = \lambda + k\mu$ denote the uniform upper bound on the transition rate at the states. Then, using the *uniformization technique* [15], [16], we can convert the above continuous time discounted cost MDP into a discrete time discounted cost MDP with discount factor $\beta = \nu/(\alpha+\nu)$, transition probability matrix

$$\bar{p}(x', \ell' \mid x, \ell, a) = \frac{\lambda}{\nu} q_+(x', \ell' \mid x, \ell, a) + \frac{\min\{x,k\}\mu}{\nu} q_-(x', \ell' \mid x, \ell) + \left[1 - \frac{\lambda + \min\{x,k\}\mu}{\nu}\right]\mathbb{1}\{(x', \ell') = (x, \ell)\}, \tag{6}$$

and per-step cost

$$\bar{\rho}(x, \ell, a) = \frac{1}{\alpha + \nu}\rho(x, \ell, a).$$

Therefore, we have the following.

Theorem 1 *Consider the following dynamic program*

$$V(x, \ell) = \min\{Q(x, \ell, 0), Q(x, \ell, 1)\} \tag{7}$$

where V is called the value function and

$$\begin{aligned} Q(x,\ell,0) &= \frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell)\big] \\ &\quad + \beta\Bigg[\frac{\lambda}{\nu}\sum_{r=1}^{\mathsf{R}} P(r) V([x+1]_{\mathsf{X}}, [\ell+r]_{\mathsf{L}}) \\ &\qquad + \frac{\min\{x,k\}\mu}{\nu}\sum_{r=1}^{\mathsf{R}} P(r) V([x-1]^+, [\ell-r]^+) \\ &\qquad + \left[1 - \frac{\lambda + \min\{x,k\}\mu}{\nu}\right]V(x, \ell)\Bigg] \end{aligned}$$

and

$$\begin{aligned} Q(x,\ell,1) &= \frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell) + p(\ell)\big] \\ &\quad + \beta\Bigg[\frac{\min\{x,k\}\mu}{\lambda + \min\{x,k\}\mu}\sum_{r=1}^{\mathsf{R}} P(r) V([x-1]^+, [\ell-r]^+) \\ &\qquad + \left[1 - \frac{\lambda + \min\{x,k\}\mu}{\nu}\right]V(x, \ell)\Bigg], \end{aligned}$$

where $[x]_{\mathsf{B}}$ denotes $\min\{x, \mathsf{B}\}$.

Let $\pi(x, \ell) \in \mathcal{A}$ denote the argmin the right hand side of (7). Then, the time-homogeneous policy $\pi(x, \ell)$ is optimal for the original continuous-time optimization problem.

PROOF The proof is present in Appendix A. ■

Thus, for all practical purposes, the decision maker has to solve a discrete-time MDP, where he has to take decisions at the instances when a new request arrives. In the sequel, the per-step cost has been scaled by $1/(\alpha + \nu)$ and we will not write it explicitly.

When the system parameters are known, the above dynamic program can be solved using standard techniques such as value iteration, policy iteration, or linear programming. However, in practice, the system parameters may slowly change over time. Therefore, instead of pursuing a planning solution, we consider reinforcement learning solutions which can adapt to time-varying environments.

IV. STRUCTURE-AWARE REINFORCEMENT LEARNING

Although, in principle, the optimal admission control problem formulated above can be solved using deep RL algorithms, such algorithms require significant computational resources to train, are brittle to the choice of hyperparameters, and generate policies which are difficult to interpret. For the aforementioned reasons, we investigate an alternate class of RL algorithms which circumvents these limitations.

A. Structure of the optimal policy

We first establish basic monotonicity properties of the value function and the optimal policy.

Proposition 1 *For a fixed queue length x, the value function is weakly increasing in the CPU utilization ℓ.*

PROOF The proof is present in Appendix B. ■

Proposition 2 *For a fixed queue length x, if it is optimal to reject a request at CPU utilization ℓ, then it is optimal to reject a request at all CPU utilizations $\ell' > \ell$.*

PROOF The proof is present in Appendix C. ■

B. The SALMUT algorithm

Proposition 2 shows that the optimal policy can be represented by a threshold vector $\tau = (\tau(x))_{x=0}^{\mathsf{X}}$, where $\tau(x) \in \{0, \dots, \mathsf{L}\}$ is the smallest value of the CPU utilization such that it is optimal to accept the packet for CPU utilization less than or equal to $\tau(x)$ and reject it for utilization greater than $\tau(x)$.

The SALMUT algorithm was proposed in [19] to exploit a similar structure in admission control for multi-class queues. It was originally proposed for the average cost setting. In the remainder of this section, we present a generalization to the discounted-time setting.

We use π_τ to denote a threshold-based policy with the parameters $(\tau(x))_{x=0}^{\mathsf{X}}$ taking values in $\{0, \dots, \mathsf{L}\}^{\mathsf{X}+1}$. The key idea behind SALMUT is that, instead of deterministic threshold-based policies, we consider a random policy parameterized with parameters taking value in the compact set $[0, \mathsf{L}]^{\mathsf{X}+1}$. Then, for any state (x, ℓ), the randomized policy π_τ chooses action $a = 0$ with probability $f(\tau(x), \ell)$ and chooses action $a = 1$ with probability $1 - f(\tau(x), \ell)$, where $f(\tau(x), \ell)$ is any continuous decreasing function w.r.t ℓ, which is differentiable in its first argument, e.g., the sigmoid function

$$f(\tau(x), \ell) = \frac{\exp((\tau(x) - \ell)/\theta)}{1 + \exp((\tau(x) - \ell)/\theta)}, \tag{8}$$

where $\theta > 0$ is a hyper-parameter (often called "temperature").

Let $p_\tau(x', \ell' | x, \ell)$ denote the transition function under policy π_τ, i.e.

$$\begin{aligned} p_\tau(x', \ell' | x, \ell) &= f(\tau(x), \ell)\bar{p}(x', \ell' | x, \ell, 0) \\ &\quad + (1 - f(\tau(x), \ell))\bar{p}(x', \ell' | x, \ell, 1) \end{aligned} \tag{9}$$

Similarly, let $\bar{\rho}_\tau(x, \ell)$ denote the expected per-step cost under policy π_τ, i.e.,

$$\bar{\rho}_\tau(x, \ell) = f(\tau(x), \ell)\bar{\rho}(x, \ell, 0) + (1 - f(\tau(x), \ell))\bar{\rho}(x, \ell, 1). \tag{10}$$

Let ∇ denote the gradient with respect to τ.

If the initial state (x_0, ℓ_0) is known, we consider $J(\tau)$ as the performance of policy π_τ. From Performance Derivative formula [23, Eq. 2.44], we know that

$$\begin{aligned} \nabla J(\tau) = \frac{1}{1-\beta} \sum_{x=0}^{\mathsf{X}} \sum_{\ell=0}^{\mathsf{L}} \mu(x, \ell) \big[\nabla \bar{\rho}_\tau(x, \ell) \\ + \beta \sum_{x'=0}^{\mathsf{X}} \sum_{\ell'=0}^{\mathsf{L}} \nabla p_\tau(x', \ell' | x, \ell) V_\tau(x', \ell') \big] \end{aligned} \tag{11}$$

where $\mu(x, \ell)$ is the occupancy measure on the states starting from the initial state (x_0, ℓ_0).

From (9), we get that

$$\begin{aligned} \nabla p_\tau(x', \ell' | x, \ell) = (\bar{p}(x', \ell' | x, \ell, 0) - \bar{p}(x', \ell' | x, \ell, 1)) \\ \nabla f(\tau(x), \ell). \end{aligned} \tag{12}$$

Similarly, from (10), we get that

$$\nabla \bar{\rho}_\tau(x, \ell) = (\bar{\rho}(x, \ell, 0) - \bar{\rho}(x, \ell, 1)) \nabla f(\tau(x), \ell). \tag{13}$$

Substituting (9) and (10) in (11) and simplifying, we get

$$\nabla J(\tau) = \frac{1}{1-\beta} \sum_{x=0}^{\mathsf{X}} \sum_{\ell=0}^{\mathsf{L}} \mu(x, \ell) [\Delta Q(x, \ell)] \nabla f(\tau(x), \ell), \tag{14}$$

where $\Delta Q(x, \ell) = Q(x, \ell, 0) - Q(x, \ell, 1)$.

Therefore, when (x, ℓ) is sampled from the stationary distribution μ, an unbiased estimator of $\nabla J(\tau)$ is proportional to $\Delta Q(x, \ell) \nabla f(\tau(x), \ell)$.

Thus, we can use the standard two time-scale Actor-Critic algorithm [8] to simultaneously learn the policy parameters τ and the action-value function Q as follows. We start with an initial guess Q_0 and τ_0 for the action-value function and the optimal policy parameters. Then, we update the action-value function using temporal difference learning:

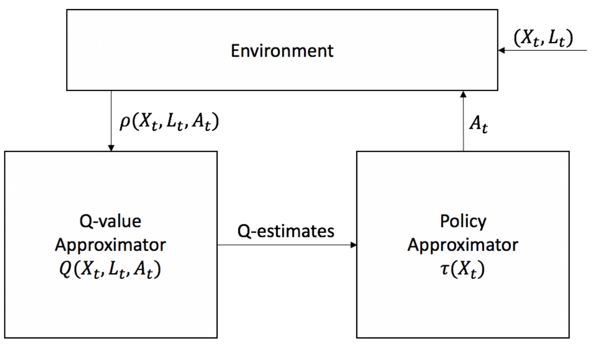

Environment
(X_t, L_t)
$\rho(X_t, L_t, A_t)$
A_t
Q-value Approximator $Q(X_t, L_t, A_t)$
Q-estimates
Policy Approximator $\tau(X_t)$

Fig. 3. Illustration of the two-timescale SALMUT algorithm.

$$Q_{n+1}(x, \ell, a) = Q_n(x, \ell, a) + b_n^1\big[\bar{\rho}(x, \ell, a) + \beta \min_{a' \in A} Q_n(x', \ell', a') - Q_n(x, \ell, a)\big], \quad (15)$$

and update the policy parameters using stochastic gradient descent while using the unbiased estimator of $\nabla J(\tau)$:

$$\tau_{n+1}(x) = \mathrm{Proj}\big[\tau_n(x) + b_n^2 \nabla f(\tau(x), \ell)\Delta Q(x, \ell)\big], \quad (16)$$

where Proj is a projection operator which clips the values to the interval $[0, \mathsf{L}]$ and $\{b_n^1\}_{n\geq 0}$ and $\{b_n^2\}_{n\geq 0}$ are learning rates which satisfy the standard conditions on two time-scale learning: $\sum_n b_n^k = \infty$, $\sum_n (b_n^k)^2 < \infty$, $k \in \{1, 2\}$, and $\lim_{n\to\infty} b_n^2/b_n^1 = 0$.

```
Algorithm 1: Two time-scale SALMUT algorithm
  Result: $\tau$
  Initialize action-value function $\forall x, \forall \ell$, $Q(x, \ell, a) \leftarrow 0$
  Initialize threshold vector $\forall x$, $\tau(x) \leftarrow \mathrm{rand}(0, \mathsf{L})$
  Initialize start state $(x, \ell) \leftarrow (x_0, \ell_0)$
  while TRUE do
      if EVENT == ARRIVAL then
          Choose action $a$ according to Eq. (8)
          Observe next state $(x', \ell')$
          Update $Q(x, \ell, a)$ according to Eq. (15)
          Update threshold $\tau$ using Eq. (16)
          $(x, \ell) \leftarrow (x', \ell')$
      end
  end
```

The complete algorithm is presented in Algorithm 1 and illustrated in Fig. 3.

Theorem 2 *The two time-scale SALMUT algorithm described above converges almost surely and* $\lim_{n\to\infty} \nabla J(\tau_n) = 0$.

PROOF The proof is present in Appendix D. ∎

Remark 1 The idea of replacing the "hard" threshold $\tau(x) \in \{0, \ldots, \mathsf{L}\}^{\mathsf{X}+1}$ with a "soft" threshold $\tau(x) \in [0, \mathsf{L}]^{\mathsf{X}+1}$ is same as that of the SALMUT algorithm [19]. However, our simplification of the performance derivative (11) given by (14) is conceptually different from the simplification presented in [19]. The simplification in [19] is based on viewing $\sum_{x'=0}^{\mathsf{X}} \sum_{\ell'=0}^{\mathsf{L}} \nabla p_\tau(x', \ell'|x, \ell) V_\tau(x', \ell')$ term in (11) as

$$2\mathbb{E}\big[(-1)^\delta \nabla f(\tau(x), \ell) V_\tau(\hat{x}, \hat{\ell})\big]$$

where $\delta \sim \mathrm{Unif}\{0, 1\}$ is an independent binary random variable and $(\hat{x}, \hat{\ell}) \sim \delta p(\cdot|x, \ell, 0) + (1 - \delta)p(\cdot|x, \ell, 1)$. In contrast, our simplification is based on a different algebric simplification that directly simplifies (11) without requiring any additional sampling. Therefore, the algorithm presented here will have less variance compared to the algorithm in [19].

Remark 2 The update of the Q-function in (15) is standard temporal difference learning. At each step, a single state (x, ℓ) is updated and the update requires a maximization over 2 actions. Therefore, the computational complexity of implementing (15) is $\mathcal{O}(1)$. The update of the policy parameters in (16) requires $\mathcal{O}(1)$ operations. Therefore, the computation complexity of implementing (16) is also $\mathcal{O}(1)$. Therefore, the per-iteration complexity of the Algorithm 1 is $\mathcal{O}(1)$. The policy of SALMUT is completely characterized by the threshold vector τ. In addition to the vector τ, we need to store the Q-value function of every state-action pair. This results in a storage complexity of $O(2\mathsf{XL})$.

V. Numerical experiments - Computer Simulations

In this section, we present detailed numerical experiments to evaluate the proposed reinforcement learning algorithm on various scenarios described in Sec. II-A.

We consider an edge server with buffer size $\mathsf{X} = 20$, CPU capacity $\mathsf{L} = 20$, $k = 2$ cores, service-rate $\mu = 3.0$ for each core, holding cost $h = 0.12$. The CPU capacity is discretized into 20 states for utilization 0 – 100%, with $\ell = 0$ corresponding to a state with CPU load $\ell \in [0\% - 5\%)$, and so on.

The CPU running cost is modelled such that it incurs a positive reinforcement for being in the optimal CPU range, and a high cost for an overloaded system.

$$c(\ell) = \begin{cases} 0 & \text{for } \ell \leq 5 \\ -0.2 & \text{for } 6 \leq \ell \leq 17 \\ 10 & \text{for } \ell \geq 18 \end{cases}$$

The offload penalty is modelled such that it incurs a small fixed cost to enable the offloading behavior only when the system is loaded. It also incurs a very high cost when the system is idle to discourage offloading in such scenarios.

$$p(\ell) = \begin{cases} 1 & \text{for } \ell \geq 3 \\ 10 & \text{for } \ell \leq 3 \end{cases}$$

The probability mass function of resources requested per request is as follows

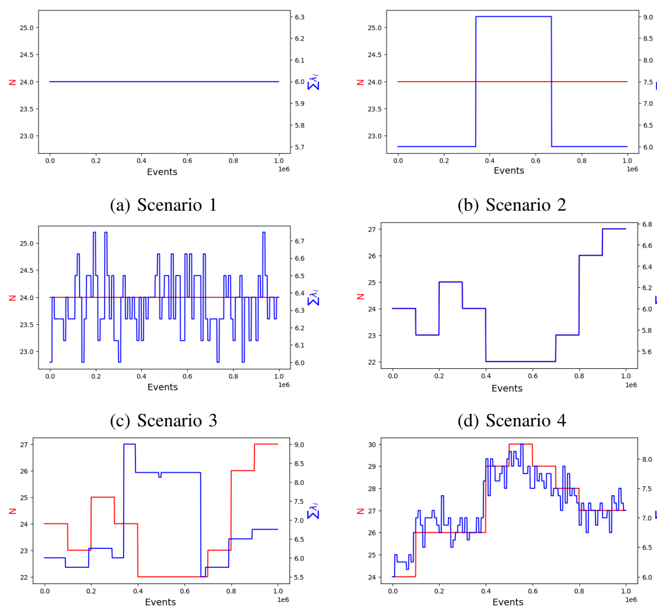
$$P(r) = \begin{cases} 0.6 & \text{if } r = 1 \\ 0.4 & \text{if } r = 2 \end{cases}.$$

Rather than simulating the system in continuous-time, we simulate the equivalent discrete-time MDP by generating the next event (arrival or departure) using a Bernoulli distribution with probabilities and costs described in Sec. III. See Sec. V-C for details. We assume that the discrete time discount factor

$\beta = \alpha/(\alpha+\nu)$ equals 0.95 and assume it has been scaled by $1/(\alpha+\nu)$ and do not write it explicitly.

A. Simulation scenarios

We consider a number of traffic scenarios which are increasing in complexity and closeness to the real-world setting. Each scenario runs for a $T = 10^6$ events. The scenarios capture variation in the transmission rate and the number of users over time, their realization can be seen in Fig. 4.



(a) Scenario 1 (b) Scenario 2

(c) Scenario 3 (d) Scenario 4

(e) Scenario 5 (f) Scenario 6

Fig. 4. The evolution of λ and N for the different scenarios that we described. In scenarios 1 and 4, λ and N overlap in the plots.

The evolution of the arrival rate λ and the number of users N for the more dynamic environments is shown in Fig. 4.

a) Scenario 1: This scenario tests how the learning algorithms perform in the time-homogeneous setting. We consider a system with $N = 24$ users with arrival rate $\lambda_i = 0.25$. Thus, the overall arrival rate $\lambda = N\lambda_i = 6$.

b) Scenario 2: This scenario tests how the learning algorithms adapt to occasional but significant changes to arrival rates. We consider a system with $N = 24$ users, where each user generates requests at rate $\lambda_{\text{low}} = 0.25$ for the interval $(0, 3.33 \times 10^5]$, then generates requests at rate $\lambda_{\text{high}} = 0.375$ for the interval $(3.34 \times 10^5, 6.66 \times 10^5]$, and then generates requests at rate λ_{low} again for the interval $(6.67 \times 10^5, 10^6]$.

c) Scenario 3: This scenario tests how the learning algorithms adapt to frequent but small changes to the arrival rates. We consider a system with $N = 24$ users, where each user generates requests according to rate $\lambda \in \{\lambda_{\text{low}}, \lambda_{\text{high}}\}$ where $\lambda_{\text{low}} = 0.25$ and $\lambda_{\text{high}} = 0.375$. We assume that each user starts with a rate λ_{low} or λ_{high} with equal probability. At time intervals $m \times 10^4$, each user toggles its transmission rate with probability $p = 0.1$.

d) Scenario 4: This scenario tests how the learning algorithm adapts to change in the number of users. In particular, we consider a setting where the system starts with $N_1 = 24$ user. At every 10^5 time steps, a user may leave the network, stay in the network or add another mobile device to the network with probabilities 0.05, 0.9, and 0.05, respectively. Each new user generates requests at rate λ.

e) Scenario 5: This scenario tests how the learning algorithm adapts to large but occasional change in the arrival rates and small changes in the number of users. In particular, we consider the setup of Scenario 2, where the number of users change as in Scenario 4.

f) Scenario 6: This scenario tests how the learning algorithm adapts to small but frequent change in the arrival rates and small changes in the number of users. In particular, we consider the setup of Scenario 3, where the number of users change as in Scenario 4.

B. The RL algorithms

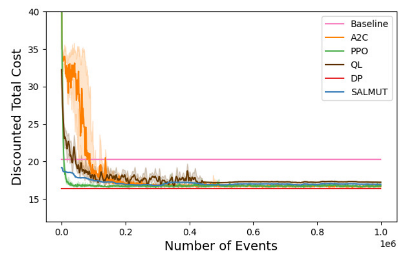
For each scenarios, we compare the performance of the following policies

1) Dynamic Programming (DP), which computes the optimal policy using Theorem 1.
2) SALMUT, as described in Sec. IV-B.
3) Q-Learning (QL), a model-free reinforcement learning algorithm to learn the value of an action in a particular state using (15) and chooses an action using ϵ-greedy policy with $\epsilon = 0.001$.
4) PPO [20], which is a family of trust region policy gradient method and optimizes a surrogate objective function using stochastic gradient ascent.
5) A2C [21], which is a two time-timescale learning algorithms where the critic estimates the value function and actor updates the policy distribution in the direction suggested by the critic.
6) Baseline, which is a fixed-threshold based policy, where the node accepts requests when $\ell < 17$ and offloads requests otherwise. We ran the baseline policy for multiple thresholds ranging from $\ell = 14$ to $\ell = 19$ and observed that $\ell = 17$ gives us the best performance. Such static policies are currently deployed in many real-world systems.

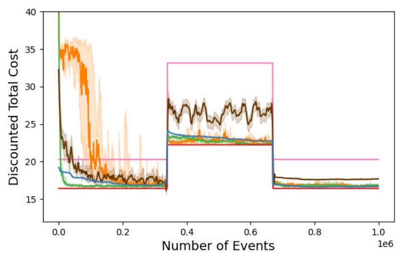
For SALMUT, we use ADAM [24] optimizer with initial learning rates $(b^1 = 0.03, b^2 = 0.002)$. For Q-learning, we use Stochastic Gradient Descent with $b^1 = 0.01$. We used the stable-baselines [25] implementation of PPO and A2C with learning rates 0.0003 and 0.001 respectively.

C. Simulation Details

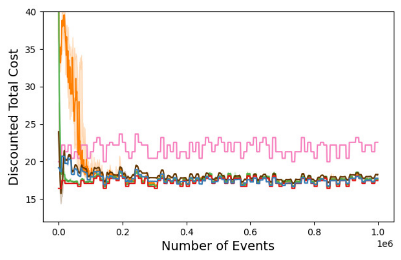
We train each RL algorithm described above for $T = 10^6$ events. To simulate the continuous time system in discrete time, we generate events corresponding to the arrival or the completion of requests as described in the dynamic program of Theorem 1. In particular, we generate a sequence $\{z_t\}_{t=1}^{T}$ of independent Unif[0, 1] random variables. At discrete-time t, we set the event to be an arrival event if z_t is less than or equal to $\lambda_t/(\lambda_t + k\mu)$; we set the event to be a departure event if z_t is greater than $\lambda_t/(\lambda_t + k\mu)$ but less than or equal to $(\lambda_t + \min\{x_t, k\}\mu)/(\lambda + k\mu)$; otherwise, we set the event to be



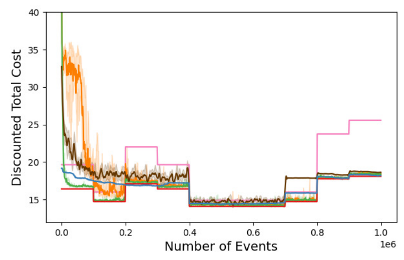
(a) Scenario 1



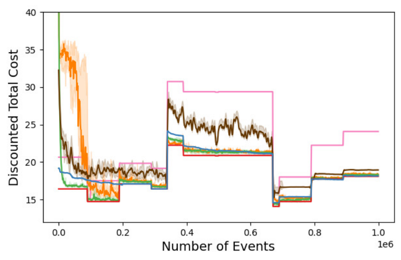
(b) Scenario 2



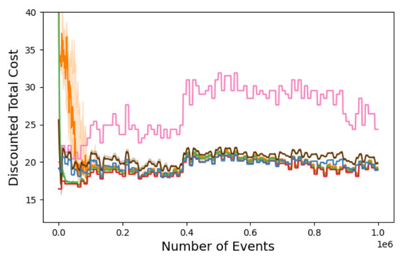
(c) Scenario 3



(d) Scenario 4



(e) Scenario 5



(f) Scenario 6

Fig. 5. Performance of RL algorithms for different scenarios. The legend shown in (a) applies to all subfigures.

a null event (so that the state does not change). We train all RL algorithms on the same sequence $\{z_t\}_{t-1}^{T}$.

We store the policy parameters obtained during training and evaluate their performance every 10^3 events as follows: for each evaluation, we do 100 independent rollouts of $H = 1000$ events (generated in the same manner as above), compute the (discrete-time) expected discounted cost over horizon H for each rollout, and use the median across the 100 rollouts as an estimate for the performance.

We then repeat the end-to-end training (i.e., generating the sequence $\{z_t\}_{t=1}^{T}$, training all RL algorithms, and estimating the performance after 10^3 events) for 10 random seeds and generate summary statistics of the performance estimated at each evaluation step. The summary statistic consists of the median and the uncertainty band from the first to the third quartile (for the 10 random seeds). The plot of the median and the uncertainty band versus the number of events is shown in Fig. 5.

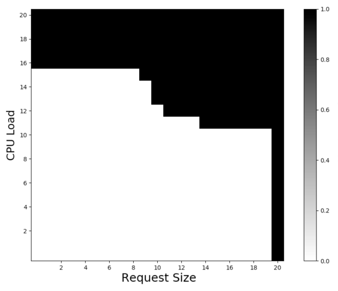
D. Analysis of the Results

For Scenario 1, all RL algorithms (SALMUT, Q-learning, PPO, A2C) converge to a close-to-optimal policy and remain stable after convergence. Since all policies converge quickly, SALMUT, PPO, and A2C are also able to adapt quickly in Scenarios 2–6 and keep track of the time-varying arrival rates and number of users. There are small differences in the performance of the RL algorithms, but these are minor. Note that, in contrast, Q-learning policy does not perform well when the dynamics of the requests changes drastically, whereas the baseline policy performs poorly when the server is overloaded.

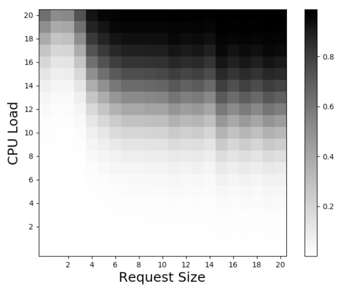
The plots for Scenario 1 (Fig. 5a) show that PPO converges to the optimal policy in less than 10^5 events, SALMUT and A2C takes around 2×10^5 events, whereas Q-learning takes around 5×10^5 events to converge. The policies for all the algorithms remain stable after convergence. Upon further analysis on the structure of the optimal policy, we observe that the structure of the optimal policy of SALMUT (Fig. 6b) differs from that of the optimal policy computed using DP (Fig. 6a). There is a slight difference in the structure of these policies when buffer size (x) is low and CPU load (ℓ) is high, which occurs because these states are reachable with a very low probability and hence SALMUT doesn't encounter these states in the simulation often to be able to learn the optimal policy in these states. The plots from Scenario 2 (Fig. 5b) show similar behavior in the intervals where λ is constant. When λ changes significantly, we observe all RL algorithms except Q-learning are able to adapt to the drastic but stable changes in the environment. Once the load stabilizes, all the algorithms are able to readjust to the changes and perform close to the optimal policy. The plots from Scenario 3 (Fig. 5c) show similar behavior to Scenario 1, i.e., small but frequent changes in the environment do not impact the learning performance of reinforcement learning algorithms.

The plots from Scenario 4-6 (Fig. 5d-5f) show consistent performance with varying users. The RL algorithms including Q-learning show similar performance for most of the time-steps except in Scenario 5, which is similar to the behavior observed in Scenario 2. The Q-learning algorithm also performs poorly when the load suddenly changes in Scenarios 4 and 6. This could be due to the fact that Q-learning takes longer to adjust to a more aggressive offloading policy.

Remark 3 Only scenario 1 satisfies the Poisson arrival assumptions of our model. Scenario 2 is a Cox process, and the other scenarios may be viewed as general point processes. Nonetheless, the results show that the SALMUT algorithm performs competitively against the more general reinforcement learning algorithms.



(a) DP $\lambda = 6$



(b) SALMUT $\lambda = 6$

Fig. 6. Comparing the optimal policy and converged policy of SALMUT along one of the sample paths. The colorbar represents the probability of the offloading action.

E. Analysis of Training Time and Policy Interpretability

The main difference among these four RL algorithms is the training time and interpretability of policies. We ran our experiments on a server with Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz processor. The training time of all the RL algoirthms is shown in Table II. The mean training time is computed based on a single scenario over different runs and averaged across all the six scenarios. SALMUT is about 28 times faster to train than PPO and 17 times faster than A2C. SALMUT does not require a non-linear function approximator such as Neural Networks (NN) to represent its policy, making the training time for SALMUT very fast. We observe that Q-learning is around 1.5 times faster than SALMUT as it does not need to update its policy parameters separately. Even though Q-learning is faster than SALMUT, Q-learning does not converge quickly to an optimal policy when the request distribution changes.

TABLE II
TRAINING TIME OF RL ALGORITHMS

Algorithm	Mean Time (s)	Std-dev (s)
SALMUT	98.23	4.86
Q-learning	62.73	1.57
PPO	2673.17	23.33
A2C	1677.33	9.99

By construction, SALMUT searches for (randomized) threshold based policies. For example, for Scenario 1, SALMUT converges to the policy shown in Fig. 6b.

It is easy for a network operator to interpret such threshold based strategies and decide whether to deploy them or not. In contrast, in deep RL algorithms such as PPO and A2C, the policy is parameterized using a neural network and it is difficult to visualize the learned weights of such a policy and decide whether the resultant policy is reasonable. Thus, by leveraging on the threshold structure of the optimal policy, SALMUT is able to learn faster and at the same time provide threshold based policies which are easier to interpret.

The policy of SALMUT is completely characterized by the threshold vector τ, making it storage efficient too. The threshold-nature of the optimal policy computed by SALMUT, can be easily interpreted by just looking at the threshold vector τ (see Fig. 6b), making it easy to debug and estimate the behavior of the system operating under such policies. However, the policies learned by A2C and PPO are the learned weights of the NN, which are undecipherable and may lead to unpredictable results occasionally. It is very important that the performance of real-time systems be predictable and reliable, which has hindered the adoption of NNs in real-time deployments.

F. Behavioral Analysis of Policies

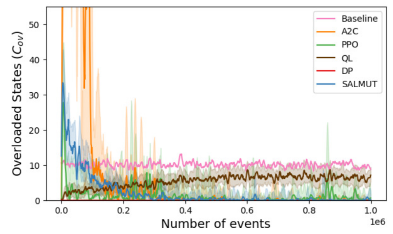
We performed further analysis on the behavior of the learned policy by observing the number of times the system enters into an overloaded state and offloads incoming request. For each training run (i.e., random seed), we divide the 10^6 training events into intervals of size 1000. For each interval we count the number C_{ov} of times the system enters into an overload state and the number C_{off} of times the system offloads new requests. We repeat these calculations for each of the 10 random seeds and compute the summary statistics of C_{ov} and C_{off}. The median and the uncertainty band from the first to the third quartile is plotted in Figs. 8 and 7.

We observe in Fig. 7, that all algorithms learn not to enter into the overloaded state. As seen in the case of total discounted cost (Fig. 5), PPO learns it instantly, followed by SALMUT, and A2C. The policy learned by A2C is such that it enters the overloaded states many times during the initial stages of training and learns a good policy after 0.8×10^6, the policy is drastically improved and we can see the behavior similar to other RL algorithms. On the other hand, Q-learning policy tend to perform more offloading initially and eventually learns a good balance between the two actions. The observation is valid for all the different scenarios we tested. We observe that for Scenario-4, PPO enters the overloaded state at around 0.8×10^6 which is due to the fact the $\sum_i \lambda_i$ increases drastically at that point (seen in Fig. 4d) and we also see its effect on the cost in Fig. 5d at that time. We also observe that SALMUT enters into overloaded states when the request distribution changes drastically in Scenario 2 and 5. It is able to recover quickly and adapt its threshold policy to a more aggressive offloading policy. The baseline algorithms, on the other hand, enters into the overloaded state quiet often, whereas Q-learning enters overloaded state but not as often as the baseline algorithm.

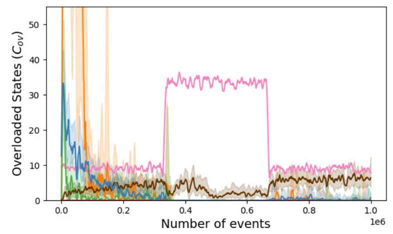
We observe in Fig. 8, that all RL algorithms learn to adjust their offloading rate to avoid overloaded state. The number of times requests have been offloaded is directly proportional to the total arrival rate of all the users at that time. When the arrival rate increases, the number of times the offloading occurs also increases in the interval. We perform further analysis of this behavior for the docker-testbed (see Fig. 13) and the results are similar for the simulations too.

VI. Testbed Implementation and Results

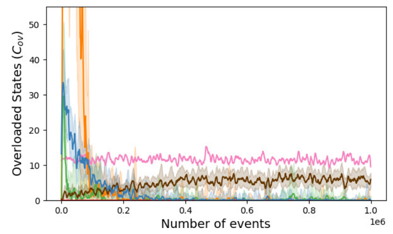
We test our proposed algorithm on a testbed resembling the MEC architecture in Fig. 1, but without the core network and backend cloud server for simplicity. We consider an edge node which serves a single application. Both the edge nodes and clients are implemented as containerized environments in a virtual machine. The overview of the testbed is shown in



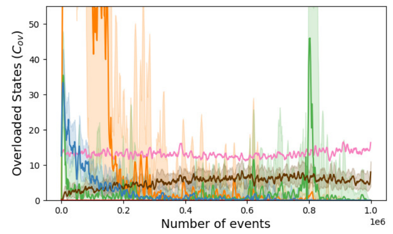
(a) Scenario 1



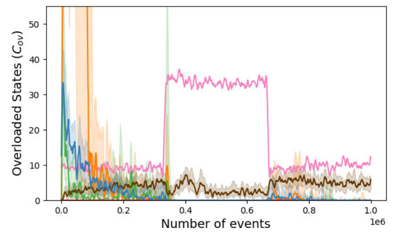
(b) Scenario 2



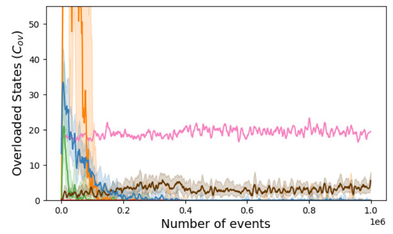
(c) Scenario 3



(d) Scenario 4

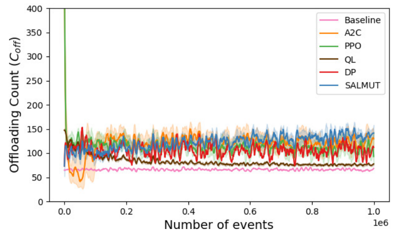


(e) Scenario 5

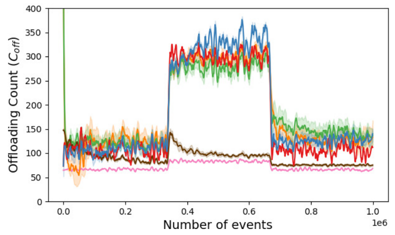


(f) Scenario 6

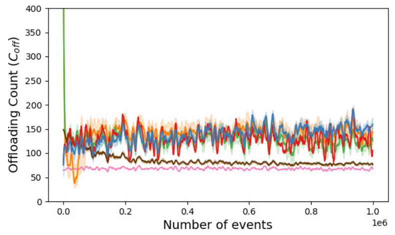
Fig. 7. Comparing the number of times the system goes into the overloaded state at each evaluation step. The trajectory of event arrival and departure is fixed for all evaluation steps and across all algorithms for the same arrival distribution. The C_{ov} for A2C algorithm reach upto 140 during initial stages and have been cut off from the figures to have a better clarity on the rest of the results. The legend shown in (a) applies to all subfigures.



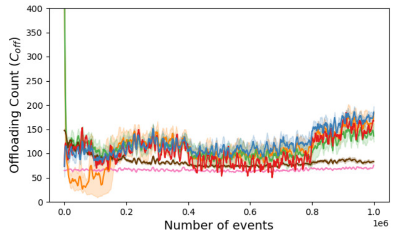
(a) Scenario 1



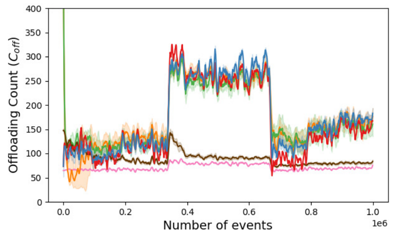
(b) Scenario 2



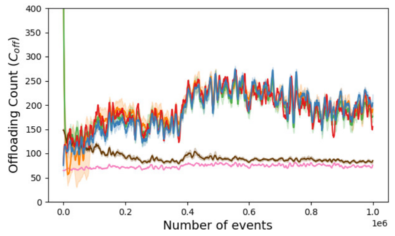
(c) Scenario 3



(d) Scenario 4



(e) Scenario 5



(f) Scenario 6

Fig. 8. Comparing the number of times the system performs offloading at each evaluation step. The trajectory of event arrival and departure is fixed for all evaluation steps and across all algorithms for the same arrival distribution.The legend shown in (a) applies to all subfigures.

Fig. 9. The load generator generates requests for each client independently according to a time-varying Poisson process. The requests at the edge node are handled by the controller which decides either to accept the request or offload the request based on the policy for the current state of the edge node. If the action is "accept", the request is added to the request queue of the edge node, otherwise the request is offloaded to another healthy edge node via the proxy network. The Key Performance Indicator (KPI) collector copies the KPI metrics into a database at regular intervals. The RL modules uses these metrics to update its policies. The Subscriber/Notification (Sub/Notify) module notifies the controller about the updated policy. The controller now uses the updated policy to serve all future requests.

We consider an edge server with buffer size $\mathsf{X} = 20$, CPU capacity $\mathsf{L} = 20$, $k = 2$ cores, service-rate $\mu = 3.0$ for each core, holding cost $h = 0.12$. The CPU capacity is discretized into 20 states for utilization $0 - 100\%$, similar to the previous experiment. The CPU running cost is $c(\ell) = 30$ for $\ell \geq 18$, $c(\ell) = -0.2$ for $6 \leq \ell \leq 17$, and $c(\ell) = 0$ otherwise. The offload penalty is $p = 1$ for $\ell \geq 3$ and $p = 10$ for $\ell < 3$. The baseline algorithm accepts request when $\ell < 18$ and offloads otherwise. We did not perform a hyperparamater search over the baseline thresholds as it is too expensive. We assume that the discrete time discount factor $\beta = \alpha/(\alpha + \nu)$ equals 0.99.

We simulate the system for 10^5 seconds (approx 27 hours 46 minutes). We divided the total time into 1000 steps of 100 seconds each. In Scenarios $2 - 6$, we do not change the arrival

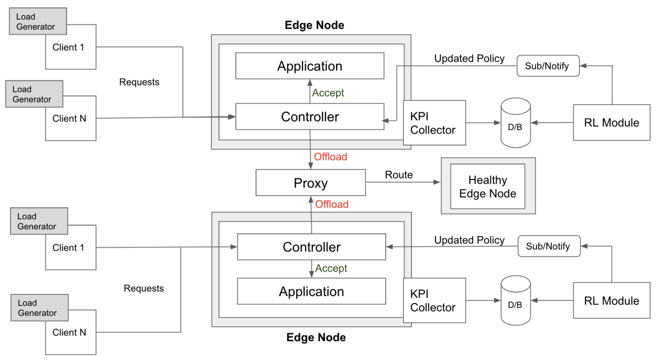


Fig. 9. The overview of the docker-testbed environment.

rate in the middle of the 100 second interval.

We generate the service requests using continuous time point process where the arrival rate λ_t changes as per Fig. 4. Each request runs a workload on the edge node and consumes CPU resources R, where R is a random variable. The states, actions, costs, next states for each step are stored in a buffer in the edge node. After the completion of a step, the KPI collector copies these buffers into a database. The RL module is then invoked, which loads its most recent policy and other parameters, and trains on this new data to update its policy. Once the updated policy is generated, it is copied in the edge node and is used by the controller for serving the requests for the next step.

We do not run neural network based RL algorithms in our testbed as these algorithms cannot be trained in real-time because the time they require to process each sample is more than the sampling interval. So, we only compare with the baseline algorithm. For SALMUT, we store the policy parameters every step and evaluate their performance by running 10 independent rollouts for 100 seconds. The experiment is repeated for 5 random seeds and the median performance with an uncertainty band from the first to the third quartile are plotted in Fig. 10 along with the total request arrival rate ($\sum_i \lambda_i$) in gray dotted lines.

A. Analysis of the Results

For Scenario 1 (Fig. 10a), we observe that the SALMUT algorithm outperforms the baseline algorithm right from the start, indicating that SALMUT updates its policy swiftly at the start and slowly converges towards optimal performance, whereas the baseline incurs high cost throughout. Since SALMUT policies converge towards optimal performance after some time, they are also able to adapt quickly in Scenarios 2–6 (Fig. 10b-10f) and keep track of the time-varying arrival rates and number of users. We observe that SALMUT takes some time to learn a good policy, but once it learns the policy, it adjusts to frequent but small changes in λ and N very well (see Fig. 10c and 10d). If the request rate changes drastically, the performance decreases a little (which is bound to happen as the total requests to process are much larger than the server's capacity) but the magnitude of the performance drop is much lesser in SALMUT as compared to the baseline, seen in Fig. 10b, 10e and 10f. It is because the baseline algorithms incur high overloading cost for these requests whereas SALMUT incurs offloading costs for the same requests. Further analysis on this is present in Section 4.2.2.

B. Behavioral Analysis

We perform behavior analysis of the learned policy by observing the number of times the system enters into an overloaded state (denoted by C_{ov}) and the number of incoming request offloaded by the edge node (denoted by C_{off}) in a window of size 100. These plots are shown in Fig. 11 & 12.

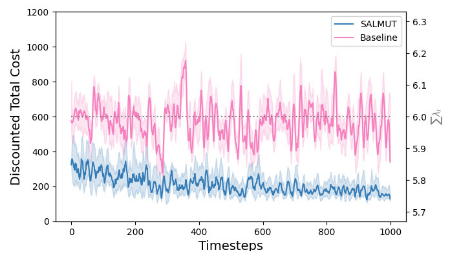
We observe from Fig. 11 that the number of times the edge node goes into an overload state while following policy executed by SALMUT is much less than the baseline algorithm. Even when the system goes into an overloaded state, it is able to recover quickly and does not suffer from performance deterioration. From Fig. 12b and 12e we can observe that in Scenarios 2 and 5, when the request load increases drastically (at around 340 steps), C_{off} increases and its effects can also be seen in the overall discounted cost in Fig. 10b and 10e at around the same time. SALMUT is able to adapt its policy quickly and recover quickly. We observe in Fig. 12 that SALMUT performs more offloading as compared to the baseline algorithm.

A policy that offloads often and does not go into an overloaded state may not necessarily minimize the total cost. We did some further investigation by visualizing the scatter-plot (Fig. 13) of the overload count (C_{ov}) on the y-axis and the offload count (C_{off}) on the x-axis for both SALMUT and the baseline algorithm for all the scenarios described in Fig. 4. We observe that SALMUT keeps C_{ov} much lower than the baseline algorithm at the cost of increased C_{off}. We observe from Fig. 13 that the slope for the plot is linear for baseline algorithms because they are offloading reactively. SALMUT, on the other hand, learns a behavior that is analogous to pro-active offloading, where it benefits from the offloading action it takes by minimizing C_{ov}.

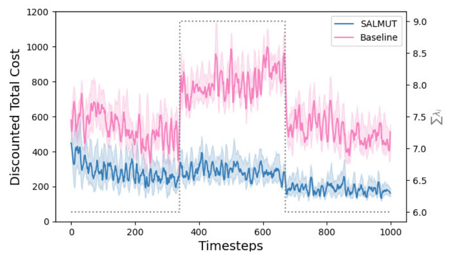
VII. Conclusion

In this paper we considered a single node optimal policy for overload protection on the edge server in a time varying environment. We proposed a RL-based adaptive low-complexity admission control policy that exploits the structure of the optimal policy and finds a policy that is easy to interpret. Our proposed algorithm performs as well as the standard deep RL algorithms but has a better computational and storage complexity, thereby significantly reducing the total training time. Therefore, our proposed algorithm is more suitable for deployment in real systems for online training.

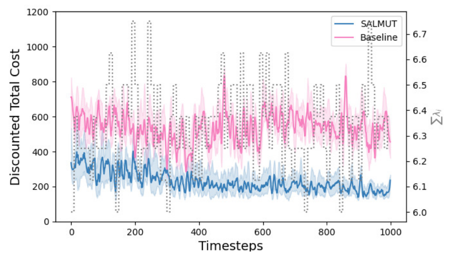
The results presented in this paper can be extended in several directions. In addition to CPU overload, one could consider other resource bottlenecks such as disk I/O, RAM utilization, etc. It may be desirable to simultaneously consider multiple resource constraints. Along similar lines, one could consider multiple applications with different resource requirements and different priority. If it can be established that the optimal policy in these more sophisticated setup has a threshold structure similar to Proposition 2, then we can apply the framework developed in this paper.



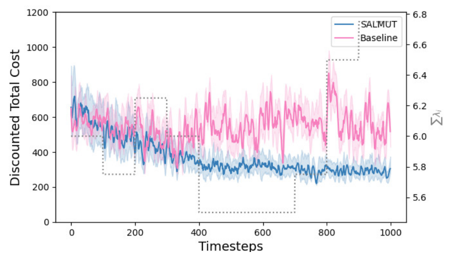
(a) Scenario 1



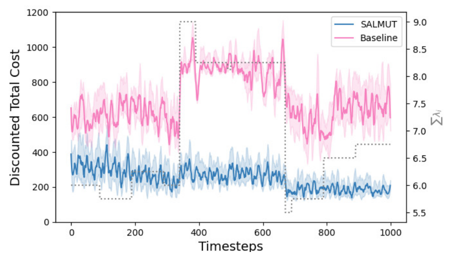
(b) Scenario 2



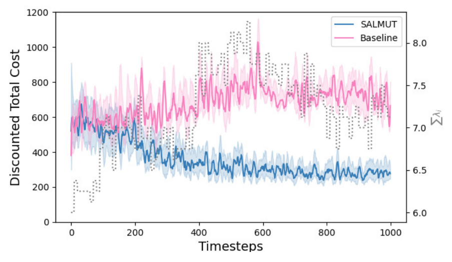
(c) Scenario 3



(d) Scenario 4

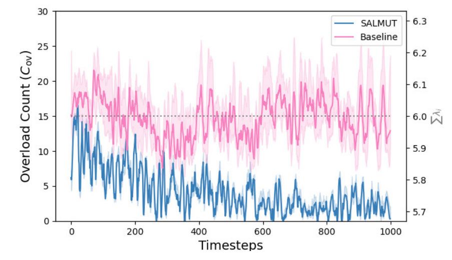


(e) Scenario 5

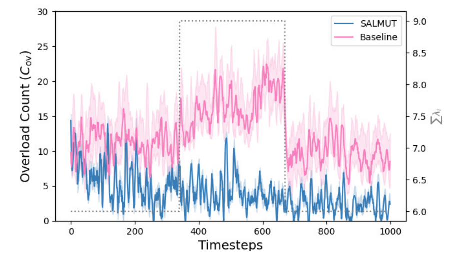


(f) Scenario 6

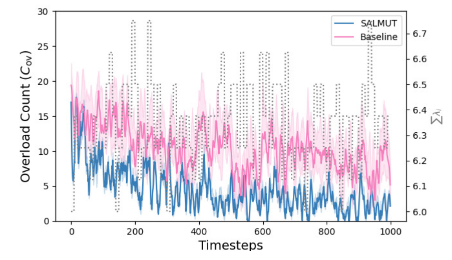
Fig. 10. Performance of RL algorithms for different scenarios in the end-to-end testbed we created. The y-axis represents the total discounted cost incurred by the system and the x-axis represent the step described in Section VI. We also plot the total request arrival rate ($\sum_i \lambda_i$) on the right-hand side of y-axis in gray dotted lines.



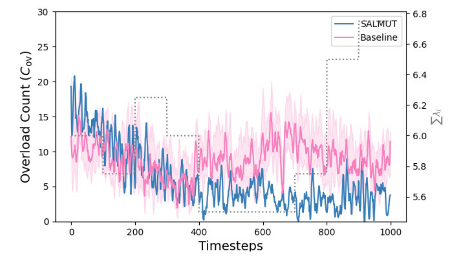
(a) Scenario 1



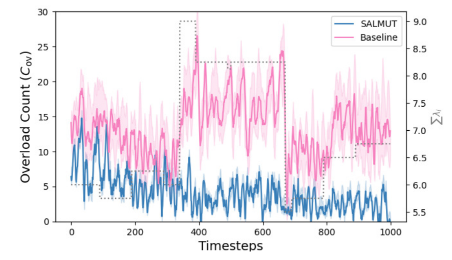
(b) Scenario 2



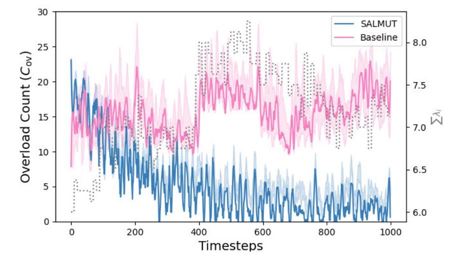
(c) Scenario 3



(d) Scenario 4



(e) Scenario 5



(f) Scenario 6

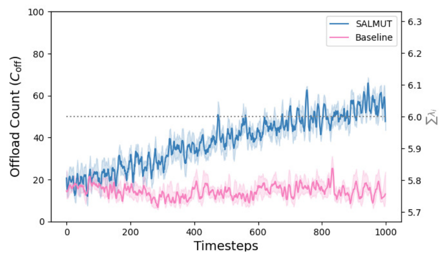
Fig. 11. Comparing the number of times the system goes into the overloaded state at each step in the end-to-end testbed we created. The y-axis represents the total discounted cost incurred by the system and the x-axis represent the step described in Section VI. We also plot the total request arrival rate ($\sum_i \lambda_i$) on the right-hand side of y-axis in gray dotted lines.

The discussion in this paper was restricted to a single node. These results could also provide a foundation to investigate node overload protection in multi-node clusters where additional challenges such as routing, link failures, and network topology shall be considered.

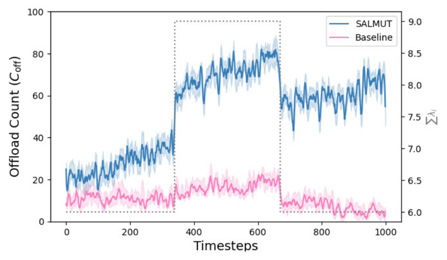
Appendix A
Proof of Theorem 1

The equivalence between the continuous and discrete time MDPs follows from the uniformization technique [15]–[17]. Since the transition rates depend on the current state, we construct a uniform version of the process by allowing fictitious transitions from a state to itself. Specifically, in the transition probability $\bar{p}(x', \ell' | x, \ell, a)$, we allow allow transitions at a faster rate of $\lambda + k\mu$, but with probability $1 - \frac{\lambda + \min\{x,k\}\mu}{\nu}$ transitions do not lead to a change in state. Since the transition rates of $\bar{p}$ does not depend on state, it is called the uniform version of p. It can be shown that for any policy π, and any initial state (x_0, ℓ_0), time t, and state (x, ℓ), the probability $P^\pi(X_t = x, L_t = \ell \mid X_0 = x_0, L_0 = \ell_0)$ is identical for the original process and its uniform version.

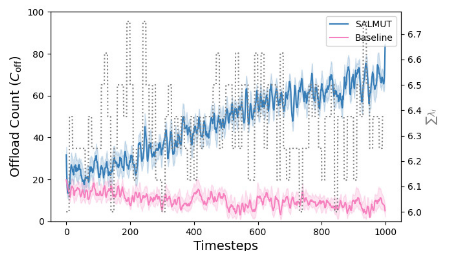
Now let T_n denote the time when the state (X_t, L_t) jumps



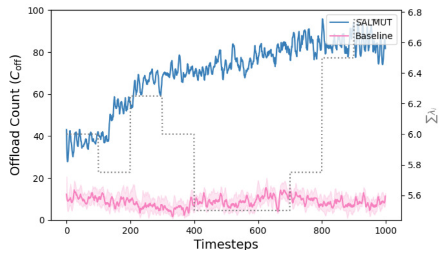
(a) Scenario 1



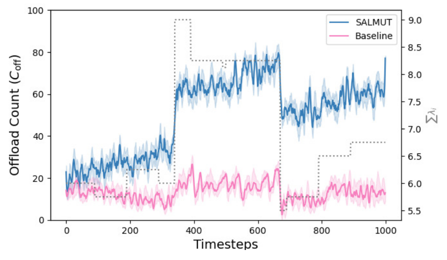
(b) Scenario 2



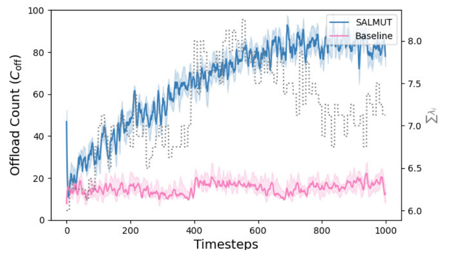
(c) Scenario 3



(d) Scenario 4

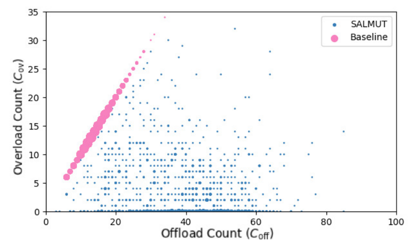


(e) Scenario 5

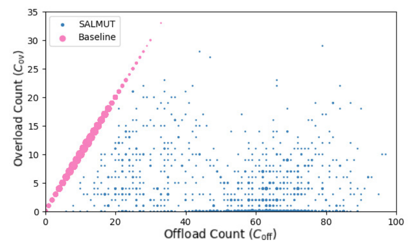


(f) Scenario 6

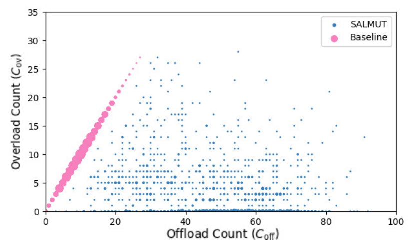
Fig. 12. Comparing the number of times the system performs offloading at each step in the end-to-end testbed we created. We also plot the total request arrival rate ($\sum_i \lambda_i$) on the right-hand side of y-axis in gray dotted lines.



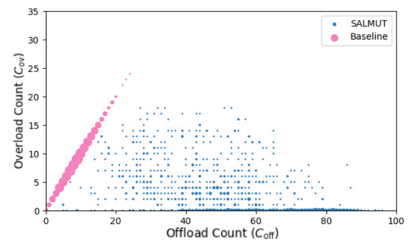
(a) Scenario 1



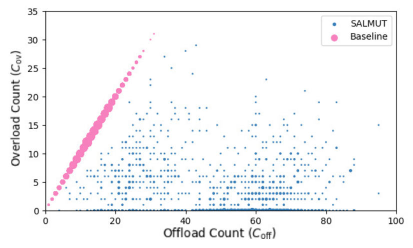
(b) Scenario 2



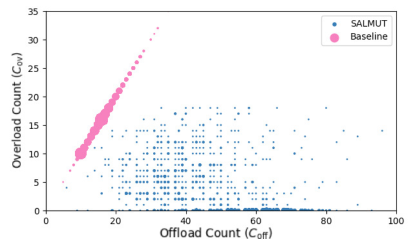
(c) Scenario 3



(d) Scenario 4



(e) Scenario 5



(f) Scenario 6

Fig. 13. Scatter-plot of C_{ov} Vs C_{off} for SALMUT and baseline algorithm in the end-to-end testbed we created. The width of the points is proportional to its frequency.

for the n-th time. Note that (X_t, L_t) is constant in the interval $[T_n, T_{n+1})$ and so is $A_t = \pi(X_t, L_t)$ for any time homogeneous Markov policy π. Thus,

$$V^\pi(x,\ell) = \mathbb{E}\Big[\int_0^\infty e^{-\alpha t}\rho(X_t, L_t, A_t)dt | X_0 = x, L_0 = \ell\Big]$$

$$\begin{aligned}
&= \sum_{n=0}^{\infty} \mathbb{E}^\pi\Big[\int_{T_n}^{T_{n+1}} e^{-\alpha t}\rho(X_t, L_t, A_t)dt | X_0 = x, L_0 = \ell\Big] \\
&\overset{(a)}{=} \sum_{n=0}^{\infty} \mathbb{E}^\pi\Big[\int_{T_n}^{T_{n+1}} e^{-\alpha t} dt\Big] \mathbb{E}\big[\rho(X_{T_n}, L_{T_n}, A_{T_n}) \big| X_0 = x, L_0 = \ell\big] \\
&\overset{(b)}{=} \sum_{n=0}^{\infty} \frac{\beta^n(1-\beta)}{(\alpha+\lambda+k\mu)} \mathbb{E}\big[\rho(X_{T_n}, L_{T_n}, A_{T_n}) \big| X_0 = x, L_0 = \ell\big] \\
&= \frac{1}{\alpha+\lambda+k\mu}\mathbb{E}\Big[\sum_{n=0}^{\infty} \beta^n(1-\beta)\rho(X_{T_n}, L_{T_n}, A_{T_n})\Big] \qquad (17)
\end{aligned}$$

where (a) uses the fact that in the uniform version of the process dwell time is independent of the state (b) uses the fact that the dwell time $\tau_n = T_{n+1} - T_n$ has an Poisson distribution with rate $(\lambda + k\mu)$. Note that Eq. (17) is a discrete time controlled Markov process. The optimality of the policy π is obtained in Theorem 1 follows from the standard results for MDP [18].

Appendix B
Proof of Proposition 1

Let $\delta(x) = \min(k, x)\mu$. We define a sequence of value functions $\{V_n\}_{n\geq 0}$ as follows: $V_0(x, \ell) = 0$ and for $n \geq 0$

$$V_{n+1}(x, \ell) = \min\{Q_{n+1}(x, \ell, 0), Q_{n+1}(x, \ell, 1)\},$$

where

$$\begin{aligned}Q_{n+1}(x,\ell,0) = {} & \frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell)\big] \\ & + \beta\Bigg[\frac{\lambda}{\nu}\sum_{r=1}^{\mathsf{R}} P(r)V_n([x+1]_{\mathsf{X}}, [\ell+r]_{\mathsf{L}}) \\ & + \frac{\delta(x)}{\nu}\sum_{r=1}^{\mathsf{R}} P(r)V_n([x-1]^+, [\ell-r]^+) \\ & + \left[1 - \frac{\lambda+\delta(x)}{\nu}\right]V_n(x,\ell)\Bigg]\end{aligned}$$

and

$$\begin{aligned}Q_{n+1}(x,\ell,1) = {} & \frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell) + p(\ell)\big] \\ & + \beta\Bigg[\frac{\delta(x)}{\nu}\sum_{r=1}^{\mathsf{R}} P(r)V_n([x-1]^+, [\ell-r]^+) \\ & + \left[1 - \frac{\lambda+\delta(x)}{\nu}\right]V_n(x,\ell)\Bigg]\end{aligned}$$

where $[x]_{\mathsf{B}}$ denotes $\min\{x, \mathsf{B}\}$.

Note that $\{V_n\}_{n\geq 0}$ denotes the iterates of the value iteration algorithm, and from [18], we know that

$$\lim_{n\to\infty} V_n(x, \ell) = V(x, \ell), \quad \forall x, \ell \tag{18}$$

where V is the unique fixed point of (7).

We will show that (see Lemma 1 below) each $V_n(x, \ell)$ satisfies the property of Proposition 1. Therefore, by (18) we get that V also satisfies the property.

Lemma 1 *For each $n \geq 0$ and $x \in \{0, ..., X\}$, $V_n(x, \ell)$ is weakly increasing in ℓ.*

Proof We prove the result by induction. Note that $V_0(x, \ell) = 0$ and is trivially weakly increasing in ℓ. This forms the basis of the induction. Now assume that $V_n(x, \ell)$ is weakly increasing in ℓ. Consider iteration $n + 1$. Let $x \in \{0, ..., X\}$ and $\ell_1, \ell_2 \in \{0, ..., L\}$ such that $\ell_1 < \ell_2$. Then,

$$\begin{aligned}Q_{n+1}(x,\ell_1,0) = {} & \frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell_1)\big] \\ & + \beta\Bigg[\frac{\lambda}{\nu}\sum_{r=1}^{\mathsf{R}} P(r)V_n([x+1]_{\mathsf{X}}, [\ell_1+r]_{\mathsf{L}}) \\ & + \frac{\delta(x)}{\nu}\sum_{r=1}^{\mathsf{R}} P(r)V_n([x-1]^+, [\ell_1-r]^+) \\ & + \left[1 - \frac{\lambda+\delta(x)}{\nu}\right]V_n(x,\ell)\Bigg] \\ \overset{(a)}{\leq} {} & \frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell_2)\big] \\ & + \beta\Bigg[\frac{\lambda}{\nu}\sum_{r=1}^{\mathsf{R}} P(r)V_n([x+1]_{\mathsf{X}}, [\ell_2+r]_{\mathsf{L}}) \\ & + \frac{\delta(x)}{\nu}\sum_{r=1}^{\mathsf{R}} P(r)V_n([x-1]^+, [\ell_2-r]^+) \\ & + \left[1 - \frac{\lambda+\delta(x)}{\nu}\right]V_n(x,\ell)\Bigg] \\ = {} & Q_{n+1}(x, \ell_2, 0),\end{aligned} \tag{19}$$

where (a) follows from the fact that $c(\ell)$ and $V_n(x, \ell)$ are weakly increasing in ℓ.

By a similar argument, we can show that

$$Q_{n+1}(x, \ell_1, 1) \leq Q_{n+1}(x, \ell_2, 1). \tag{20}$$

Now,

$$\begin{aligned}V_{n+1}(x, \ell_1) &= \min\{Q_{n+1}(x, \ell_1, 0), Q_{n+1}(x, \ell_1, 1)\} \\ &\overset{(b)}{\leq} \min\{Q_{n+1}(x, \ell_2, 0), Q_{n+1}(x, \ell_2, 1)\} \\ &= V_{n+1}(x, \ell_2),\end{aligned} \tag{21}$$

■

where (b) follows from (19) and (20). Eq. (21) shows that $V_{n+1}(x, \ell)$ is weakly increasing in ℓ. This proves the induction step. Hence, the result holds for the induction.

Appendix C
Proof of Proposition 2

Proof Let $\delta(x) = \min\{x, k\}\mu$. Consider

$$\begin{aligned}\Delta Q(x, \ell) &= := Q(x, \ell, 1) - Q(x, \ell, 0) \\ &= -\beta\frac{\delta(x)}{\nu}\sum_{r=1}^{\mathsf{R}} P(r)V([x]_{\mathsf{X}}, [\ell+r]_{\mathsf{L}}) - p.\end{aligned}$$

For a fixed x, by Proposition 1, $\Delta Q(x, \ell)$ is weakly decreasing in ℓ. If it is optimal to reject a request at state (x, ℓ) (i.e., $\Delta Q(x, \ell) \leq 0$), then for any $\ell' > \ell$,

$$\Delta Q(x, \ell') \leq \Delta Q(x, \ell) \leq 0;$$

■

therefore, it is optimal to reject the request.

Appendix D
Proof of Optimality of SALMUT

Proof The choice of learning rates implies that there is a separation of timescales between the updates of (15) and (16). In particular, since $b_n^2/b_n^1 \to 0$, iteration (15) evolves at a faster timescale than iteration (16). Therefore, we first consider update (16) under the assumption that the policy π_τ, which updates at the slower timescale, is constant.
We first provide a preliminary result.

Lemma 2 *Let Q_τ denote the action-value function corresponding to the policy π_τ. Then, Q_τ is Lipscitz continuous in τ.*

Proof This follows immediately from the Lipscitz continuity of π_τ in τ. ■

Define the operator $\mathcal{M}_\tau : \mathbb{R}^{\mathsf{N}} \to \mathbb{R}^{\mathsf{N}}$, where $\mathsf{N} = (\mathsf{X}+1)\times(\mathsf{L}+1)\times\mathcal{A}$, as follows:

$$[\mathcal{M}_\tau Q](x,\ell,a) = \big[\bar{\rho}(x,\ell,a) + \beta \sum_{x',\ell'} p(x',\ell'|x,\ell,a) \min_{a'\in\mathcal{A}} Q(x',\ell',a')\big] - Q(x,\ell,a). \quad (22)$$

Then, the step-size conditions on $\{b_n^1\}_{n\geq 1}$ imply that for a fixed π_τ, iteration (15) may be viewed as a noisy discretization of the ODE (ordinary differential equation):

$$\dot{Q}(t) = \mathcal{M}_\tau[Q(t)]. \quad (23)$$

Then we have the following:

Lemma 3 *The ODE* (23) *has a unique globally asymptotically stable equilibrium point Q_τ.*

Proof Note that the ODE (23) may be written as

$$\dot{Q}(t) = \mathcal{B}_\tau[Q(t)] - Q(t)$$

where the Bellman operator $\mathcal{B}_\tau : \mathbb{R}^{\mathsf{N}} \to \mathbb{R}^{\mathsf{N}}$ is given by

$$\mathcal{B}_\tau[Q](x,\ell) = \big[\bar{\rho}(x,\ell,a) + \beta \sum_{x',\ell'} p(x',\ell'|x,\ell,a) \times \min_{a'\in\mathcal{A}} Q(x',\ell',a')\big]. \quad (24)$$

Note that $\mathcal{B}_\tau$ is a contraction under the sup-norm. Therefore, by Banach fixed point theorem, $Q = \mathcal{B}_\tau Q$ has a unique fixed point, which is equal to Q_τ. The result then follows from [26, Theorem 3.1]. ■

We now consider the faster timescale. Recall that (x_0, ℓ_0) is the initial state of the MDP. Recall

$$J(\tau) = V_\tau(x_0, \ell_0)$$

and consider the ODE limit of the slower timescale iteration (16), which is given by

$$\dot{\tau} = -\nabla J(\tau). \quad (25)$$

Lemma 4 *The equilibrium points of the ODE* (25) *are the same as the local optima of $J(\tau)$. Moreover, these equilibrium points are locally asymptotically stable.*

Proof The equivalence between the stationary points of the ODE and local optima of $J(\tau)$ follows from definition. Now consider $J(\tau(t))$ as a Lyapunov function. Observe that

$$\frac{d}{dt}J(\tau(t)) = -\big[\nabla J(\tau(t))\big]^2 < 0,$$

as long as $\nabla J(\tau(t)) \neq 0$. Thus, from Lyapunov stability criteria all local optima of (25) are locally asymptotically stable. ■

Now, we have all the ingredients to prove convergence. Lemmas 2-4 imply assumptions (A1) and (A2) of [27]. Thus, the iteration (15) and (16) converges almost surely to a limit point (Q°, τ°) such that $Q^\circ = Q_{\tau^\circ}$ and $\nabla J(\tau^\circ) = 0$ provided that the iterates $\{Q_n\}_{n\geq 1}$ and $\{\tau_n\}_{n\geq 1}$ are bounded.

Note that $\{\tau_n\}_{n\geq 1}$ are bounded by construction. The boundness of $\{Q_n\}_{n\geq 1}$ follows from considering the scaled version of (23):

$$\dot{Q} = \mathcal{M}_{\tau,\infty} Q \quad (26)$$

where,

$$\mathcal{M}_{\tau,\infty} Q = \lim_{c\to\infty} \frac{\mathcal{M}_\tau[cQ]}{c}.$$

It is easy to see that

$$[\mathcal{M}_{\tau,\infty} Q](x,\ell,a) = \beta \sum_{x',\ell'} p(x',\ell'|x,\ell,a) \min_{a'\in\mathcal{A}} Q(x',\ell',a') - Q(x,\ell,a) \quad (27)$$

Furthermore, origin is the asymptotically stable equilibrium point of (26). Thus, from [28], we get that the iterates $\{Q_n\}_{n\geq 1}$ of (15) are bounded. ■

Acknowledgment

The numerical experiments were enabled in part by support provided by Compute Canada. The authors are grateful to Pierre Thibault from Ericsson Systems for setting up the virtual machines to run the docker-testbed experiments. The authors also acknowledge all the help and support from Ericsson Systems, especially the Global Aritifical Intelligent Accelerator (GAIA) Montreal Team.

References

[1] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[2] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[3] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief, "Delay-optimal computation task scheduling for mobile-edge computing systems," in *Int. Symp. Inform. Theory (ISIT)*, 2016, pp. 1451–1455.

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

[5] D. Van Le and C.-K. Tham, "Quality of service aware computation offloading in an ad-hoc mobile cloud," *IEEE Trans. Veh. Technol.*, vol. 67, no. 9, pp. 8890–8904, 2018.

[6] M. Chen and Y. Hao, "Task offloading for mobile edge computing in software defined ultra-dense network," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 3, pp. 587–597, 2018.

[7] 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 Trans. Netw.*, vol. 27, no. 3, pp. 1272–1288, Jun. 2019.

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

[9] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning," *IEEE Internet Things J*, vol. 6, no. 3, pp. 4005–4018, 2018.

[10] L. Huang, S. Bi, and Y.-J. A. Zhang, "Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks," *IEEE Transactions on Mobile Computing*, vol. 19, no. 11, pp. 2581–2593, 2019.

[11] J. Wang, J. Hu, G. Min, W. Zhan, Q. Ni, and N. Georgalas, "Computation offloading in multi-access edge computing using a deep sequential model based on reinforcement learning," *IEEE Commun. Mag.*, vol. 57, no. 5, pp. 64–69, 2019.

[12] J. Li, H. Gao, T. Lv, and Y. Lu, "Deep reinforcement learning based computation offloading and resource allocation for mec," in *2018 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2018, pp. 1–6.

[13] D. Van Le and C.-K. Tham, "Quality of service aware computation offloading in an ad-hoc mobile cloud," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 9, pp. 8890–8904, 2018.

[14] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Transactions on Services Computing*, vol. 12, no. 5, pp. 712–725, 2018.

[15] A. Jensen, “Markoff chains as an aid in the study of Markoff processes,” *Scandinavian Actuarial Journal*, vol. 1953, no. sup1, pp. 87–91, 1953.
[16] R. A. Howard, *Dynamic Programming and Markov Processes*. The MIT Press, 1960.
[17] R. F. Serfozo, “An equivalence between continuous and discrete time markov decision processes,” *Operations Research*, vol. 27, no. 3, pp. 616–620, 1979.
[18] M. Puterman, *Markov decision processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.
[19] A. Roy, V. Borkar, A. Karandikar, and P. Chaporkar, “Online reinforcement learning of optimal threshold policies for Markov decision processes,” *arXiv:1912.10325*, 2019.
[20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv:1707.06347*, 2017.
[21] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba, “Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation,” *arxiv:1708.05144*, 2017.
[22] A. Jitani, A. Mahajan, Z. Zhu, H. Abou-zeid, E. T. Fapi, and H. Purmehdi, “Structure-aware reinforcement learning for node overload protection in mobile edge computing,” in *IEEE International Conference on Communications (ICC)*, 2021.
[23] X. Cao, *Stochastic Learning and Optimization - A Sensitivity-Based Approach*. Springer, 2007. [Online]. Available: https://doi.org/10.1007/978-0-387-69082-7
[24] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.
[25] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, “Stable baselines3,” https://github.com/DLR-RM/stable-baselines3, 2019.
[26] V. S. Borkar and K. Soumyanatha, “An analog scheme for fixed point computation – Part I: Theory,” *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 44, no. 4, pp. 351–355, 1997.
[27] V. S. Borkar, “Stochastic approximation with two time scales,” *Systems & Control Letters*, vol. 29, no. 5, pp. 291–294, 1997.
[28] V. S. Borkar and S. P. Meyn, “The ODE method for convergence of stochastic approximation and reinforcement learning,” *SIAM Journal on Control and Optimization*, vol. 38, no. 2, pp. 447–469, 2000.

Anirudha Jitani received the B.Tech degree in Computer Science and Engineering from Vellore Institute of Technology, India in 2015. He is pursuing his M.Sc. in Computer Science at McGill University, Montreal since 2018. He is a research assistant at Montreal Institute of Learning Algorithms (MILA) and a Data Scientist Intern at Ericsson Systems. He also worked as a software developer for Cisco Systems and Netapp. His research interests include application of machine learning techniques such as deep reinforcement learning, multi-agent reinforcement learning, and graph neural networks in communication and networks.

Aditya Mahajan (S’06-M’09-SM’14) received B.Tech degree from the Indian Institute of Technology, Kanpur, India, in 2003, and M.S. and Ph.D. degrees from the University of Michigan, Ann Arbor, USA, in 2006 and 2008. From 2008 to 2010, he was a Postdoctoral Researcher at Yale University, New Haven, CT, USA. He has been with the department of Electrical and Computer Engineering, McGill University, Montreal, Canada, since 2010 where he is currently Associate Professor. He serves as Associate Editor of IEEE Transactions on Automatic Control and Springer Mathematics of Control, Signal, and Systems. He is the recipient of the 2015 George Axelby Outstanding Paper Award, 2014 CDC Best Student Paper Award (as supervisor), and the 2016 NecSys Best Student Paper Award (as supervisor). His principal research interests include decentralized stochastic control, team theory, multi-armed bandits, and reinforcement learning.

Zhongwen Zhu received the B.Eng in Mechanical Engineering and M.Eng. in Turbomachinery from Shanghai Jiao Tong University, Shanghai, P.R. China, and also a Ph.D. in Applied Science from Free University of Brussels, Belgium, in 1996. He joined in Aerospace Lab in National Research Council of Canada (NRC) in 1997. One year later, he started to work in Bombardier Aerospace. Since 2001, he has been working for Ericsson Canada with different roles, e.g. Data Scientist, SW Designer, System manager, System designer, System/Solution Architect, Product owner, Product manager, etc. He was an associated editor for Journal of Security and Communication Network (Wiley publisher). He was the invited Technical committee member for International Conference on Multimedia Information Networking and Security. He was the recipient of the best paper award in 2008 IEEE international conference on Signal Processing and Multimedia applications. His current research interests are 5G network, network security, edge computing, Reinforcement learning, Machine learning for 2D/3D object detections, IoT (Ultra Reliable Low Latency application).

Hatem Abou-Zeid is an Assistant Professor in the Department of Electrical and Software Engineering at the University of Calgary. Prior to joining the University of Calgary, he was a 5G Systems Developer at Ericsson Canada. He has 10+ years of R&D experience in communication networks spanning radio access networks, routing, and traffic engineering. He currently leads 5G system designs and intellectual property development in the areas of network intelligence and low latency communications. He serves on the Ericsson Government Industry Relations and University Engagements Committee where he directs academic research partnerships on autonomous networking and augmented reality communications. His research investigates the use of reinforcement learning, stochastic optimization, and deep learning to architect robust 5G/6G networks and applications - and his work has resulted in 60+ filed patents and publications in IEEE flagship journals and conferences. Prior to joining Ericsson, he was at Cisco Systems designing scalable traffic engineering and IP routing protocols for service provider and data-center networks. He holds a Ph.D. in Electrical and Computer Engineering from Queen’s University. His collaborations with industry through a Bell Labs DAAD RISE Fellowship led to the commercialization of aspects of predictive video streaming, and his Thesis was nominated for an Outstanding Thesis Medal.

Emmanuel Thepie Fapi is currently a Data Scientist with Ericsson Montreal, Canada. He holds a master’s degree in engineering mathematics and computer tools from Orleans University in France and a PhD in signal processing and telecommunications from IMT Atlantique in France (former Ecole des Telecommunications de Bretagne) in 2009. From 2010 to 2016 he worked with GENBAND US LLC, QNX software System Limited as audio software developer, MDA system as analyst and EasyG as senior DSP engineer in Vancouver, Canada. In 2017 he joined Amazon Lab 126 in Boston, USA as audio software developer for echo dot 3rd generation. His main areas of interest are 5G network, anomaly detection-based AI/ML, multi-resolution analysis for advanced signal processing, real-time embedded OS and IoT, voice and audio quality enhancement.

Hakimeh Purmehdi is a data scientist at Ericsson Global Artificial Intelligence Accelerator, which leads innovative AI/ML solutions for future wireless communication networks. She received her Ph.D. degree in electrical engineering from the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada. After completing a postdoc in AI and image processing at the Radiology Department, University of Alberta, she co-founded Corowave, a startup to develop bio sensors to monitor human vital signals by leveraging radio frequency technology and machine learning. Before joining Ericsson, she was with Microsoft Research (MSR) as a research engineer, and contributed in the development of TextWorld, which is a testbed for reinforcement learning research projects. Her research focus is basically on the intersection of wireless communication (5G and beyond including resource management and edge computing), AI solutions (such as online learning, federated learning, reinforcement learning, deep learning), optimization, and biotech.