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

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

***Abstract*—Mobile Edge Computing (MEC) refers to the concept of placing computational capability and applications at the edge of the network. Running applications and computations closer to the clients provide benefits such as reduced latency in handling client requests, reduced network congestion, and improved performance of applications. Edge server performance and reliability is degraded when it is overloaded, especially if it crashes due to overload and causes service failures. In this work, an adaptive admission control policy to prevent edge node from getting overloaded is presented. This approach is based on a low complexity RL (Reinforcement Learning) called SALMUT (Structure-Aware Learning for Multiple Thresholds). 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 SALMUT performs as well as the state-of-the-art deep RL algorithms such as PPO (Proximal Policy Optimization) and A2C (Advantage Actor Critic) in the computer simulations. Our empirical evaluations also show that SALMUT performs better than the baseline algorithm in the docker-testbed. The main advantage of our proposal is that it requires an order of magnitude less time to train compared to PPO and A2C, outputs easily interpretable policy, and can be deployed in an online manner.**

***Index Terms*—Reinforcement learning, structure-aware reinforcement learning, markov decision process, mobile edge computing, node-overload protection.**

I. Introduction

In the last decade, we have seen a shift in the computing paradigm from co-located datacenters and compute serves 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 focus on the quality of experience of the end users has become a very important aspect for network providers, specially with the proliferation of Video-on-Demand (VoD) services, Internet-of-Things (IoT), real-time online gaming platforms, and Virtual Reality (VR) applications. 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 a 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 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, 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 a lot of 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 level have not been investigated as extensively. When an edge server receives a large number of requests in a short period of time (for example be 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, 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

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 Ericsson, Ottawa, Canada.
Emails: zhongwen.zhu@ericsson.com, emmanuel.thepie.fapi@ericsson.com, hakimeh.purmehdi@ericsson.com, hatem.abou-zeid@ericsson.com.
This work was supported in part by MITACS Accelerate, Grant IT16364.

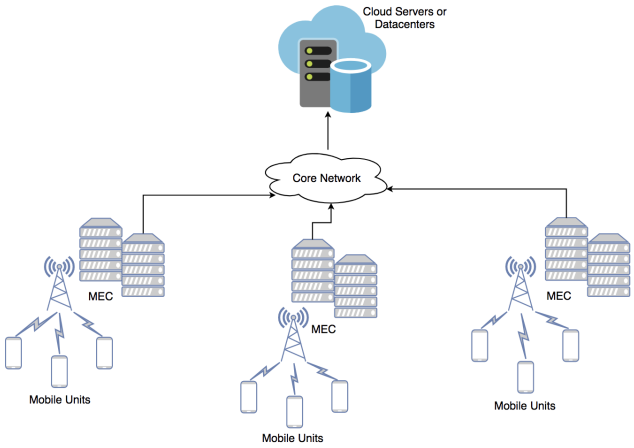


Fig. 1. A Mobile Edge Computing (MEC) system. User may be mobile and will connect to the closest edge server. The MEC servers are connected to the backend cloud server or datacenters through the core network.

takes extra time and reduces the resources available for other services provided by the cluster. Therefore, it is paramount to design *pro-active* 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 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]–[16] 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. We first model the problem to incorporate practical considerations of server holding, processing and offloading costs. Then we develop an offloading policy offloads new requests to balance the overall running cost of the system. In the simplest case, when the request arrival process

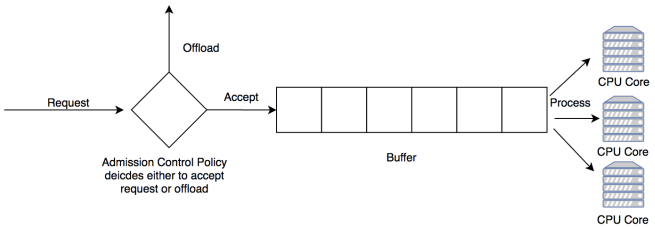


Fig. 2. System model of admission control in a single edge server.

is time-homogeneous, we model the system as a continuous-time MDP and use the *uniformization technique* [17], [18] to convert the continuous-time MDP to a discrete-time MDP, which can then be solved using standard dynamic programming algorithms [19].

The main contributions of the paper are as follows:

- We design a node-overload protection scheme that uses a recently proposed low-complexity RL algorithm called Structure-Aware Learning for Multiple Thresholds (SALMUT) [20]. SALMUT exploits the structure of the optimal policy, requires considerably fewer computational resources to train and provides policies which are easy to interpret.
- 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 Promixal Policy Optimization (PPO) [21] and Advantage Actor Critic (A2C) [22], but requires an order of magnitude less time to train and provides policies which are easy to interpret.
- 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.

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 [20] 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, in Sec. VII, we provide the conclusion, limitations of our model, and future directions.

II. Model and Problem Formulation

A. System model

A simplified MEC system consists of an edge server and several mobile users accessing that server (see Fig. 1). Mobile users independently generate service requests according to a Poisson process. The rate of requests and the number of

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 factor in continuous MDP
$V^\pi(x,\ell)$	Performance of the policy π
$p(x',\ell'\|x,\ell,a)$	Transition probability function
β	Discount factor 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 SALMULT policy π_τ
$f(\tau(x),\ell)$	Probability of accepting new request
T	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

users may also change with time. The edge server takes CPU resources to serve each request from mobile users. The request is buffered in a queue before it is served. When a new request comes, the server has the option to offload the request. 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 for 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:** 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 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:** Each 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 of users is not fixed and changes over time. We call them Scenario 4, 5, and 6 respectively.

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.
- a penalty of $p(\ell)$ for offloading a packet at CPU load ℓ.

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 increasing in ℓ.

Whenever a new request arrives, the server uses a memoryless 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 setting 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 $\{X_t, L_t\}_{t\geq 0}$ is a finite-state continuous-time MDP controlled through $\{A_t\}_{t\geq 0}$. 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

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

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,

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

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. In general, the reduction in CPU utilization will correspond to the resources requested by the request whose service was completed. However, keeping track of those resources would mean that we would need to expand the state and include $(R_1, \ldots, R_k)$ as part of the state, where R_i denotes the resources required by the request which is being processed by CPU i. In order to avoid such an increase in state dimension, we assume that when a request is completed, CPU utilization reduces by amount $\ell - \ell'$ with probability $P(\ell - \ell')$.

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

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

Let $\nu = \lambda + k\mu$ denote the uniform upper bound on the transition rate at the states. Then, using the *uniformization technique* [17], [18], 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 $p(x', \ell' | x, \ell, a)$ 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*

$$V(x, \ell) = \min\{Q(x, \ell, 0), Q(x, \ell, 1)\} \quad (6)$$

where

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

and

$$Q(x,\ell,1) = \frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell) + p(\ell)\big] + \beta \frac{\min\{x,k\}\mu}{\lambda + \min\{x,k\}\mu} \sum_{r=1}^{\mathsf{R}} P(r) V([x-1]^+, [\ell - r]^+)$$

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

Let $\pi(x, \ell) \in \mathcal{A}$ *denote the argmin the right hand side of* (6)*. Then, the time-homogeneous policy* $\pi(x, \ell)$ *is optimal for the original continuous-time optimization problem.*

Proof The equivalence between the continuous and discrete time MDPs follows from the uniformization technique [17], [18]. The optimality of the time-homogeneous policy π follows from the standard results for MDPs [19]. ∎

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, we will ignore the $1/(\alpha+\nu)$ term in front of the per-step cost and assume that it has been absorbed in the constant h, and the functions $c(\cdot)$, $p(\cdot)$.

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 A. ∎

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 B. ∎

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 [20] to exploit a similar structure in admission control for multi-class queues. It was proposed for the average cost setting but, as explained below, it generalizes to the discounted cost setting as well.

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

$$f(\tau(x), \ell) = \frac{\exp((\tau(x) - \ell)/T)}{1 + \exp((\tau(x) - \ell)/T)}, \tag{7}$$

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

Fix an initial state (x_0, ℓ_0) and let $J(\tau)$ denote the performance of policy π_τ when starting from the initial state (x_0, ℓ_0). Let V_τ and Q_τ denote the value function and action-value function of policy π_τ.

Now, from the policy gradient theorem [8], we know that

$$\nabla J(\tau) \quad = \quad \sum_{x=0}^{\mathsf{X}} \sum_{\ell=0}^{\mathsf{L}} \mu(x, \ell) \sum_{a \in \mathcal{A}} \frac{\delta \pi_\tau(a|x, \ell)}{\delta \tau} Q_\tau(x, \ell, a) \tag{8}$$

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

$$\nabla Q(x, \ell; \tau) = \sum_{x'=0}^{\mathsf{X}} \sum_{\ell'=0}^{\mathsf{L}} \nabla p^{(\tau)}(x', \ell'|x, \ell) \times \sum_{a \in \mathcal{A}} \pi_\tau(a|x, \ell) Q_\tau(x, \ell, a).$$

Therefore, an unbiased estimator of $\nabla J(\tau)$ is given by

$$\nabla p^{(\tau)}(x', \ell'|x, \ell) Q_\tau(x, \ell, a), \quad \text{where } a \sim \pi_\tau(\cdot|x, \ell). \tag{9}$$

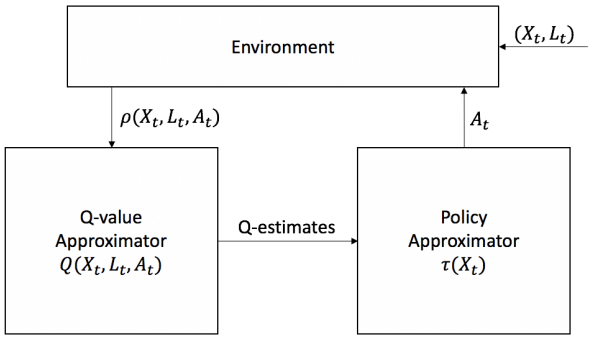


Fig. 3. Illustration of the two-timescale SALMUT algorithm.

Note that

$$\frac{\delta \pi_\tau(a|x, \ell)}{\delta \tau} = (-1)^a \nabla f(\tau(x), \ell). \tag{10}$$

Combining (8) with (10), we get that

$$(-1)^a \nabla f(\tau(x), \ell) \big[\bar{\rho}(x, \ell, a) + \beta V_\tau(x', \ell')\big], \tag{11}$$

where $a \sim \pi_\tau(\cdot|x, \ell)$ is an unbiased estimator of $\nabla J(\tau)$.

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 value function and the optimal policy. Then, we update the action-value function using temporal difference learning:

$$Q_{n+1}(x, \ell, a) = Q_n(x, \ell, a) + b_n^1 \big[\bar{\rho}(x, \ell, a) + \beta \min_{a' \in A} Q_n(x', \ell', a') - Q_n(x, \ell, a)\big], \tag{12}$$

and update the policy parameters using stochastic gradient descent while using (11) as the unbiased estimator of $\nabla J(\tau)$:

$$\tau_{n+1}(x) = \text{Proj}\big[\tau_n(x) + b_n^2 (-1)^a \nabla f(\tau(x), \ell) \big[\bar{\rho}(x, \ell, a) + \beta \min_{a' \in \mathcal{A}} Q(x', \ell', a')\big]\big], \tag{13}$$

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: τ
Initialize value function ∀x, ∀ℓ, Q(x, ℓ, a) ← 0
Initialize threshold vector ∀x, τ(x) ← rand(0, L)
Initialize start state (x, ℓ) ← (x_0, ℓ_0)
while TRUE do
    if EVENT == ARRIVAL then
        Choose action a according to Eq. (7)
        Update Q(x, ℓ, a) according to Eq. (12)
        Update threshold τ using Eq. (13)
        (x, ℓ) ← (x', ℓ')
    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 C. ∎

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 fixed cost for offloading to enable the offloading behavior only when the system is loaded and a very high cost when load is 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

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

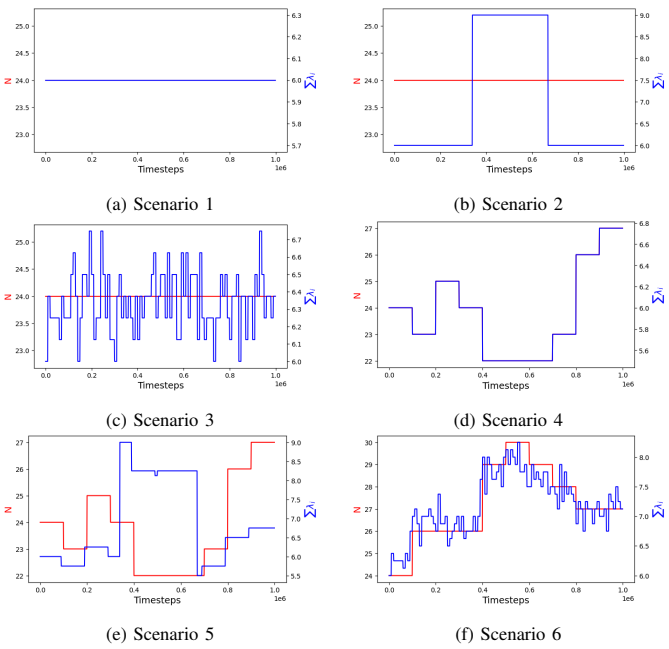
Rather than simulating the system in continuous-time, we simulate the equivalent discrete-time MDP by generating the next event (arrival or departure) using a Bernoulli distribution with probabilities and costs described in Sec. III. We assume that the parameter $1/(\alpha + \nu)$ in (6) has been absorbed in the cost function. We assume that the discrete time discount factor $\beta = \alpha/(\alpha + \nu)$ equals 0.95.

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 horizon of $T = 10^6$. The scenarios capture variation in the transmission rate and the number of users over time, their realization can be seen in Fig. 4.

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 $\lambda = N\lambda_i = 6$.



(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.

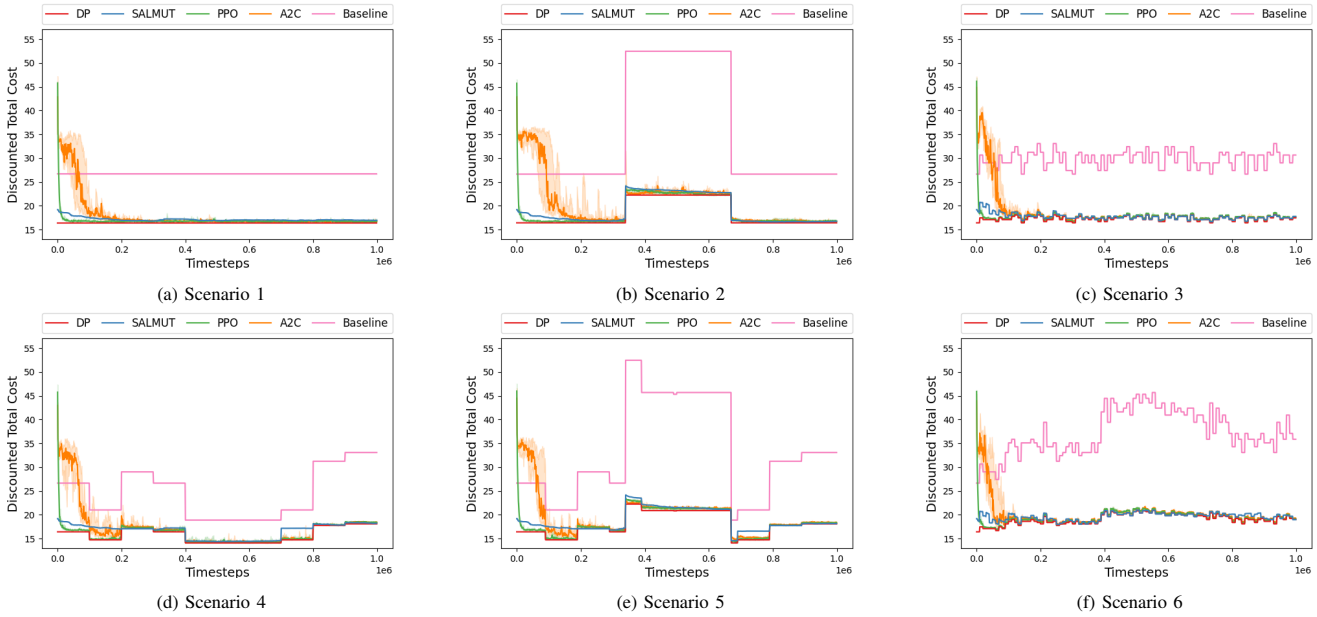
b) Scenario 2: This scenario tests how the learning algorithms adapt to occasional but significant changes to arrival rates. We consider a system with $N = 24$ users, where each user generates requests at rate $\lambda_{\text{low}} = 0.25$ for the interval $(0, 3.33 \cdot 10^5]$, then generates requests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

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 $m \cdot 10^4$, 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.



(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.

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) PPO [21], which is a family of trust region policy gradient method and optimizes a surrogate objective function using stochastic gradient ascent.
4) A2C [22], 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.
5) Baseline, which is a fixed-threshold based policy, where the node accepts requests when $\ell < 18$ (non-overloaded state) and offloads requests otherwise. Such static policies are currently deployed in many real-world systems.

For SALMUT, we use ADAM [23] optimizer with initial learning rates ($b^1 = 0.002, b^2 = 0.03$). We used the stable-baselines [24] implementation of PPO and A2C with learning rates 0.0003 and 0.001 respectively.

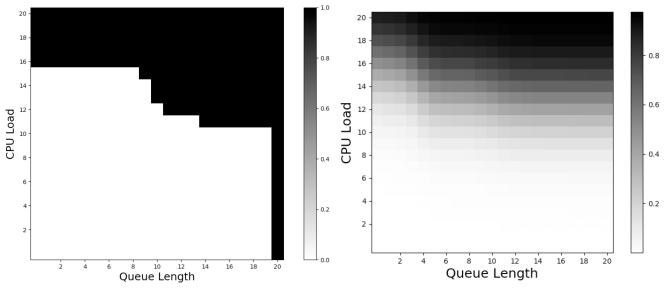
C. Results

For each of the algorithm described above, we train SALMUT, PPO, and A2C for 10^6 steps. The performance of each algorithm is evaluated every 10^3 steps using independent rollouts of length $H = 1000$ for 100 different random seeds. The experiment is repeated for the 10 sample paths and the median performance with an uncertainty band from the first to the third quartile are plotted in Fig. 5.

For Scenario 1, all RL algorithms (SALMUT, PPO, A2C) converge to a close-to-optimal policy relatively quickly (with A2C being slightly slower) and remain stable after convergence. Since all policies converge quickly, they 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 peformance of the RL algorithms, but these are minor. Note that, in contrast, the baseline policy of offloading when the server is overloaded performs poorly.

The plots for Scenario 1 (Fig. 5a) show that PPO converges to the optimal policy in less than $3 \cdot 10^4$ steps, SALMUT takes around $1 \cdot 10^5$ steps, whereas A2C takes a little longer to convege (around $2 \cdot 10^5$ steps). 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 high and CPU load (ℓ) is low, 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 when λ is constant. When λ changes significantly, we observe that SALMUT is able to adapt to the drastic but stable changes in the environment slightly better than the Deep RL algorithms (PPO, A2C). 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 SALMUT show similar performance for most of the time-steps except in cases when the load decreases, in which case SALMUT performs slightly worse than PPO and A2C. This could be due to the fact that SALMUT takes longer to adjust its more aggresive offloading policy when it was in a more



(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.

loaded state.

D. Analysis of Training Time and Policy Interpretability

The main difference among these three 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. 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.

TABLE II
TRAINING TIME OF RL ALGORITHMS

Algorithm	Mean Time (s)	Std-dev (s)
SALMUT	95.67	3.29
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.

E. Behavioral Analysis of Policies

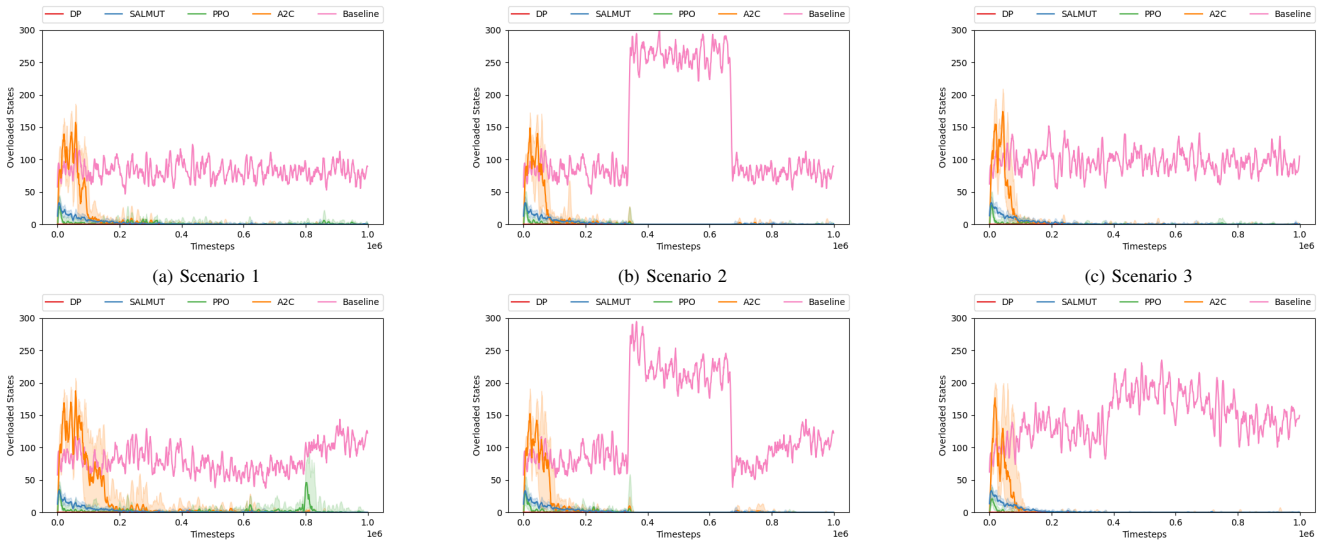
We performed further analysis on the behavior of the learned policy by observing the number of times the system enters into an overloaded state and offloads incoming request. Let us define C_{ov} to be the number of times the system enters into an overloaded state and C_{off} to be the number of times the system offloads requests for every 1000 steps of training iteration. We generated a set of 10^6 random numbers between 0 and 1, defined by z_t, where t is the step count. We use this set of random numbers to fix the trajectory of events (arrival or departure) for all the experiments in this section. Similar to the experiment in the previous section, the number of users N and the arrival rate λ are fixed for 1000 steps and evolve according to the scenarios described in Fig. 4. The event is set to arrival if z_t is less than or equal to $\lambda_t/(\lambda_t + \min\{x_t, k\}\mu)$, and set to departure otherwise. These experiments were carried out during the training time for 10 different seeds. We plot the median of the number of times a system goes into an overloaded state (Fig. 7) and the number of requests offloaded by the system (Fig. 8) along with the uncertainty band from the first to the third quartile for every 1000 steps.

We observe in Fig. 7, that all the algorithms (SALMUT, PPO, A2C) 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 takes some time to learn this behavior. The observation is valid for all the different scenarios we tested. We observe that for Scenario-4, PPO enters the overloaded state at around $0.8 \cdot 10^6$ 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. The baseline algorithms, on the other hand, enters into the overloaded state quiet often.

We observe in Fig. 8, that the algorithms (SALMUT, PPO, A2C) 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 arrrival rate increases, the number of times the offloading occurs also increases in the interval. We see that even though the offloaded requests are higher for the RL algorithms than the baseline algorithm in all scenarios and timesteps, the difference between the number of times they offload is not significant implying that the RL algorithms learn policies that offload at the right moment as to not lead the system into an overloaded state. 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

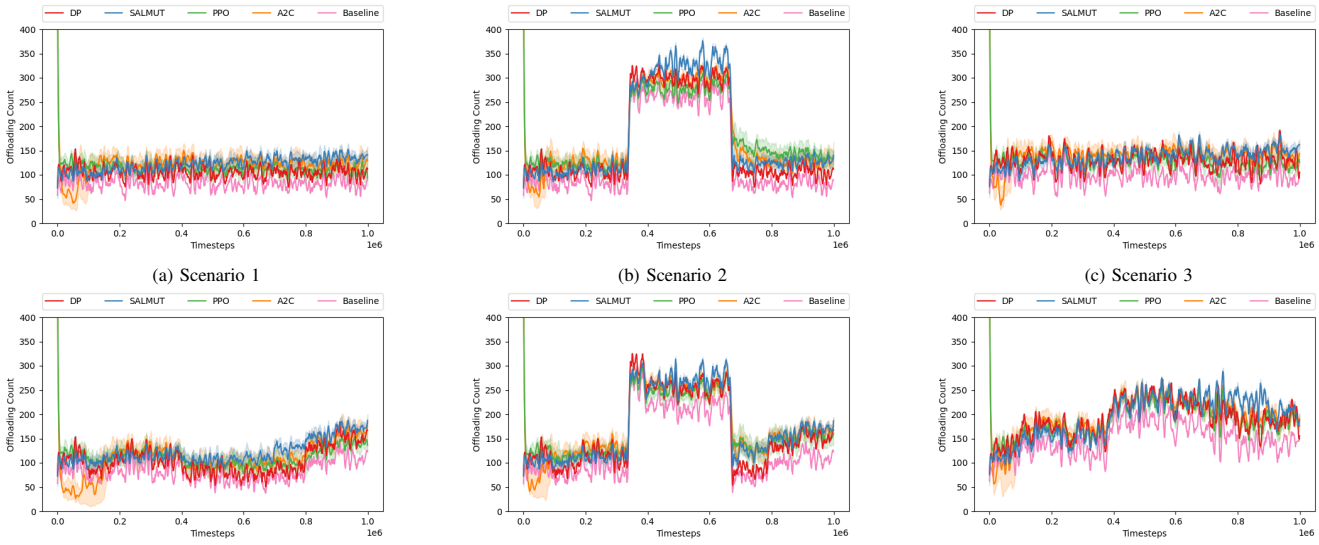
We test our proposed algorithm on a testbed resembling the MEC architecture in Fig. 1, but without the core network and backend cloud server. We consider an edge node which serves a single application. Both the edge nodes and clients are implemented as containerized environments in a virtual machine. The architecture of the testbed is shown in Fig. 9. The



(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.



(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.

load generator generates request 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 action outputted by the current 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 server 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.

In our implementation, the number of clients served by an edge node N and the request rate of the clients λ is constant for at-least 100 seconds. We define a step to be the execution of the testbed for 100 seconds. 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 run our experiments for a total of 1000 steps, where N and λ evolve according to Fig. 4 for different scenarios, similar to the previous set of experiments. We consider an

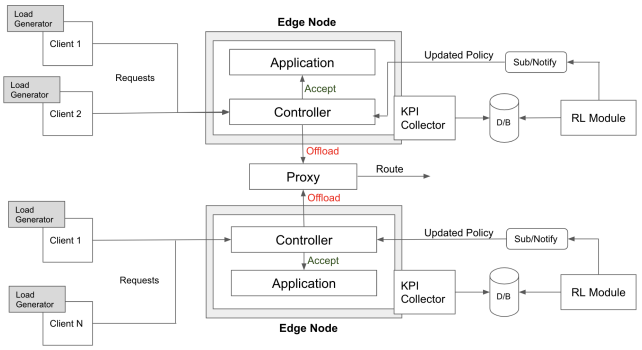


Fig. 9. Architecture for the docker-testbed we run the experiments on.

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 $\ell < 3$. We assume that the discrete time discount factor $\beta = \alpha/(\alpha + \nu)$ equals 0.99.

A. Results

We run the experiments for SALMUT and baseline algorithm for a total of 1000 steps. We do not run the simulations for PPO and A2C 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. The performance of SALMUT and baseline algorithm is evaluated at every step by computing the discounted total cost for that step using the cost buffers which are stored in the database. The experiment is repeated 5 times 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.

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 very quickly at the start and slowly converges towards optimal performance after 400 steps, 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 (Fig. 11) and the number of incoming request offloaded by the requests (Fig. 12) at every step. Let us define C_{ov} to be the number of times the system enters into an overloaded state and C_{off} to be the number of times the system offloads requests for every step (100s) of the training iteration.

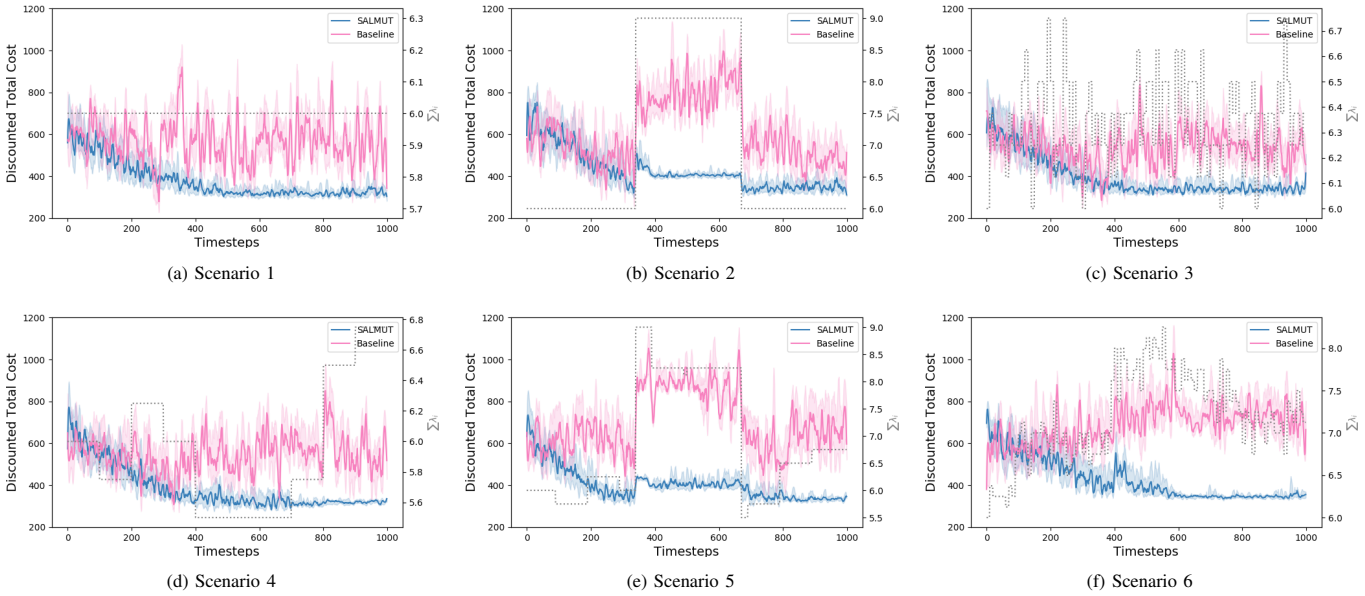
Fig. 11 shows that the number of times the edge node goes into an overload state while following policy executed by SALMUT is much less than the baseline algorithm. Even when the system goes into an overloaded state, it is able to recover quickly and does not suffer from performance deterioration. From Fig. 11b and 11e we can observe that in Scenarios 2 and 5, when the request load increases drastically (at around 340 steps), C_{ov} 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 recovers within 50 steps. 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 can 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 and Limitations

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. Therefore, our proposed algorithm is more suitable for deployment in real systems for online training.

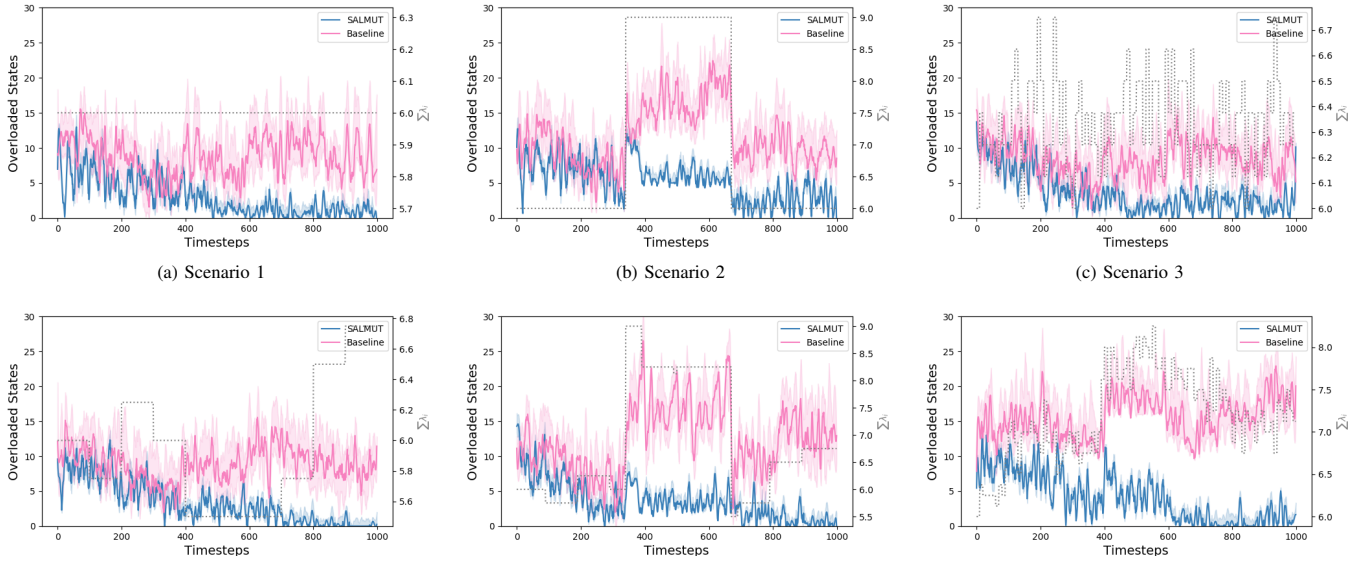
The results presented in this paper can be extended in several directions. In addition to CPU overload, one could consider other resource bottlenecks such as disk I/O, RAM utilization, etc. It may be desirable to simultaneously consider multiple resource constraints. Along similar lines, one could consider multiple applications with different resource requirements and different priority. The framework developed in this paper will be applicable to these more sophisticated setups as well provided the optimal policy has some kind of a threshold structure



(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. 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. 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 there are additional challenges such as routing, link failures, and changing network topology.

Appendix A
Proof of Proposition 1

Let $\delta(x) = \lambda/(\lambda + \min(k,x)\mu)$. We define a sequence of value functions $\{V_n\}_{n\geq 0}$ as follows

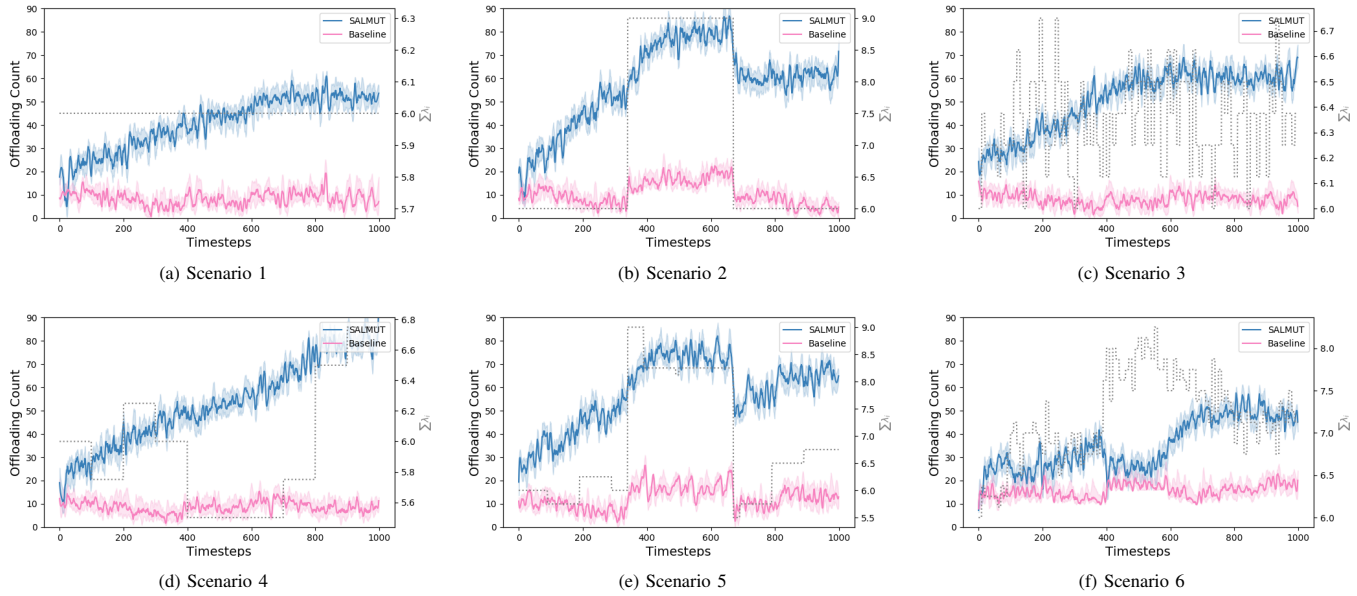
$$V_0(x,\ell) = 0$$

and for $n \geq 0$

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

where

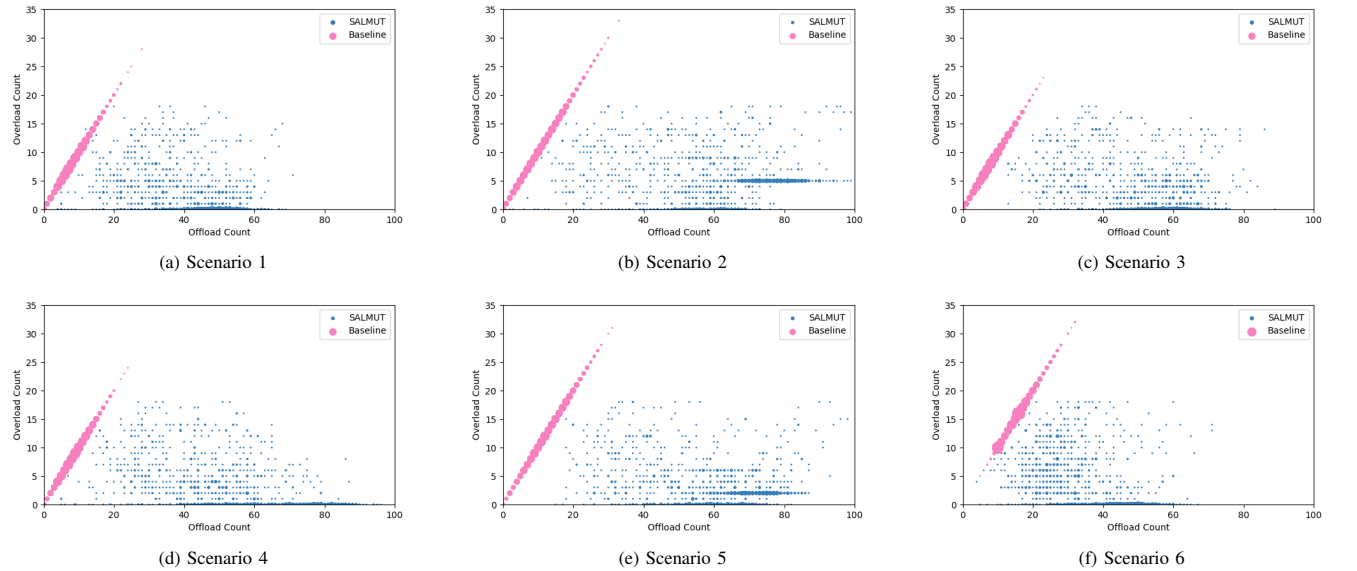
$$\begin{aligned} Q_{n+1}(x,\ell,0) = \frac{1}{\alpha+\nu}\Big[h[x-k]^+ + c(\ell)\Big] \\ + \beta\Bigg[\delta(x)\sum_{r=1}^{\mathsf{R}} P(r)V_n([x+1]_{\mathsf{X}},[\ell+r]_{\mathsf{L}}) \\ + (1-\delta(x))\sum_{r=1}^{\mathsf{R}} P(r)V_n([x-1]^+,[\ell-r]^+)\Bigg] \end{aligned}$$



(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.

and

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

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 [19], we know that

$$\lim_{n\to\infty} V_n(x,\ell) = V(x,\ell), \quad \forall x,\ell \tag{14}$$

where V is the unique fixed point of (6).

We will show that (see Lemma 1 below) each $V_n(x,\ell)$ satisfies the property of Proposition 1. Therefore, by (14) 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,

$$Q_{n+1}(x,\ell_1,0) = \frac{1}{\alpha+\nu}\left[h[x-k]^+ + c(\ell_1)\right]$$

$$
\begin{aligned}
&+\beta\Big[\delta(x)\sum_{r=1}^{\mathsf{R}}P(r)V_n([x+1]_{\mathsf{X}},[\ell_1+r]_{\mathsf{L}})\\
&\quad+(1-\delta(x))\sum_{r=1}^{\mathsf{R}}P(r)V_n([x-1]^+,[\ell_1-r]^+)\Big]\\
&\overset{(a)}{\leq}\frac{1}{\alpha+\nu}\big[h[x-k]^+ + c(\ell_2)\big]\\
&\quad+\beta\Big[\delta(x)\sum_{r=1}^{\mathsf{R}}P(r)V_n([x+1]_{\mathsf{X}},[\ell_2+r]_{\mathsf{L}})\\
&\quad+(1-\delta(x))\sum_{r=1}^{\mathsf{R}}P(r)V_n([x-1]^+,[\ell_2-r]^+)\Big]\\
&=Q_{n+1}(x,\ell_2,0), \qquad (15)
\end{aligned}
$$

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

$$Q_{n+1}(x,\ell_1,1)\leq Q_{n+1}(x,\ell_2,1). \qquad (16)$$

Now,

$$
\begin{aligned}
V_{n+1}(x,\ell_1)&=\min\{Q_{n+1}(x,\ell_1,0),Q_{n+1}(x,\ell_1,1)\}\\
&\overset{(b)}{\leq}\min\{Q_{n+1}(x,\ell_2,0),Q_{n+1}(x,\ell_2,1)\}\\
&=V_{n+1}(x,\ell_2) \qquad (17)
\end{aligned}
$$

■

where (b) follows from (15) and (16). Eq. (17) 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 B
Proof of Proposition 2

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

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

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 C
Proof of Optimality of SALMUT

Proof The choice of learning rates implies that there is a separation of timescales between the updates of (12) and (13). In particular, since $b_n^2/b_n^1\to 0$, iteration (12) evolves at a faster timescale than iteration (13). Therefore, we first consider update (13) 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:

$$
\begin{aligned}
[\mathcal{M}_\tau Q](x,\ell,a)=\big[\bar\rho(x,\ell,a)+\beta\sum_{x',\ell'}p(x',\ell'|x,\ell,a)\\
\min_{a'\in\mathcal{A}}Q(x',\ell',a')\big]-Q(x,\ell,a). \qquad (18)
\end{aligned}
$$

Then, the step-size conditions on $\{b_n^1\}_{n\geq 1}$ imply that for a fixed π_τ, iteration (12) may be viewed as a noisy discretization of the ODE (ordinary differential equation):

$$\dot Q(t)=\mathcal{M}_\tau[Q(t)]. \qquad (19)$$

Then we have the following:

Lemma 3 *The ODE* (19) *has a unique globally asymptotically stable equilibrium point Q_τ.*

Proof Note that the ODE (19) 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

$$
\begin{aligned}
\mathcal{B}_\tau[Q](x,\ell)=\big[\bar\rho(x,\ell,a)+\beta\sum_{x',\ell'}p(x',\ell'|x,\ell,a)\\
\min_{a'\in\mathcal{A}}Q(x',\ell',a')\big]. \qquad (20)
\end{aligned}
$$

Note that $\mathcal{B}_\tau$ is a contraction under the sup-norm. Therefore, by Banach fixed point equation, $Q=\mathcal{B}_\tau Q$ has a unique fixed point, which is also equal to Q_τ. The result then follows from [25, 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 (13), which is given by

$$\dot\tau=-\nabla J(\tau). \qquad (21)$$

Lemma 4 *The equilibrium points of the ODE* (21) *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, all local optima are locally asymptotically stable. ■

Now, we have all the ingredients to prove convergence. Lemma 3 and 4 imply assumptions (A1) and (A2) of [26]. Thus, the iteration (12) and (13) 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 (19):

$$\dot{Q} = \mathcal{M}_{\tau,\infty} Q \tag{22}$$

where,

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

It is easy to see that

$$[\mathcal{M}_{\tau,\infty} Q](x,\ell,a) = \beta \sum_{x',\ell'} p(x',\ell'|x,\ell,a) \min_{a'\in\mathcal{A}} Q(x',\ell',a') - Q(x,\ell,a) \tag{23}$$

■

Furthermore, origin is the asymptotically stable equilibrium point of (22). Thus, from [27], we get that the iterates $\{Q_n\}_{n\geq 1}$ of (12) 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.

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*. 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] 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 of Things Journal*, vol. 6, no. 3, pp. 4005–4018, 2018.

[14] 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.

[15] ——, "A deep reinforcement learning based offloading scheme in ad-hoc mobile clouds," in *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2018, pp. 760–765.

[16] 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.

[17] A. Jensen, "Markoff chains as an aid in the study of Markoff processes," *Scandinavian Actuarial Journal*, vol. 1953, no. sup1, pp. 87–91, 1953.

[18] R. A. Howard, *Dynamic Programming and Markov Processes*. The M.I.T. Press, 1960.

[19] M. Puterman, *Markov decision processes: Discrete Stochastic Dynamic Programming*. John Wiley and Sons, 1994.

[20] A. Roy, V. Borkar, A. Karandikar, and P. Chaporkar, "Online reinforcement learning of optimal threshold policies for Markov decision processes," *arXiv:1912.10325*, 2019.

[21] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv:1707.06347*, 2017.

[22] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," 2017.

[23] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[24] A. Raffin, A. Hill, M. Ernestus, A. Gleave, A. Kanervisto, and N. Dormann, "Stable baselines3," https://github.com/DLR-RM/stable-baselines3, 2019.

[25] 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.

[26] V. S. Borkar, "Stochastic approximation with two time scales," *Systems & Control Letters*, vol. 29, no. 5, pp. 291–294, 1997.

[27] 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 joined as a M.Sc. student in Computer Science at McGill University in 2018 and currently pursuing it. He is a research assistant at Montreal Intstitute of Learning Algorithms (MILA). His research interests include application of machine learning in communication and networks, and multi-agent reinforcement learning.

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 currently serves as Associate Editor of Springer Mathematics of Control, Signal, and Systems. He was an Associate Editor of the IEEE Control Systems Society Conference Editorial Board from 2014 to 2017. 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, real-time communication, information theory, and reinforcement learning.

Zhongwen Zhu Biography text here.

Hatem Abou-Zeid Biography text here.

Emmanuel Thepie Fapi Biography text here.

Hakimeh Purmehdi Biography text here.