This is a postprint version of the following published document: M. Ferens, D. Hortelano, I. de Miguel, R. J. D. Barroso and S. Kosta, "STEROCEN: Simulation and Training Environment for Resource Orchestration in Cloud-Edge Networks," *2024 15th International Conference on Network of the Future (NoF)*, Castelldefels, Spain, 2024, pp. 133-141, doi: 10.1109/NoF62948.2024.10741443.

STEROCEN: Simulation and Training Environment for Resource Orchestration in Cloud-Edge Networks

Mieszko Ferens
Aalborg University
Copenhagen, Denmark
mjfm@es.aau.dk

Diego Hortelano
Universidad Rey Juan Carlos
Madrid, Spain
diego.hortelano@urjc.es

Ignacio de Miguel
Universidad de Valladolid
Valladolid, Spain
ignacio.miguel@tel.uva.es

Ramón J. Durán Barroso
Universidad de Valladolid
Valladolid, Spain
rduran@tel.uva.es

Sokol Kosta
Aalborg University
Copenhagen, Denmark
sok@es.aau.dk

***Abstract*—Large scale deployment of Internet-of-Things (IoT) devices is projected to grow in the coming years. These devices are expected to be low-cost while supporting applications with growing computational demands. To enable the necessary computations, offloading of computational tasks to Edge and Cloud nodes is a fundamental technology. However, orchestration for such networks is a complex problem which affects both the network design and the decision system. To aid in solving this problem, simulation tools are essential for predicting the performance of networks in different conditions and under different orchestration policies. In this paper, we propose STEROCEN, a Cloud-Edge network resource orchestration simulation and training tool which allows for different configurations of up-to a four-layer network composed of: (i) end-device, (ii) Close Edge, (iii) Far Edge, and (iv) Cloud layers. Our tool collects delay metrics for flexibly defined applications, especially in regard to computation in the network nodes and including uncertainty in processing times. Additionally, the tool only needs the initial configuration and an independently defined orchestrator, allowing for testing of many strategies. As an example, we provide results of testing some Deep Reinforcement Learning (DRL) algorithms using the same training and simulation environment.**

***Index Terms*—Simulator, Orchestration, Edge-Cloud Computing, Computation Offloading, Reinforcement Learning, IoT.**

I. Introduction

The ever-growing demand for more complex and demanding applications puts significant stress on current and future network design [1]. High computation and strict latency requirements are especially challenging when dealing with low-cost devices, including Internet-of-Things (IoT) devices. Given the large expected growth of the IoT industry [2], this problem is a popular topic where many solutions are proposed.

One key solution is to offload computational tasks from devices with low resources to better equipped servers [1], [3]. By performing some or all computations remotely, small IoT devices can support applications for which they are locally under-equipped, which allows for the devices to, e.g., experience lower computational delays or conserve battery, among other advantages. However, the problem of orchestration of the computational task among the different network nodes, i.e., the decision of where to offload different tasks, has no simple solution. Different applications, network architectures and orchestration algorithms significantly affect the performance of the whole system, making the design of both network and orchestration algorithms a complicated but fundamental challenge [4], [5]. Moreover, testing different configurations on real networks is challenging due to limitations imposed by concerns for the effects that this can have on the quality of service provided on those networks [6]. Meanwhile, setting up large networks only for the sake of testing is expensive. For these reasons, network simulation tools are essential.

In this paper, we propose an open-source Python training and simulation tool called STEROCEN for testing the performance of different network configurations under the management of user-defined orchestration policies. The tool allows to define networks of up to four layers: (i) device layer, (ii) close Edge, (iii) far Edge, and (iv) Cloud. Each layer can be configured to have different nodes with different computational resources, with customizable communication links. Additionally, the tool allows for the applications running on the device-layer to be configured as well. Moreover, a standout feature is the configurability of processing uncertainty, which is typically neglected in other tools and accounts for unpredictable variations in the time to process computational tasks. All this enables a great number of easily definable network configurations, which can be tested against different orchestration algorithms. We provide an example of use for STEROCEN based on our previous work [7], where four Deep Reinforcement Learning (DRL) algorithms were trained and tested on a defined network, and compared against some heuristic algorithms. We now demonstrate how many more algorithms can be easily tested (sixteen), and we include

This work is supported by the IoTalentum project, funded by the European Union Horizon 2020 research and innovation program within the framework of Marie Skłodowska-Curie Actions ITN-ETN with grant number 953442. It has also been supported by Grant PID2020-112675RB-C42 funded by MCIN/AEI/10.13039/501100011033 and by Consejería de Educación de la Junta de Castilla y León and the European Regional Development Fund (Grant VA231P20).

an analysis on the processing time uncertainty, which was not previously considered. The results show that despite most algorithms failing to perform well due to the complexity of the defined scenario, using the tool we manage to find an orchestration policy which complies with the applications requirements close to 100% of the time. Moreover, in contrast to [7], the focus of this paper is to show the design and capabilities of the STEROCEN software tool and to make it available to the research community. As such the contributions of this paper are as follows:

- We provide an open-source Python training and simulation tool called STEROCEN for easily testing orchestration algorithms in different network and application scenarios.
- The tool allows for flexible network architecture, computational node, communication link, application configurations, and processing uncertainty.
- We provide an example of use of STEROCEN by defining a challenging orchestration scenario and finding an adequate policy using DRL that complies with application latency requirements, and analyze the effect of processing time uncertainty.

The rest of this paper is organized as follows. In Section II we give a basic explanation of computation offloading and DRL. Next, we provide an overview of similar works, i.e., other simulation tools, in Section III. Section IV describes the open-source tool and the configurability it possesses. An example use case with details of its experimental setup and results is presented in Section V. Then, Section VI discusses the flexibility and applicability of the network configuration. Finally, Section VII concludes the paper.

II. Background

In this section we briefly explain computation offloading and orchestration to ease the understanding of the STEROCEN simulation tool. Additionally, we provide an introduction to DRL for the convenience of the readers given its relevance to our use case example.

A. Computation offloading

Computation offloading consists of fully or partially processing computational tasks of applications running on some device remotely. Due to the resource limitations of low-cost devices (such as IoT devices), it is a popular solution for many applications [1], [3].

Many network architectures have been considered in the past for computation offloading [7]–[9], but generally three layers are defined: (i) Cloud, (ii) Edge, and (iii) end-device layers. Typically, the closer a node is to the end-device layer, the less resources it has available for local computations, but it experiences lower communication delays. However, it is possible to have networks diverting from these common ideas, specifically, some layers may be omitted or separated into multiple layers. Regarding the different metrics of interest, the most popular ones are related to node resources (energy and processing constrains), transmission and propagation delays of the network links, task inter-dependencies, privacy concerns, and mobility.

As described in Section IV, to provide as much flexibility as possible, STEROCEN defines the previous three layers, with the option to separate the Edge layer into two. Additionally, the nodes and resources can be defined without limitations, except for the Cloud layer. Regarding performance metrics, the focus is on the computational delays caused by limited computing resources on network nodes. Thus, the tool is useful for testing whether network resources can realistically handle some load and application requirements.

B. Deep Reinforcement Learning (DRL)

Reinforcement Learning (RL) is a field of machine learning which uses an agent to interact with an environment through actions. The environment is represented by a set of states, where transitions between states are determined by the action taken in each of them. By defining rewards associated to transitions between states, the RL goal is to find an optimal policy for taking actions at each possible state. This is done by the agent through trial-and-error by exploring the state-space of the environment and updating its own policy with the goal of maximizing the long-term accumulated reward.

This can be done with relatively simple algorithms like Q-learning [10] as long as the environment state-space is finite. However, in most applications the state representation of the environment is hard if not impossible to define as finite. In these situations, a mechanism called function approximation is required. Basically, states are mapped to a set of elements which vary between certain values (potentially allowing for infinite possible combinations). By the observation that, if two states are very similar the optimal actions are probably the same, an approximation to the optimal policy is still obtainable. In such cases, the policy must be implemented with, e.g., a neural network, to allow for approximating any function. Thus, DRL builds on the original idea by extending to problems with a very large or infinite state-space. It is worth noting that RL and DRL techniques have been widely proposed to address the computation offloading problem [1], [7]–[9], [11].

For a detailed explanation of RL we refer the reader to [10], and on the use of RL for computation offloading to [11].

III. Related Work on Simulation Tools

Multiple simulation tools which allow for testing different orchestration strategies have been made available. CloudSim [12] is an extendable simulation toolkit which supports cloud computing systems with application provisioning. For inter-cloud simulations, SimIC [13] is a discrete event simulator for multi-cloud collaborative and distributed services. Meanwhile, HeROsim [14] targets serverless cloud computing environments for dynamic cloud orchestration. Most recent proposals shift their focus towards edge computing. Sphere [5] is an edge computing simulator with flexible network topology, orchestration strategy, source traffic, and task scheduling. For edge and fog computing environments, iFogSim2 [6] allows

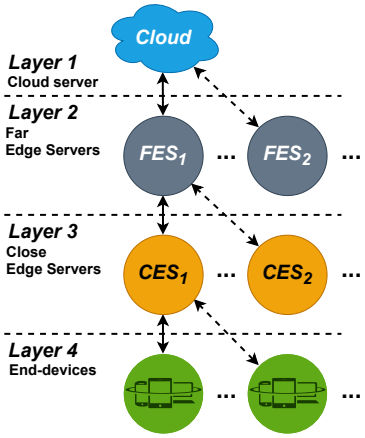

Fig. 1. Network definition in the tool.

simulations of service migration, dynamic distributed cluster formation, and micro-service orchestration. Regarding the testing of intelligent edge orchestration, ElSim [4] enables task offloading and resource pricing simulations using techniques such as DRL. Finally, focusing on network slicing, SliceNet [15] perform flow-level simulations for optimization of slicing techniques.

Based on the previous works we can see that there is a gap in Cloud-Edge combined simulation environments. Additionally, there is no readily available simulator that can model the processing time uncertainty. Thus, these two elements become the main design choices, as well as differentiating characteristics of our simulator, which also focuses on orchestration, but allows for a mix of user-configurable cloud and edge computing scenarios with processing time uncertainty as an explicit feature.

IV. STEROCEN Description

In this section we explain the inner-workings of STEROCEN[1], which defines a layered network with computation resources and treats applications running on end-devices as traffic sources. The following subsections explain the different elements that together allow for a customizable scenario to be simulated.

A. Network

We consider computation offloading in a hierarchical network to which generic end-devices are connected as shown in Fig. 1. The network has four layers: (i) Cloud server, (ii) Far Edge servers, (iii) Close Edge servers, and (iv) end-device communities. In this architecture, the previously mentioned Edge layer has been divided into Close and Far Edge to give more granularity in the compromise between computation resources and latency.

STEROCEN treats the Cloud layer as a special case compared to the rest. First, the tool expects that at least one node is defined in each layer. Each node i is defined to have a specific number of CPU cores (n^i_{CPU}), each capable of processing a single computational task at a time with a certain clock speed (CLK^i) and queue size (Q^i_{lim}). It is possible to define the resources of nodes from different layers independently, however nodes in the same layer are expected to all have the same resources. Second, the special Cloud layer needs to be defined with exactly one node. This single node only requires a clock speed value (CLK^0), as it is considered to have infinite cores and queue. Essentially it simulates a black-box node with virtually infinite resources. Another exception is the end-device community nodes, which group from one-to-many end-devices. The total number of end-devices is defined per simulation and distributed uniformly across all end-device community nodes. However, the end-devices in a community node each have their own equal set of resources.

The links that connect different nodes can be placed freely, but connecting nodes from non-adjacent layers (i.e., breaking the hierarchy) is not possible. It is possible however to connect Edge servers on the same layer with each other. Whichever the configuration, the tool uses shortest path routing to determine the paths that data transfers from and to end-devices will take when communicating with each node. The links are defined as bidirectional with specific transmission speeds and propagation delays. This means that each individual link from node i to j has a specific transmission rate r^{i-j}_{TX} in Mbps and propagation delay t^{i-j}_{prop} in ms.

It is worth noting that while breaking the hierarchy defined in Fig. 1 is not possible, it is possible to leave layers unused from the top. This means that one can define a network were an end-device community node is connected to a single Edge server (layer 3). Additionally, this Edge server can be defined with a large number of resources to emulate a large Cloud server. It is also possible to configure the network so that layer 2 and/or layer 3 nodes simply work as routers, by preventing the orchestrator from using them for computation offloading (see Section IV-C). Finally, while it is not possible to directly define the number of end-devices per community node explicitly (as the total is distributed across all end-device community nodes), more than one end-device community node can be connected to a single Edge server. This allows for non-uniform distribution of end-devices connected to different Edge servers, since multiple end-device community nodes are functionally equal to a single end-device community node with the sum of their end-devices.

B. Applications

To run simulations, a list of at least one application needs to be defined. Each end-device runs an independent instance of each defined application. Each application instance generates tasks as input packets with no dependencies based on an exponential distribution (Poisson process). Every time a packet is generated, it must be processed to generate an output packet that must be received at the originating node within a defined maximum latency. The applications are defined with six parameters, namely:

[1]Software and software architecture available at https://github.com/gcoUVa/STEROCEN

- Processing cost (A_c): The CPU cycles required to process each bit of a single input packet (i.e. task) for this application.
- Input packet size (A_{in}): The size of a single input packet in kilobits.
- Output packet size (A_{out}): The size of a single output packet in kilobits.
- Maximum latency (A_d): The maximum tolerable latency in milliseconds to receive the output packet of an input packet once generated.
- Average inter-packet time (A_t): The average time it takes for the following input packet (i.e., task) to be generated (that is, the mean of the exponential distribution).
- Priority (A_p): The priority of the application relative to the others.

C. Orchestration interface

During simulations, all offloading decisions are managed by an orchestrator implemented with, e.g., some DRL or heuristic algorithm. The orchestrator's task is to decide where to send each input packet for processing. There are no limitations on this except for the fact that tasks from one end-device can never be offloaded to another end-device. Note that processing a task locally (in the end-device that generated it) is a possibility. This means that the agent must choose an action which selects one of the network nodes or the local end-device for processing. This is defined as:

$$Action \in [0, K) \tag{1}$$

where K represents the number of candidate nodes, i.e., Cloud server, Far Edge servers, Close Edge servers, and the end-device. Notice that since the orchestrator is user defined, it is possible to prevent it from offloading to certain nodes, giving more flexibility to the defined scenarios.

For each request, the environment created by the tool will return a representation of the load of all CPU core queues in the network and end-device that generated the input packet at the time of the request handling. Some information on the parameters of the application which is involved with the request and the end-device location (Close Edge server to which it is connected) are also included. Additionally, the orchestrator is facilitated a precalculated estimated delay for offloading a packet of any defined application to any of the candidate nodes. All this is formally defined by having the orchestrator receive an observation per request which concatenates the previous pieces of information:

$$Obs = [Q_1, ..., Q_X, A_c, A_{in}, A_{out}, A_d, D_1, ..., D_K, N_1, ..., N_M] \tag{2}$$

where Q_x for $x \in [1, \cdots, X]$ is the current availability of all X CPU cores queues in the candidate nodes defined as (3) (with Q_{lim}^{node} the queue limit in ms for the respective node), D_k is the expected delay for the processing at node k if no other tasks are present and defined as (4) (with $t_{T,est}^k$ being the estimated total delay in ms), and N_m is the Close Edge server to which the end-device is connected (one-hot encoded).

$$Q_x = \frac{\text{free space in queue of CPU core x}}{Q_{lim}^{node}} \tag{3}$$

$$D_k = \max(0, 1 - \frac{t_{T,est}^k}{A_d}) \tag{4}$$

Thus, the orchestrator simply has to give the tool an action upon receiving an observation, and the simulation will move forward in time to the next request. Note that all this information is simply provided, but the orchestrator has no obligation to use it. When implementing an orchestrator for use with STEROCEN one may choose to, e.g., ignore some parts of the observation and only use others for the decision. Moreover, one can choose to ignore some of the observations and provide old ones to the orchestrator. For example, the observation provided to the orchestrator could be updated periodically (rather than for each request) to emulate a system where the orchestrator does not possess accurate network metrics for every single request.

As for the reward that the orchestrator may use to evaluate its performance and update its model, the tool currently provides two options: Penalty Reward (PR) and Weighted Binary Reward (WBR). These rewards are given with each observation, and account for the previous action. Once again, note that an orchestrator does not need to use the rewards, and it is also possible to create your own performance metrics with the tool's provided information. The rewards are based on the real total delay t_T that each packet experienced.

$$PR = \begin{cases} -1000, & \text{if unable to process} \\ -100 - (t_T - A_d), & \text{if } t_T > A_d \\ 0, & \text{if } t_T \leq A_d \end{cases} \tag{5}$$

$$WBR = \begin{cases} -A_p, & \text{if } t_T \geq A_d \text{ or unable to process} \\ +A_p, & \text{if } t_T \leq A_d \end{cases} \tag{6}$$

D. Scheduling

Whenever a request is handled by the orchestrator and a node for processing an application packet is selected, a scheduling process is initiated. The moment a node for processing is selected, that node is notified of the upcoming packet. The node will handle on its own the allocation of a time slot in the queue of one of its CPU cores. The allocation is performed based on a simple best-effort rule. The node will minimize the processing delay of the packet by allocating the earliest slot is has available taking into consideration all CPU core queues. However, already scheduled tasks cannot have their position in the schedule changed by this process.

The queues are not divided into fixed time slots, but instead use dynamic ones. This means that the node uses the estimated processing time ($t_{proc,est}$ as defined in (7)) and estimated time of arrival ($t_{arrival,est}$ as defined in (8)) for the packet to find a sufficiently large slot in the queue. This allocation does not take into account any uncertainty that may arise from variable transmission or propagation delays, as well as processing time variations.

$$t_{proc,est} = \frac{A_{in} \cdot A_c}{CLK^{node}} \tag{7}$$

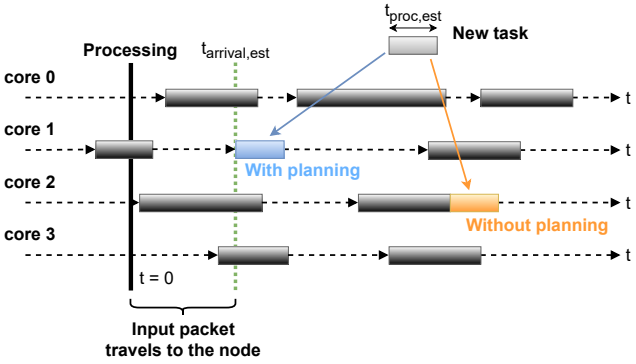

Fig. 2. Scheduling process for a task with and without planning.

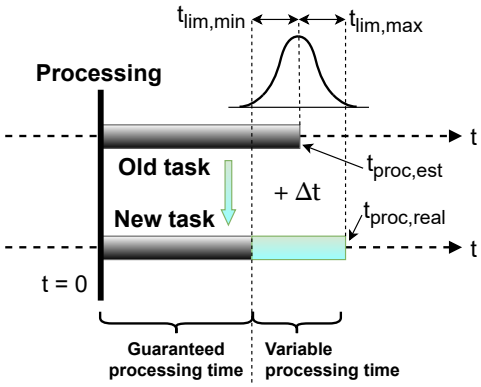

Fig. 3. Uncertainty of processing time for a scheduled task based on Gaussian noise.

$$t_{arrival,est} = \sum_{links} (r_{TX}^{link} \cdot A_{in} + t_{prop}^{link}) \tag{8}$$

There are two possible mechanisms for deciding the slot in a queue: with or without planning. 'With planning' employs a void-filling mechanism [16] which considers sufficiently large slots between scheduled tasks as suitable. 'Without planning' forces the new task to be scheduled at the end of the queues [16]. Fig. 2 illustrates an example of the scheduling process. The state of the queues changes between requests as time moves forward, but the scheduling is performed based on a snapshot at the time of the request.

E. Processing noise

Regarding uncertainty (i.e., noise), we only consider noise in the processing times of computational tasks of CPU cores, since STEROCEN is intended to study the effect of different orchestration techniques given a certain network with some computational resources. While other variations in delays due to, e.g., transmission and propagation delays can affect system performance, this has been extensively studied to date [1]. On the other hand, to the best of our knowledge, no study on the effect of processing time uncertainty has been conducted, therefore we choose to isolate the effect of this factor and include it as a configurable element.

We model the uncertainty in processing time based on an additive Gaussian noise, where the standard deviation of the normal distribution is proportional to the total processing time of a single task. This is done by defining a specific value for the Coefficient of Variation (CV) and calculating the standard deviation (as defined in (9)). Since the mean will be taken as the estimated processing time ($t_{proc,est}$), the noise affects longer tasks more, i.e., it is proportional to the number of bits to process. We limit the minimum and maximum time that can be respectively added or subtracted from $t_{proc,est}$ by defining $t_{lim,min}$ and $t_{lim,max}$. This is illustrated in Fig. 3.

$$\sigma = CV \cdot t_{proc,est} \tag{9}$$

Whenever a task that was scheduled requires more time than expected, the following task in the respective queue might become affected. If this happens the following task is simply delayed until the current task has finished processing. This of course can lead to the second task in the queue pushing the third, and so on. In those cases, they are all delayed the minimum time possible using any time slots in between the tasks. If a task is pushed in the queue to the point at which it goes over Q_{lim}, then it is discarded and will affect the next reward as if the task had not been processed.

F. Limitations

STEROCEN has certain limitations which we now proceed to explain. First, we do not consider mobility in the network. Second, the tool also does not consider end-devices exiting and entering the network during the simulation. This is realistic for systems where end-devices are stationary, but deviates from others, e.g., vehicular applications. That said, considering that if the maximum latency per packet is very low and packets are independent, an end-device changing its location will not affect the overall results significantly, because the probability of an end-device changing its connected Close Edge server is very low per packet. Another important factor is task dependencies, which are ignored by our simulator in its current iteration but planned for future updates.

We also apply a simplification to our design which involves the assumption that all communications that do not transport application data are instantaneous, i.e., the delay of the control plane which transports network information to the orchestrator (physically located on some computing node) and its decisions to the end-devices and network nodes. This simplification includes the specific protocol interactions that are required to perform offloading. This is done because we consider that these exchanges are much lighter than the application data packets. Still, if considered, the main difference would be that the orchestrator managing the offloading would receive requests and send commands with some delay. This adds to the final delay that has to be considered in the application latency, and on average can be estimated using the defined propagation delay values. Finally, while device heterogeneity is accounted for by considering end-devices as generic, our simulator does not currently support the definition of multiple types of end-

TABLE I
DEFINED APPLICATIONS FOR EXAMPLE USE CASE.

App	Processing cost, A_c (CPU cycles/bit)	Input packet size, A_{in} (kbits)	Output packet size, A_{out} (kbits)	Max latency, A_d (ms)	Avg. inter-packet time, A_t (ms)	Priority, A_p
1	1	700	200	1	100	1
2	400	10	5	4	100	1
3	1000	900	25	500	100	1
4	3000	100000	25000	200000	100	1
5	200	650	500	200	100	1
6	500	40	1	30	100	1

TABLE II
LINK PARAMETERS FOR EXAMPLE USE CASE.

	Tx speed, r_{TX} (Mbps)	Propagation delay, t_{prop} (ms)
Layer 1-2	10^4	50
Layer 2-3	10^4	20
Layer 3-4	10^1	0.1

TABLE III
NODE PARAMETERS FOR EXAMPLE USE CASE.

	CPU cores, n_{CPU}	Clock speed, CLK (GHz)	Core queue length, Q_{lim} (ms)
Cloud server	∞	3.6	-
Regional Data Center	16	3.6	200
MEC server	8	2.4	200
End-device	2	1.2	200

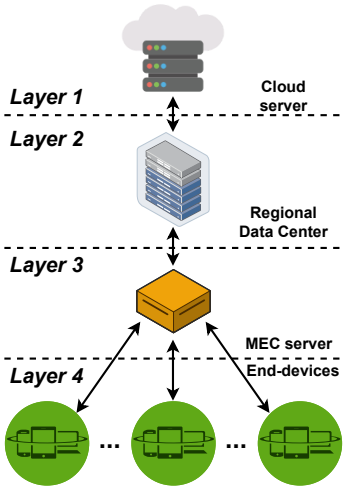

Fig. 4. Network architecture for example use case.

devices for a single simulation. Both the device heterogeneity and the control plane delays are planned as priority future extensions to the simulator.

V. STEROCEN USE CASE

This section provides an example use case for STEROCEN, where we first describe the experimental setup, and then give details on what could be achieved by analyzing the results. There are many possible experiments that can be performed, e.g., evaluate the network offloading cost compared to processing on the end-devices, testing orchestration algorithms, etc. Here we provide one example.

A. Experimental setup

We run the simulations using a system with i7-11800H CPU, with no GPU or multi-threading. STEROCEN is based on Python 3.8. We have used the open-source library ChainerRL [17], which provides a set of DRL algorithms. In particular, we have used Deep-Q Network (DQN), Double DQN (DDQN), Residual DQN, Categorical DQN, Categorical DDQN, Advanced Learning (AL), Persistent AL (PAL), Double PAL, Dynamic Policy Programming (DPP), SARSA, Asynchronous Avantage Actor-Critic (A3C), PCL, Trust Region Policy Optimization (TRPO), Proximal Policy Optimization (PPO), REINFORCE, and Implicit Quantile Networks (IQN).

The configuration of the network architecture is set in Comma-Separated Value (CSV) files by specifying the node and link parameters in Tables II and III, and is available in the GitHub repository. In this paper, we define the network architecture based on [7] as illustrated in Fig. 4. We consider a four-layer network with 50 end-devices connected to a Multi-access Edge Computing (MEC) server. The MEC server is in turn connected to a Regional Data Center (RDC) which also grants access to a Cloud server with virtually infinite resources. It should be emphasized that multi-node MEC and RDC networks are possible as well.

The parameters used for links are described in Table II and are shared for all links connecting the same type of nodes. Meanwhile, the resources of all types of nodes are presented in Table III. Regarding the traffic, we define six applications with mixed requirements, the details of which can be seen in Table I. All the values are based on the scenario defined in [7].

Based on the observations from [7] we will only use WBR in this paper, since it improves the performance of DRL, and ignore the already tested heuristic algorithms. All test procedures are included in the GitHub repository.

TABLE IV
BEST OBTAINED CONFIGURATION FOR AL ALGORITHM.

Hyperparameter	Best
Discount factor (γ)	0.5
Exploration type	Linear decay
Exploration probability (ϵ)	From 0.25 to 0.05 in $2.5 \cdot 10^5$ steps
Replay buffer size	$5 \cdot 10^5$
Replay start	$5 \cdot 10^4$
Target update interval	$2.5 \cdot 10^4$
Activation function	$reLU()$
Number of hidden layers	1
Number of neurons per hidden layer	60
Weight of persistent advantages (α)	0.9

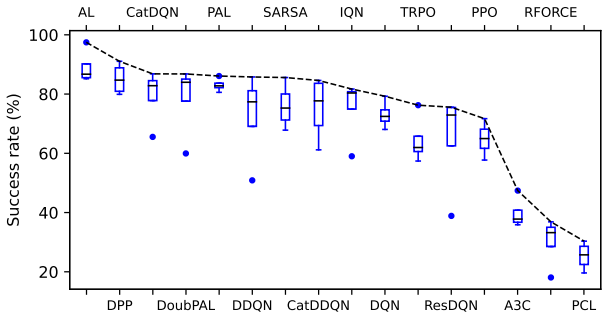

Fig. 5. DRL algorithm performance comparison with best results for each.

B. Results

The goal of this use case is to show how STEROCEN can be used for different purposes. Thus, we explain how we performed the following: (i) training and optimization of orchestration algorithms (DRL in this case), (ii) performance comparison of different orchestrators, (iii) analysis of orchestration policy and its effects, and (iv) analysis of the effect of processing uncertainty on orchestration policy.

1) Training and optimization of algorithms: First, STEROCEN can be used to train and optimize algorithms. In our use case we do this with DRL algorithms which require both optimization of hyperparameters and training of their neural networks. To train the algorithms we allow them to orchestrate the simulated network with a random seed up to, e.g., 10^6 requests. The trained algorithms can then be tested with an independent simulation of, e.g., 10^5 requests. This process can be repeated with different algorithm configurations for optimization, where some metrics are considered. In this use case we define (using information provided by the tool) the success rate defined as the ratio between the number of application tasks that are processed complying with their respective latency requirements, and the total number of handled requests.

While we did perform an in-depth optimization process for all sixteen considered DRL algorithms, the focus of this paper is STEROCEN and not the analysis of DRL algorithms. Thus, for brevity, we provide the final obtained configuration of the AL algorithm in Table IV, which showed the best results in the comparison we provide next. For a better understanding of this configuration we refer the reader to [17], [18].

2) Performance comparison: By using STEROCEN it is easy to compare a variety of orchestrators (based on DRL algorithms in this case). We simply have to run a simulation with the same network configuration to test all of them independently. Additionally, we can choose to simulate the network with the same or different random seeds, which determines the arrival of the application requests. Since DRL algorithms are typically trained multiple times until a good model is obtained, for our use case we choose to test four different random seeds on all sixteen algorithms. As mentioned before, this is simple using the tool, as we just have to change the seed for training and testing to obtain the final metrics.

The result of this process gives the performance of the algorithms in all runs based on our success rate metric and is shown in Fig. 5. As discussed previously, and in-depth analysis of the DRL techniques (including their latency for offloading) is not the focus on this paper, however looking at the comparison we can clearly differentiate between algorithms that perform well in the network and those that do not. Based on these results we can conclude that the AL algorithm performs better than the rest and is the best candidate for an orchestrator if the system were to be implemented in real-life. Moreover, we can observe that this algorithm has been successfully trained once to obtain a success rate of 97%. Thus, we could now use this trained orchestration policy and apply it to the same network in real-life. While there might be some differences due to the limitations discussed in Section IV-F, this is still a more feasible deployment practice than training the algorithm from scratch on the real network.

3) Analysis of orchestration policy: The previous two steps of our analysis allowed us to optimize the configuration of algorithms and train many orchestrators to find a suitable one for the defined network. Now we proceed to use the information provided by the tool to analyze the best policy. Such an analysis is important to provide granular information on why the policy works and what is the quality of service for each of the defined applications. There are many metrics one could choose to define and monitor, but for simplicity we focus on each application individually both in success rate and average total delay. Additionally, we define the processing rate as the number of tasks that are processed in a node, but whose output data packet does not necessarily reach the end-device within the required latency.

Table V shows the success rates, processing rates and average total delay of all applications for the best policy. With this information we clarify that the orchestrator does in fact deal with the processing of all applications regardless of their frequency and requirements. Note that the average delay of App 1 is higher than its maximum latency because a few tasks during the simulation are offloaded to have a very high latency (thus the success rate is not 100%). Additionally, in Table VI we show the distribution of tasks for each application

TABLE V
INDIVIDUAL APPLICATION PERFORMANCE WITH BEST POLICY.

	Success rate	Processing rate	Average total delay (ms)
App 1	98.00%	100%	1.25
App 2	98.53%	100%	3.69
App 3	100%	100%	482.88
App 4	100%	100%	95998.53
App 5	88.11%	100%	182.18
App 6	100%	100%	12.74

TABLE VI
OFFLOADING DISTRIBUTION FOR EACH APPLICATION.

	Cloud server	RDC	MEC	End-device
App 1	0%	0.03%	24.73%	75.24%
App 2	0%	0%	0.08%	99.92%
App 3	100%	0%	0%	0%
App 4	100%	0%	0%	0%
App 5	0%	0.43%	4.79%	94.78%
App 6	0%	0%	100%	0%

across the four nodes where the agent can decide to offload. We chose to collect this data during the simulations too, as it provides valuable insight into how the orchestrator decided to deal with the requests. We can conclude that the AL algorithm did in fact learn the differences between applications and distributes their tasks across the network with regard to their maximum tolerable latency and the nodes' processing capabilities. Moreover, we can see that since not all tasks from any certain application are always offloaded to the same node, the orchestrator considers the network load at the time of each request.

To reiterate, our goal with this analysis is to showcase the possibilities of using STEROCEN, not to analyze the orchestrators themselves. We show how with the information provided by the tool we can collect our own defined metrics for the specific use case at hand and obtain a provably suitable orchestration policy for the network.

4) Analysis of the effect of processing uncertainty: A differentiating characteristic of STEROCEN is that is allows for the configuration of processing time uncertainty (noise) to simulate the CPU core queues non-deterministically. We note that, to the best of our knowledge, we have not found any study that takes this factor into account. Thus, to conclude our use case example we repeat the previously described analysis but with noise in the processing times defined with $CV = 1$ and $[Q_{lim,min}, Q_{lim,max}] = [0, 2t_{proc,est}]$ (see Section IV-E). We do not repeat the optimization or comparison process and only apply the AL algorithm with the parameters from Table IV.

The results can be seen in Table VII and show an expected degradation in performance. Out of four runs, the best policy obtained an average success rate of 77.53%, which is reasonable seeing the success rates of individual applications. Notice how we can still see that the orchestrator manages to maintain a very good processing rate for all applications. Based on these results, we can conclude that the added noise affected the system significantly as expected.

TABLE VII
INDIVIDUAL APPLICATION PERFORMANCE IN NOISY SCENARIO.

	Success rate	Processing rate	Average total delay (ms)
App 1	67.60%	100%	8.23
App 2	52.12%	100%	7.60
App 3	100%	100%	482.88
App 4	100%	100%	95998.53
App 5	47.77%	94.79%	189.29
App 6	98.35%	100%	16.83

There are plenty of further analyses that could be conducted. For example, we could apply the best policy from the AL algorithm trained without noise on the noisy simulation and compare its performance with the one that was trained directly with processing noise. We could also tweak the noise values to test how resilient we can make the network with a specific orchestrator to different levels of processing uncertainty. Given the focus of this paper on STEROCEN, we choose to exclude such details here, but encourage future research in this direction.

VI. DISCUSSION ON TOOL APPLICABILITY

In the presented use case we have analyzed a network configuration which easily adapts to the hierarchy that must be followed in STEROCEN, as described in Section IV-A. It is important to note that, while it is not possible to break this hierarchy, this does not significantly limit the flexibility of the configuration. Notice that the tool does not consider the computational nodes in the different layers in different ways, except for the Cloud layer. The only exception to this is that offloading to other end-devices is forbidden. Additionally, users can design their orchestrator to not offload to certain nodes of the network. Thanks to these design choices, one can use the Edge layer nodes as computational nodes or simply as hops in the communication towards a higher layer. It is also possible to exclude any layers above the second layer from the offloading decision. Since the computational resources of the nodes can be made as large or are small as desired, it is also possible to simulate a virtually infinite computational node in the Edge servers[2].

All in all, it is possible to define networks of two, three or four layers, with communication links connecting any Edge servers to any other and to the Cloud server (this may require using an Edge server as a router).

VII. CONCLUSION

In this paper, we have presented STEROCEN, an open-source resource orchestration computational offloading simulation and training tool for evaluation of customizable Edge and Cloud computing scenarios. A hierarchical network architecture is created with (i) a Cloud server, (ii) one or more Far

[2]However, we do not recommend this approach and instead the Cloud layer should be used for this, since processing very large queues makes the simulations run slower.

Edge servers, (iii) one or more Close Edge servers, and (iv) generic end-devices. The bidirectional communication links of the network can be placed freely with individual transmission and propagation parameters, and the computation nodes are defined with specific resources each (number of CPU cores, clock speed, and size of their queue). While the network configuration must follow certain rules, we provide information on how to simulate networks which do not explicitly follow a four-layer hierarchy. The orchestrators used with the tool have to comply with a flexible interface, allowing for a variety of algorithms to be applied. Moreover, the processing time can be made uncertain by means of additive Gaussian noise, which is a feature that, to the best of our knowledge, is absent from other studies.

STEROCEN focuses on the effect of different orchestration algorithms, providing a simplified communication modeling, but with more detailed models of computational tasks in the queues of individual CPU cores. The tool also has two additional limitations in the form of end-device mobility and control plane communications. However, we argue how these simplifications can be handled so that estimations of the effects of additional factors can be included in the simulations.

We have also presented an example use case where we used the tool to optimize, train and compare 16 different DRL algorithms in a computation offloading scenario. We showed how this allows us to identify adequate hyperparameter configurations and suitable algorithms for the problem at hand. This information is useful before any real deployment and is an inexpensive method to pre-train orchestrators. The results show how we are able to effectively obtain a good orchestration policy for the use case, and we are able to explain the policy with additional information obtained from the simulations. Moreover, we compared these results with those obtained when considering processing uncertainty.

The tool can be downloaded from https://github.com/gcoUVa/STEROCEN. We expect that interested researchers may use this tool to aid in their endeavors in designing and testing computational offloading orchestrators. We also consider future extensions for the simulator which would have the goal of adding additional features. Specifically, end-device mobility and heterogeneity, communication delay uncertainty, and control plane delays with a defined physical location for the orchestrator on the network are planned and will enhance the capabilities of the tool.

REFERENCES

[1] A. Shakarami, M. Ghobaei-Arani, and A. Shahidinejad, "A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective," *Computer Networks*, vol. 182, p. 107496, 2020. [Online]. Available: https://doi.org/10.1016/j.comnet.2020.107496

[2] Transforma Insights, "Number of internet of things (iot) connected devices worldwide from 2019 to 2023, with forecasts from 2022 to 2030 (in billions) [graph]," Sep. 2023. [Online]. Available: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

[3] ETSI, "Multi-access edge computing (mec)." [Online]. Available: https://www.etsi.org/technologies/multi-access-edge-computing

[4] H. Kokkonen, S. Pirttikangas, and L. Lovén, "Eisim: A platform for simulating intelligent edge orchestration solutions," 2023. [Online]. Available: https://www.proquest.com/working-papers/eisim-platform-simulating-intelligent-edge/docview/2885672248/se-2

[5] D. Fernández-Cerero, A. Fernández-Montes, F. Javier Ortega, A. Jakóbik, and A. Widlak, "Sphere: Simulator of edge infrastructures for the optimization of performance and resources energy consumption," *Simulation Modelling Practice and Theory*, vol. 101, p. 101966, 2020, modeling and Simulation of Fog Computing. [Online]. Available: https://doi.org/10.1016/j.simpat.2019.101966

[6] R. Mahmud, S. Pallewatta, M. Goudarzi, and R. Buyya, "ifogsim2: An extended ifogsim simulator for mobility, clustering, and microservice management in edge and fog computing environments," *Journal of Systems and Software*, vol. 190, p. 111351, 2022. [Online]. Available: https://doi.org/10.1016/j.jss.2022.111351

[7] M. Ferens, D. Hortelano, I. de Miguel, R. J. Durán Barroso, J. C. Aguado, L. Ruiz, N. Merayo, P. Fernández, R. M. Lorenzo, and E. J. Abril, "Deep reinforcement learning applied to computation offloading of vehicular applications: A comparison," in *2022 International Balkan Conference on Communications and Networking (BalkanCom)*, 2022, pp. 31–35. [Online]. Available: https://doi.org/10.1109/BalkanCom55633.2022.9900545

[8] Y. Bai, X. Li, X. Wu, and Z. Zhou, "Dynamic computation offloading with deep reinforcement learning in edge network," *Applied Sciences*, vol. 13, no. 3, 2023. [Online]. Available: https://doi.org/10.3390/app13032010

[9] T. Alam, A. Ullah, and M. Benaida, "Deep reinforcement learning approach for computation offloading in blockchain-enabled communications systems," *Journal of Ambient Intelligence and Humanized Computing*, Jan 2022. [Online]. Available: https://doi.org/10.1007/s12652-021-03663-2

[10] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[11] D. Hortelano, I. de Miguel, R. J. Durán Barroso, J. C. Aguado, N. Merayo, L. Ruiz, A. Asensio, X. Masip-Bruin, P. Fernández, R. M. Lorenzo, and E. J. Abril, "A comprehensive survey on reinforcement-learning-based computation offloading techniques in edge computing systems," *Journal of Network and Computer Applications*, vol. 216, p. 103669, 2023. [Online]. Available: https://doi.org/10.1016/j.jnca.2023.103669

[12] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011. [Online]. Available: https://doi.org/10.1002/spe.995

[13] S. Sotiriadis, N. Bessis, N. Antonopoulos, and A. Anjum, "Simic: Designing a new inter-cloud simulation platform for integrating large-scale resource management," in *2013 IEEE 27th International Conference on Advanced Information Networking and Applications (AINA)*, 2013, pp. 90–97. [Online]. Available: https://doi.org/10.1109/AINA.2013.123

[14] V. Lannurien, "Herosim: An allocation and placement simulator for evaluating serverless orchestration policies," 2024. [Online]. Available: https://hal.science/hal-04468894

[15] V. KumarSkandPriya, A. Dandoush, and G. Diaz, "Slicenet: a simple and scalable flow-level simulator for network slice provisioning and management," 2023. [Online]. Available: https://www.proquest.com/working-papers/slicenet-simple-scalable-flow-level-simulator/docview/2878534089/se-2

[16] N. Harki, A. Ahmed, and L. Haji, "Cpu scheduling techniques: A review on novel approaches strategy and performance assessment," *Journal of Applied Science and Technology Trends*, vol. 1, no. 1, pp. 48 – 55, May 2020. [Online]. Available: https://doi.org/10.38094/jastt1215

[17] Y. Fujita, P. Nagarajan, T. Kataoka, and T. Ishikawa, "Chainerrl: A deep reinforcement learning library," *Journal of Machine Learning Research*, vol. 22, no. 77, pp. 1–14, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-376.html

[18] M. G. Bellemare, G. Ostrovski, A. Guez, P. Thomas, and R. Munos, "Increasing the action gap: New operators for reinforcement learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, Feb. 2016. [Online]. Available: https://doi.org/10.1609/aaai.v30i1.10303