POLITECNICO DI TORINO
Repository ISTITUZIONALE

Adaptive SDN Autoscaling via Generalizable Multi-Agent Reinforcement Learning with EAGLE

Original
Adaptive SDN Autoscaling via Generalizable Multi-Agent Reinforcement Learning with EAGLE / Monaco, D., Sacco, A., Esposito, F., Marchetto, G.. - In: IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT. - ISSN 1932-4537. - 23:(2026), pp. 6507-6522. [10.1109/TNSM.2026.3717361]

Availability:
This version is available at: 11583/3013690 since: 2026-08-06T09:55:01Z

Publisher:
IEEE

Published
DOI:10.1109/TNSM.2026.3717361

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

08 October 2026

Adaptive SDN Autoscaling via Generalizable Multi-Agent Reinforcement Learning With EAGLE

Doriana Monaco, *Student Member, IEEE*, Alessio Sacco, *Member, IEEE*, Flavio Esposito, *Member, IEEE*, and Guido Marchetto, *Senior Member, IEEE*

***Abstract*—Network automation and data-driven solutions are critical for modern network efficiency and resilience, especially as traditional manual management struggles to keep pace with increasing demands. The programmability of Software-Defined Networks (SDN) and their integration with Machine Learning (ML) algorithms have significantly advanced this quest, enabling data-driven network configuration. Within this context, autoscaling network resources is essential for efficiently operating softwarized and virtualized networks. However, current autoscaling solutions are limited by their logically centralized nature and their poor model portability across different network domains. To address these challenges, we propose EAGLE, a Multi-Agent Reinforcement Learning (MARL) system that autonomously orchestrates the scaling of network resources to meet flow demands and reduce power consumption. Powered by graph embedding, it generalizes to diverse network settings. We evaluate our solution over a Mininet-based emulator and assess its generalization capabilities on Fabric, a large-scale network testbed. Our results show that our approach can reduce flow completion time (FCT) by up to 40% while achieving up to 15% power savings. In addition, we show that our trained model can "zero-shot generalize" to unseen network topologies that share structural or statistical similarity with the training domain, hence reducing training time and associated energy costs.**

***Index Terms*—Autonomous networks, reinforcement learning, marl, graph embedding.**

I. Introduction

NETWORK automation and data-driven networking solutions are prominently proliferating, given the recent advances in artificial intelligence, which enable network efficiency, resiliency, and adaptive orchestration [1], [2]. This is because traditional manual network management becomes impractical and error-prone as the demand for seamless connectivity and real-time data transmission increases.

For this reason, *network autoscaling*, through the (de)activation of switches and links, is essential to ensure that softwarized and virtualized networks can dynamically adjust their resource consumption, such as computing power, bandwidth, and energy, based on fluctuating workloads and user demands [3], [4], [5], [6]. For example, during peak usage times or unexpected traffic spikes, the network can *autonomously* allocate additional resources to temporarily alleviate the congestion, resulting in a more resilient and responsive network infrastructure. Similarly, idled resources can be turned off to reduce energy consumption and minimize carbon footprint, making the system more cost-effective while maintaining optimal performance [7], [8].

Network automation has been favored by the advent of Software-Defined Networks (SDN): data plane devices continuously monitor network status, which is then analyzed at a higher level to make decisions that modify the topology at run-time and result in new flow rules to be injected. In such a way, autoscaling actions can be the output of Machine Learning (ML) models [9] and Reinforcement Learning (RL) in particular [10], [11], [12], [13], [14]. RL offers a learning-based approach that is ideal for this problem since it uses feedback from the network environment to refine its policy. Its effectiveness is amplified by the robust control and observability of the SDN architecture, which allow the model to learn the relationship between network status and autoscaling decisions despite a complex and constantly changing network environment.

However, current autoscaling solutions are limited by two main factors. First, the majority of them rely on a logically centralized process running on a single controller [10], [12], leading to the explosion of the input space for large topologies and a rise of the propagation delay when the network size increases [15]. Second, all RL-based approaches are overly tailored to the training environments (*i.e.,* the number of nodes and links and traffic patterns) and therefore necessitate retraining to apply the model in new network environments [16], [17].

To cope with such limitations, in this paper, we propose an SDN solution that distributes the management of smaller subnets across multiple controllers, with the global aim of **E**fficient **A**utoscaling via **G**eneralizable multi-agent reinforcement **L**earning **E**nvironment (EAGLE). While defining such a network infrastructure comprising multiple controllers, we encountered and resolved several challenges. The main contributions of the paper are:

- We propose a novel heuristic algorithm that extends the Louvain method [18] for network partitioning. This algorithm balances load across controllers and minimizes

Received 18 July 2025; revised 5 June 2026 and 22 July 2026; accepted 23 July 2026. Date of publication 27 July 2026; date of current version 3 August 2026. The work has been partially supported by NSF OAC 2201536 and NSF CNS 2133407. The associate editor coordinating the review of this article and approving it for publication was F. Poltronieri. *(Corresponding author: Doriana Monaco.)*

Doriana Monaco, Alessio Sacco, and Guido Marchetto are with DAUIN, Politecnico di Torino, 10129 Turin, Italy (e-mail: doriana.monaco@polito.it; alessio_sacco@polito.it; guido.marchetto@polito.it).

Flavio Esposito is with the Department of Computer Science, Saint Louis University, St. Louis, MO 63103 USA (e-mail: flavio.esposito@slu.edu).

Digital Object Identifier 10.1109/TNSM.2026.3717361

© 2026 The Authors. This work is licensed under a Creative Commons Attribution 4.0 License.
For more information, see https://creativecommons.org/licenses/by/4.0/

TABLE I
COMPARISON WITH RELATED WORKS

Method	Learning-based	Distributed Architecture	Collaboration	Energy Aware	Scalable	Generalizable
SmartFCT [10]	✓	×	-	×	×	×
DeepConf [12]	✓	×	-	×	×	×
Mystique [13]	✓	✓	×	✓	✓	×
CATE [22]	×	×	-	✓	✓	-
EAGLE	✓	✓	✓	✓	✓	✓

controller-to-switch latency by incorporating custom objective metrics into the partitioning process.

- We develop a fully decentralized RL framework in which each controller runs an Actor-Critic (AC) model to provision adequate network resources while accounting for power consumption. These agents collaboratively learn a joint policy using a pruning-based strategy that reduces the size of exchanged model weights and thus conserves bandwidth.
- We enhance the generalization capability of our model by employing graph embedding techniques to produce low-dimensional, structurally informative representations of network topologies and traffic patterns. This input design is easy to interpret and enables the model to scale across diverse and unseen network scenarios, while reducing the MARL communication overhead.
- We extensively evaluate EAGLE in both emulated (Mininet [19]) and real-world scenarios (FABRIC testbed [20]). Results show that our system outperforms both learning-based and non-learning-based autoscaling solutions in terms of flow completion time, throughput, and delay, and can reduce the resource usage, expressed as power consumption. EAGLE outperforms the benchmarks even in a "zero-shot generalization", *i.e.,* without re-training, and even in large-scale networks as the GtsCe [21]. By training our solution only once, we can limit the time and costs of this process and favor the portability of learned decisions.

The rest of the paper is organized as follows: Section II surveys related work concerning autoscaling networks and generalizable methodologies, Section III provides an overview of the solution and provides the problem definition, while a detailed description of all the components is given in Section IV. The collaborative learning aspect is explored further in Section V, while Section VI shows the experimental setup and evaluation, and we draw conclusions in Section VII.

II. RELATED WORK

In this section, we provide an overview of autoscaling methodologies, with a focus on recent ML-based techniques in the context of SDN. Additionally, we explore the techniques adopted to enable generalization capabilities in RL models. We present a summary of the key differences between our solution and similar works in Table I.

A. Auto-Scaling Networks

Approaches used to implement adaptive network-level scaling mechanisms that modify link activation, switch availability, and routing behavior based on traffic conditions can be categorized primarily as reactive or proactive. The former approach aims to make adjustments when under-provisioning effects are observed; of course, this may lead to service interruptions for a certain period. A typical example of reactive approach is the threshold-based, where the threshold can be whether fixed or dynamic based on the application. The proactive approach focuses on predicting the phenomenon to adapt in time, which requires more precision, or reacting to the change before a service disruption.

Besides several heuristics [22], [23], proactive solutions also include the use of Machine Learning techniques for the analysis activity and reasoning. Among these, we can cite fuzzy learning [24], time-series [3], [4], [25], control theory [7], [8], and genetic algorithms [26], but RL emerges as the most employed technique by far thanks to its capacity of learning without a-priori knowledge and the possibility to define a multi-objective reward-based system [9], [27], [28], [29]. Although there are various domains of application of RL for autoscaling [9], [27], [28], [29], the most similar work to ours is the application of RL to Software-defined networks.

SmartFCT [10] is a flow consolidation scheme designed to satisfy QoS requirements in data center networks using Deep Reinforcement Learning (DRL). The DRL agent receives information about the current throughput, sends it to the neural network, and outputs a set of link margin ratios. Based on this ratio, flows are rescheduled following a best-fit decreasing bin-packing algorithm. This also allows shutting down unused links and devices. The multi-objective purpose is described by the number of shut-down devices deducted by the number of flows whose FCT was violated. In DeepConf [12], the authors present a topology augmentation use case that autonomously activates/deactivates resources based on the traffic demand. The state input is given by the topology matrix and the traffic matrix. The action results in a set of links to activate. The reward function used is given by the link utilization divided by the flow completion time of the last time step flows. Unlike them, our RL model jointly considers throughput and power consumption, and is deployed among multiple controllers to make our distributed setting scalable and resilient. Although recent multi-agent approaches are presented in [13] and [14], the former (Mystique) trains individual policies to optimize local networks rather than finding a global optimum, the latter inherently achieves collaboration by sharing a global network state and including all agent actions as input of the RL model. Conversely, our collaborative algorithm trains a joint policy for globally aware networked agents that share their knowledge in a compressed format to limit communication overhead. All the previously mentioned RL-based solutions are limited in

TABLE II
SUMMARY OF SYMBOLS AND NOTATION USED IN THE PAPER

Symbol	Description
G	Network graph
V	Set of network nodes
E	Set of network links
w_l	Capacity of link l
k	Number of traffic flows
τ	Time step
Ψ_u	Binary status of switch u (1 = on, 0 = off)
$f_i(\tau, \Psi)$	End-to-end throughput of flow i at time τ
$f_i^l(\tau, \Psi)$	Traffic of flow i traversing link l at τ
Th	Average network throughput
PC	Total network power consumption
$cs(u)$	Power consumption of switch u
P_{idle}	Idle power consumption of a switch
P_{dynamic}	Dynamic power coefficient of a switch
$Z_i(u, v)$	Flow i uses link (u, v)
B_u	Neighbors of node u
$\Phi_{u,v}$	Status of link between nodes u and v
C	Set of candidate controllers
n	Number of nodes, $n = \|V\|$
L_i	Load associated with switch i
U_j	Capacity of controller j
d_{ij}	Distance between switch i and controller j
ν_{ij}	Switch i assigned to controller j
η_j	Controller j is deployed
h	Detected communities (subnets)
c_i	Community (subnet) i
γ_i	Controller capacity in community i
ϵ	Load imbalance tolerance
ρ	Load imbalance penalty
Q_p	Modularity of partition p
z_i	Sum of weights incident to node i
$\zeta(c_i, c_j)$	Nodes i, j in the same community
ΔQ	Modularity gain after reassignment
Z	Global state embedding vector
π_θ	Policy with parameters θ
g	Path length parameter in embedding
e	Number of evaluation points in embedding
Z_i	Local embedding produced by agent i
cn_t	Communication network among agents
N	Set of learning agents (controllers)
$J(\theta)$	Global long-term average return
r_{t+1}^i	Reward obtained by agent i at time $t+1$
$R(s, a)$	Globally average reward function
$R^i(s, a)$	Local reward function of agent i
λ_t^i	Parameters of agent i's reward estimator
$A_\theta(s, a)$	Advantage function
$V_\theta(s; v)$	State-value function parameterized by v
μ_t^i	Estimate of long-term average return
θ_i	Policy (Actor) parameters of agent i
v_i	Critic parameters of agent i
δ_t^i	Local temporal-difference (TD) error
π_{θ_i}	Policy of agent i

their adaptability to other contexts, as the state input design is tailored to the characteristics of a particular network. This implies that the model cannot be easily reused on a network with different sizes and topologies and must be retrained. One of the key differences in our approach is that we create a generalizable RL model by encoding the network topology and traffic to represent multiple-structured and multiple-sized topologies in a reduced state space.

B. Generalizability of Reinforcement Learning

In recent years, there has been a growing interest in developing network optimization frameworks that can generalize to unseen data. To achieve this, researchers have explored the use of graph embeddings and Graph Neural Networks (GNNs) for their ability to learn from graph-structured data.

Heo et al. [30] deploys a graph gated recurrent neural network (GG-RNN) to train their Service Function Chaining (SFC) model to connect Virtual Network Functions (VNF) to serve user requests. The training is performed on a dataset containing multiple topologies and VNF placement configurations, enabling generalization to random topologies at test time, although the dataset may not fully capture the complexity and dynamics of real-world networks.

One of the most explored scenarios is routing: [31] deploys a GNN with the Proximal Policy Optimization (PPO) algorithm. Similarly to them, we use a one-to-one mapping of the network to a graph and incorporate network status information through nodes and links. RouteNet [32] is a framework for network modeling and optimization that is trained with a message-passing GNN. The training is performed with a dataset containing samples from three well-known networks and generalizes to unseen networks and traffic conditions. GNNs can capture intricate patterns in the graph, but multi-layer message passing aggregates information from neighboring nodes in a complex, non-linear way, obscuring the original contributions of individual nodes. On the other hand, graph embeddings produce fixed, low-dimensional vectors via simpler, unsupervised methods, such as random walks [33]. These embeddings can be easily visualized using techniques such as t-SNE, allowing us to create clear 2D or 3D plots that reveal distinct patterns, such as clusters (see Fig. 4). This straightforward visualization makes it particularly user-friendly for identifying relationships and structures in the data. An example of graph embedding utilization is provided in GRL-PS [34], where a DeepWalk technique is used to produce the embedding for the network status, which is used as input for the path selection algorithm.

Unlike them, however, we adopted graph embedding to generalize to unseen data without retraining the model and to enable agents to collaborate efficiently. Starting from the training topology, each controller creates a dynamic virtual graph that captures traffic and topological details, then embeds this graph into a compact, low-dimensional array that preserves its semantic and structural properties. We leverage these low-dimensional embeddings to share information across distributed and collaborative agents, which finally concatenate the embeddings to create a global representation for the decision-making module. We show that integrating graph embeddings

creates a generalizable and interpretable collaboration system with reduced communication overhead.

III. EAGLE Overview and Problem Formulation

In this section, we first motivate the need for an autonomous mechanism for scaling network devices. We then describe the system features and components, and finally formalize the underlying problem.

A. Motivation and Background

Handling sudden traffic spikes, such as during a major event like a football match or concert stream, poses a significant challenge for network operators. During these high-demand periods, certain routers face heavy utilization while others remain underutilized or idle. Conversely, during off-peak hours, traffic drops drastically, leaving much of the infrastructure consuming energy without contributing to performance. These inefficiencies increase operational costs, waste energy, and demand constant oversight to avoid performance bottlenecks. Manual intervention to redistribute traffic or optimize resources is not only impractical but also error-prone. At its essence, determining the optimal number of devices to activate can be modeled as a multi-commodity flow problem (formally defined in Section III-C), which is typically addressed through the application of linear programming (LP) solvers. While LP solvers are capable of providing optimal solutions, they often struggle to scale effectively in response to the increasing complexity and size of modern networks. A fast approach is represented by heuristics, such as threshold-based solutions [35], [36], but they can hardly satisfy carrier-grade standards like reliability and stability.

ML-driven autoscaling approaches emerge as a natural solution to handle the complexity of computer networks and meet the need for fast response to traffic spikes [10], [13], [37]. Well-designed algorithms can identify recurring patterns in traffic, predict future demand, and dynamically adjust resource allocation–powering down underutilized routers during low-demand periods and reactivating them seamlessly as traffic rises. This automated system reduces reliance on manual adjustments, minimizing errors and ensuring consistent responsiveness.

B. Control Loop and System Components

Our autoscaling system operates in discrete time steps and follows a closed-loop process comprising a telemetry phase and a reaction phase, as shown in Fig. 1. Assuming a large-scale network, multiple controllers are deployed to manage smaller subnets. During the telemetry phase, each network controller actively monitors the system to detect failures or congestion. While in the context of an SDN topology, a network node typically refers to a layer-2 switch, our model functions effectively when the nodes act as either routers or switches. To represent the current network state, each controller collects utilization statistics from the physical infrastructure and constructs a weighted virtual graph, where edge weights reflect the load on the corresponding links. This graph is then encoded using a graph embedding technique, producing

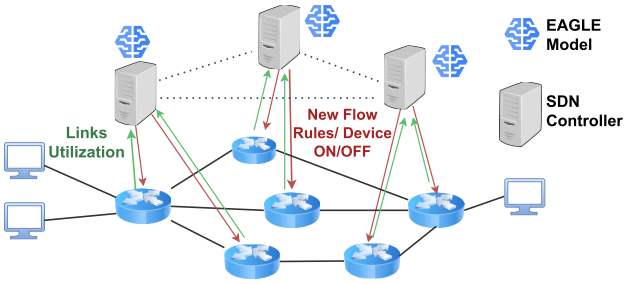


Fig. 1. EAGLE agents adapt to network demand fluctuations by learning from incoming metrics and react by proactively installing new paths on demand.

a compact representation that serves as the input state for the RL model. In this way, congestion is implicitly captured by the information embedded in the graph representation. During training, the RL agent explores different scaling decisions and receives rewards that encourage congestion mitigation while discouraging the activation of unnecessary resources. When congestion is detected or anticipated, the system enters the reaction phase. Based on the learned policy, the neural network determines outputs a binary action for each controllable switch ($0 = \text{off}$, $1 = \text{on}$). Following any topology change, the routing module updates the forwarding rules accordingly. In congestion scenarios, additional switches and links can be activated, creating new end-to-end paths between hosts. Since routing relies on Equal-Cost Multi-Path (ECMP), the availability of additional paths enables traffic to be distributed more evenly across the network, thereby alleviating congestion. After each reconfiguration, controllers exchange their updated local views to maintain a consistent global representation of the network state. Once congestion is mitigated and the additional resources employed become superfluous, the system intervenes by shutting down those assets.

In addition to congestion detection, our system also aims to respond to the failure of a network resource, *i.e.,* a node or link. In such cases, a similar approach is taken, involving the creation of one or more replacements. The system is also resilient to misconfiguration or failure of one of the MARL agents, as it redirects the management of its nodes to the working controllers, ensuring the continuation of the process without disruptions. Following the execution of these actions, the controller continues to monitor the system to ensure the restoration of normal traffic operation.

C. Network Model and Problem Formulation

Let the network be modeled as an undirected graph $G = (V, E, w)$, where V is the set of vertices (switches or routers), E is the set of links, and w_l is the capacity of link l. As in the multi-commodity flow problem, k flows are identified by $k_i = (src_i, dst_i)$ where src_i and dst_i are the source and destination of commodity i. In this setting, learning agents (network controller) receive information on the whole network status at each time step τ, and their goal is to mitigate the effects of congestion while simultaneously reducing management costs. The objective function thus aims to maximize network throughput while minimizing power consumption. For

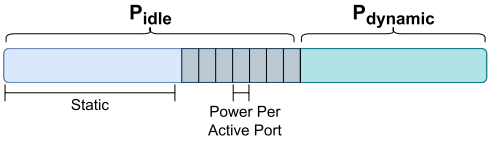


Fig. 2. Power consumption of a router.

a given switch u, Ψ_u is the binary operator defining its status, *i.e.,* on/off, while Ψ is the vector containing status indicators for all the switches. The end-to-end throughput of a flow in the time slot τ is denoted by $f_i(\tau, \Psi)$, as it depends on the available switches, while $f_i^l(\tau, \Psi)$ identifies the amount of traffic flowing along the link l. We indicate the average network throughput as follows:

$$Th = 1/k \sum_{i=1}^{k} f_i(\tau, \Psi), \tag{1}$$

and the system power consumption PC as:

$$PC = \sum_{u \in V} cs(u)\Psi_u, \tag{2}$$

where $cs(u)$ is the power cost function of switches. As in other studies in the literature, e.g., [22], we model this cost as:

$$cs(u) = P_{idle} + P_{dynamic} * rate, \tag{3}$$

where $P_{idle}(W)$ includes a static component, *i.e.,* not depending on the traffic, for device consumption and for each active port, $P_{dynamic}(W/Mbps)$ models the energy consumption variation per data rate, $rate(Mbps)$ is the data rate. The energy savings resulting from deactivating the switches is a straightforward concept. However, as illustrated in Fig. 2, it is equally important to recognize that turning off links–and consequently deactivating active ports–also contributes significantly to reducing idle power consumption. It is important to note that this power model constitutes a simplification: real network devices may exhibit non-linear or step-based power consumption profiles due to hardware characteristics such as port activation thresholds, dynamic ASIC behavior, or cooling mechanisms.

In our decentralized setting, we assign an agent n to the management of a subnet, whose problem to solve is formalized as follows:

$$\underset{\Psi}{\text{maximize}} \quad Th_{norm} - \alpha PC_{norm} \tag{4}$$

$$\text{s.t.} \quad \sum_{i=1}^{k} f_i^l(\tau, \Psi) \leq w_l, \forall l \in E, \forall \tau \tag{5}$$

$$\sum_{l=(u,v)\in E} Z_i(u,v) \cdot f_i^l(\tau, \Psi) = f_i(\tau, \Psi), \forall i, \forall \tau \tag{6}$$

$$\Phi_{u,b} \leq \Psi_u, \forall u \in V, \forall b \in B^u \tag{7}$$

$$\sum_{b \in B^u} Z_i(u,b) \leq 2, \forall u \in V, \forall b \in B^u \tag{8}$$

$$\Phi_{u,v} = \Phi_{v,u}, \forall (u,v) \in E \tag{9}$$

$$\Psi_u \in \{0,1\}, \Phi_{u,v} \in \{0,1\}, \forall u \in V, \forall (u,v) \in E, \tag{10}$$

where PC_{norm} is the normalized power cost and Th_{norm} is the normalized throughput, *i.e.,* $PC_{norm} \in [0,1]$, $Th_{norm} \in [0,1]$ to make them comparable, α is a coefficient that regulates the importance of the power consumption factor over the network performance, B^u is the set of vertices connected to node u, $Z_i(u,b)$ is a binary variable which is 1 if flow i traverses the link (u,b). Focusing on the constraints, (5)-(6) are flow conservation constraints, while (7) reinforces that if a switch is shut down, also the associated links are disabled. Additionally, since distributing flows across multiple links may lead to unwanted packet reordering effects, hindering transport-layer performance, constraint (8) imposes that a switch receiving flow i, forwards it entirely to one outgoing link, using a total of 2 attached links per flow. Finally, we establish that links operate bidirectionally with (9): hence, the cost of keeping a link active is incurred regardless of the direction of the flow (uplink or downlink).

The optimal solution to this problem involves selecting the proper number of switches to (de)activate while satisfying the traffic constraints. However, the combination of continuous and integer variables makes it a mixed-integer linear program. Due to the computational complexity needed to solve such problems, especially when dealing with large networks, we propose to address the problem with an RL-based solution. Thanks to its trial-and-error approach, it is able to learn without a-priori knowledge, and after our training process, it can be deployed in real networks and make decisions faster than traditional autoscaling methods. Even though each agent has its own utility function, we have developed a collaborative learning framework in which they can share network and RL model information, as illustrated below.

IV. EAGLE Model & Design

In what follows, we detail the strategies employed to overcome the main challenges posed by the problem, leaving the details of the collaborative protocol for Section V.

A. Controller Placement Problem

An inherent challenge of our solution is the placement of controllers and the assignment of a managing controller per switch, which can be a major issue in SDN. The load on the agents can be highly unbalanced, and controller-to-switch latency seriously affects network performance, including action execution and responsiveness to failures [15]. If this communication suffers from high latency, the autoscaling decisions and new routing configurations will be delayed, making the action ineffective if traffic demand has changed. Since we propose a solution that responds to real-time traffic characteristics, our goal is to reduce such delays and balance controller load, as heavily loaded instances suffer from performance degradation [38]. To do so, we partition the network into compact structures (subnets), which we denote as communities, using a balanced cut algorithm. A controller is placed in each community, subject to the specific constraints described below.

This problem can be reduced to the minimum k-median problem, which is known to be NP-hard [39]. Therefore, numerous heuristics have been proposed to address

it efficiently. Typically, two steps are performed: network partitioning (through community detection) and then location selection. Given a network $G(V, E, w)$, we first want to cut the network into h disjoint communities $c_1, .., c_h$, such that $V = c_1 \cup c_2 \ldots \cup c_h$ and that $c_i \cap c_j = \emptyset \forall i \neq j$. If the cut is balanced, the sizes of the communities are similar. Among community detection algorithms, the Louvain method [18] stands out as one of the most robust and scalable solutions. This approach is based on the concept of modularity of a partition p, a measure of the density of links within communities, defined as:

$$Q_p = \frac{1}{2m} \sum_{i,j} \left(w_{ij} - \frac{z_i z_j}{2m} \right) \zeta(c_i, c_j), \tag{11}$$

where $m = \frac{1}{2} \sum_{i,j} w_{ij}$, z_i is the sum of the weights of edges connected to node i, ζ returns 1 if c_i and c_j are the same community, 0 otherwise. The modularity gain ΔQ is then defined as the difference between the modularity of a new partition p' and the previous partition p, as $\Delta Q = Q_{p'} - Q_p$. The algorithm is composed of two phases that are iteratively executed until no significant modularity gain is achieved. At startup, each node is assigned to its own community. The first phase computes, for each node, the modularity gain of relocating it to its neighbors' communities and moves it to the community that maximizes this gain. A second phase of the algorithm generates a new network that maps the identified communities to supernodes.

While the Louvain heuristic design is sound, it is known to produce highly variable community sizes [18]. To cope with such suboptimality, we design a modified instance of this algorithm that balances the partition sizes and, consequently, the load on controllers. In particular, assuming that the number of switches in a community is the variable that contributes the most to the controller $i-th$'s load, such load corresponds to $|c_i|$, and our algorithm aims to limit $|c_i|$ below the controller capacity, defined as γ_i. Meanwhile, for each pair of communities, we define the load balancing condition as: $||c_i| - |c_j|| < \epsilon \forall i, j \in h$, where ϵ is a small enough variable. The algorithm incorporates γ and ϵ as follows: once the best destination community is found, if the relocation doesn't respect the load balancing condition, it is discouraged through a penalty term ρ in the modularity gain, as $\Delta Q = Q_{p'} - Q_p - \rho$. Additionally, the node is relocated only if the load of the new partition does not exceed γ_i.

Once the subnets are identified, the controller is placed on the physical node that minimizes the average controller-to-switch latency within each subnet, measured as the number of nodes between the switch and the physical machine hosting the controller functions.

We model the network as an undirected graph with V nodes and E edges. C identifies the set of controllers to place, and every switch is OpenFlow-compliant, so it can host a controller. L_i is the load of a switch i, U_j refers to the capacity of a controller j and let $\mathcal{P}_{ij}$ denote the set of all paths between nodes i and j. Then, the hop-distance between i and j, which also determines the switch-controller latency, is defined as:

$$d_{ij} = \min_{p \in \mathcal{P}_{ij}} \sum_{(u,v) \in p} 1. \tag{12}$$

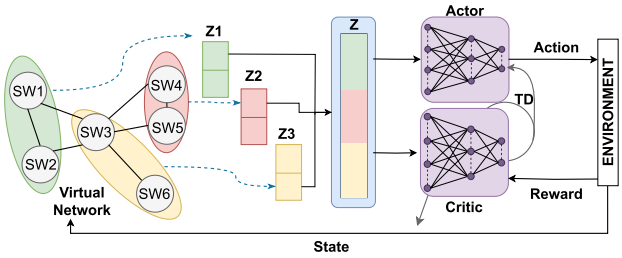


Fig. 3. EAGLE agent is composed of an Actor-Critic (AC) model fed with the network information encoded through graph embedding.

Let the binary variables $\nu_{ij} = 1$ if switch i is managed by controller j, otherwise 0, and $\eta_j = 1$ if controller j is deployed. The problem is formulated in the following way with $n = |V|$:

$$\text{minimize} \quad \frac{1}{n} \sum_{i \in V} \sum_{j \in C} d_{ij} \nu_{ij} \tag{13}$$

$$\text{s.t.} \quad \sum_{j \in C} \nu_{ij} = 1, \forall i \in V \tag{14}$$

$$\sum_{j \in C} \eta_j = h \tag{15}$$

$$\nu_{ij} \leq \eta_j, \forall i \in V, \forall j \in C \tag{16}$$

$$\sum_{i \in V} L_i \nu_{ij} \leq U_j \eta_j, \forall j \in C \tag{17}$$

$$\nu_{ij}, \eta_j \in [0, 1], \forall i \in V, \forall j \in C, \tag{18}$$

where (14) establishes that every switch must be associated to one controller, (15) limits the number of controllers placed to h as the number of communities detected and (16) imposes that switches are assigned only to the placed controllers. The load balancing constraint (17) limits to exceed the capacity of controllers in terms of the number of managed switches. Finally, (18) enforces the binary nature of ν and η.

B. DRL-Based Network Resource Scaling

In the following section, we describe the Deep Reinforcement Learning (DRL) setup to deploy agents capable of scaling resources up and down. The DRL setup is commonly described through a Markov Decision Process (MDP) by a quadruple $M = \langle S, A, P, R \rangle$, where S is the state space and represents the network environment in a certain time instant, A is the action space, P represents the probability to transition from a state s to a state s', R is the reward function. DRL finds a control policy that maximizes the long-term return by means of neural networks. The main parameters shown in Fig. 3 are:

State space: using our defined collaborative protocol, the agents can monitor the whole network utilization to identify congestion or failure occurrence and react properly. Therefore, we model the network status using a virtual network graph $G'(V', E', w')$, where G' is a $1:1$ mapping of current active resources in the network and each edge $l' \in E'$ is associated with a weight $w'_{l'}$ representing the utilization in a time slot τ. To make the approach more scalable, each agent monitors a

subnet, constructs a virtual subgraph, embeds it into a low-dimensional array of fixed size (see Section IV-C for details), and shares the result with other agents. After such parameter exchange, all the embeddings are concatenated to obtain a unique embedding Z, whose vector is then normalized and used as input for the DRL model.

Action space: the model computes a scaling action determining which switches to activate/deactivate. The action space is thus determined by a binary action $\Psi_u \in A$ for switches $u \in V$. If the status of a switch changes, the corresponding node u' is added (removed) to the virtual graph G', and the routing module updates the forwarding rules. Turning off a switch also results in shutting down the connected links.

Reward: it represents the function that the agent needs to maximize, so we directly employ the utility function, defined in Equation (4). Since it favors actions that increase the network throughput, it indirectly mitigates congestion.

To drive the learning process, we employ an Actor-Critic (AC)-based model [40]. In this architecture, the Actor outputs the action, which is then evaluated by the Critic. This lets us enable collaboration because our agents share the Critic parameters and reach consensus on the global action (see Section V). Finally, actions modify the environment, which computes a reward signal to encourage/discourage such actions, and updates the state representation for the next time step.

C. Graph Embedding and Curriculum Learning for Generalization

The state distribution is the key for generalization, as a trained model behaves similarly across similarly distributed data. Our goal is to establish a common data space where different networks coexist. We define zero-shot learning as the capacity of our RL model to respond effectively to congestion and underutilization in previously unseen network topologies and traffic patterns, distinct from those encountered during training. Unlike one-shot or few-shot learning, zero-shot learning does not require any additional model updates or training: we train the model once and apply the learned policy to new environments [41]. To create such a generalizable model, we use a fixed-size state representation via graph embedding and train it via curriculum learning.

Graph embedding encodes graphs into low-dimensional continuous vector spaces, preserving some properties such as their structure, node proximity, and link connectivity [42]. By leveraging graph embedding, we can ensure that networks with varying configurations are mapped into a comparable feature space, effectively aligning their distributions. This alignment is crucial because it allows the model to recognize patterns and relationships among diverse traffic scenarios, even if they originate from different topologies. When different networks are represented uniformly, the RL model can generalize its learning more effectively, as it can draw on similarities across these representations to inform its decisions in unseen situations.

In particular, we use a Feather Graph algorithm [33], which studies the distribution of neighborhood features in the graph and evaluates them in some discrete points e to obtain an embedding. This method explores paths of length g between pairs of nodes and iteratively chooses the next node to visit based on the weight assigned to each edge. At each step, it collects each node's contribution to the studied feature into a vector, which is then pooled using permutation-invariant functions to obtain the whole graph fingerprint as an embedding. This is crucial because differently ordered nodes can still represent the same graph, and permutation-invariant functions guarantee that the order will not influence the resulting embedding. In this way, isomorphic graphs will have the same representation.

Curriculum learning is an increasingly adopted technique to facilitate the training process. Unlike traditional RL training, which samples training environments from a fixed distribution at each iteration, curriculum learning varies the training environment distribution to gradually increase the difficulty, so that training will see more environments that are more likely to improve performance. In our case, we established the curriculum offline and defined five network traffic conditions of increasing difficulty. Prior work utilizing RL models, e.g., [43], has shown the promise of curriculum learning, including faster convergence, higher asymptotic performance, and better generalization. This process is recognized as *incremental learning*: specifically, we apply Domain Incremental Learning, in which a domain is characterized by a particular data distribution that can be generated by a new data source or context.

V. Collaborative Learning in EAGLE

We now detail our collaborative protocol that agents use to train a joint policy through a consensus-based approach.

A. Collaborative AC-Marl

Multi-Agent Reinforcement Learning (MARL) studies the interaction of multiple learning agents in a common environment. As we want to optimize the performance of the overall network, we deploy a collaborative MARL setting, where agents operate for the optimization of a joint objective. In this paper, we model networked agents as part of a time-varying communication network cn_t, which they use to exchange information with their neighbors at time t. By using this (either virtual or physical) network, we reduce communication overhead compared to a centralized approach, and the collaborative protocol is robust to agent failures, as operations continue without interruption by simply removing that agent from cn_t.

We design the protocol to enable collaboration towards maximizing a global long-term return, wherein the agents base their individual actions on a shared state space. At each time step, each controller builds the virtual graph representing the managed subnet, encodes it in a low-size array, and shares the obtained embedding with other peers. This small-sized array enables us to reduce the communication overhead while still effectively representing each subnet feature. Indeed, each agent can eventually concatenate all arrays and build the shared input of the DRL models. Additionally, they communicate their local action to other peers, therefore unlocking

global observation over actions and state through communication. The exchange of this information happens during both the training and inference phases.

Finally, once they collect a batch of data, they also periodically exchange compressed information on the local model (see Section V-C) to reach consensus in finding the joint policy that maximizes the globally averaged long-term return defined as:

$$\max_{\theta} J(\theta) = \max_{\theta} \left(\lim_{T} \frac{1}{T} \mathbb{E} \left(\sum_{t=0}^{T-1} \frac{1}{N} \sum_{i \in N} r_{t+1}^{i} \right) \right), \quad (19)$$

where r^i represents the return, *i.e.,* reward, of agent i within the time step. Since the policy maximizes the globally averaged long-term return, which is not directly available, each agent must estimate the global reward produced by the joint actions, as it is needed to train the policy.

B. Global Reward Estimation

Since the local r_{t+1}^i of other agents is not sampled, each one estimates the average reward function $\bar{R}(s,a)$ in a collaborative way. Although many function approximators exist in the literature [44], [45], [46], with this paper, we propose to use feed-forward neural networks as estimators, as they are universal function approximations (as demonstrated in [47]) that can easily scale to a large amount of data. Each agent i maintains a multi-layered feed-forward neural network as an estimator, whose set of weights is denoted by the parameter λ_t^i at time t. This local estimator is fed at each training step with a concatenation of state s and actions a as input (reconstructed through communication) and outputs the current estimated globally-averaged reward, later used to update the policy. The joint objective is to improve the accuracy of the estimation by reducing the following error:

$$[\bar{R}(s,a;\lambda) - \bar{R}(s,a)]^2, \quad (20)$$

varying the parameter λ, where $\bar{R}(s,a) = \sum_{i \in N} \frac{1}{N} \cdot R^i(s,a)$, and $R^i(s,a)$ is the reward function for agent i, *i.e.,* Eq. (4). To reach this goal, once a batch of data is collected, each agent computes the set of weights that reduce the dissimilarity with their own reward in the following way:

$$\tilde{\lambda}_t^i = \lambda_t^i + \beta_\lambda \cdot [r_{t+1}^i - \bar{R}_t(\lambda_t^i)] \cdot \nabla_\lambda \bar{R}_t(\lambda_t^i), \quad (21)$$

where β_λ is the learning rate. Finally, agents share their estimator neural network parameters with their neighbors in the consensus step and update their estimator:

$$\lambda_{t+1}^i = \sum_{j \in N} cn_t(i,j) \cdot \tilde{\lambda}_t^j, \quad (22)$$

with $cn_t(i,j) = d_t$ representing the weight of the message at time t between i and j if they're connected, otherwise 0. This allows for an estimate of the average reward.

C. Consensus-Based Gradients Update

The actor, critic, and estimator neural networks are parametrized by θ, v, λ, respectively. We define π_θ, $V(\cdot;v)$, and $\overline{R}(\cdot;\lambda)$ as our parameterized policy, state-value, and average reward function. We also keep track of the estimate of the average long-term return with a scalar value μ. As in traditional AC, our algorithm consists of two phases: the critic step and the actor step, in which the corresponding neural networks are updated following the policy gradient principle. In addition to these, we introduce a *consensus* step, in which agents exchange their local parameters $\tilde{\lambda}, \tilde{\mu}, \tilde{v}$ to train the optimal joint policy.

Inspired by Advantage Actor Critic (A2C), we first define the gradient of the globally long-term average return for a multi-agent setting as follows:

$$\nabla_{\theta^i} J(\theta) = \mathbb{E}_{s,a} \nabla_{\theta^i} log \pi_{\theta^i}^i(s,a^i) A_\theta^i(s,a) \quad (23)$$

The advantage function is typically introduced to reduce variance and increase stability in updates. However, it requires computing two functions (the state-action value and the state-value functions). We then estimate the Advantage functionusing the state-value TD errorr, which represents the difference between the current estimate of the value function and the estimate after the transition from one state to another. For any $s \in S, a \in A$:

$$A_\theta(s,a) = \mathbb{E}\left[\bar{r}_{t+1} - J(\theta) + V_\theta(s_{t+1}) - V_\theta(s_t) | s_t = s, a_t = a, \pi_\theta\right] \quad (24)$$

In the critic step, the local reward r_{t+1}^i is initially used to compute the state-value TD-error:

$$\delta_t^i = r_{t+1}^i - \mu_t^i + V_{t+1}(v_t^i) - V_t(v_t^i) \quad (25)$$

which affects the value function parameters to generate greater values for better-rewarded states. The critic parameters are then computed as follows:

$$\tilde{v}_t^i = v_t^i + \beta_v \cdot \delta_t^i \cdot \nabla_v V_t(v_t^i) \quad (26)$$

where β_v represents the critic learning rate. The local parameters μ and λ (Eq. (21)) are determined and shared in the critic step as well:

$$\tilde{\mu}_t^i = (1 - \beta_v) \cdot \mu_t^i + \beta_v \cdot r_{t+1}^i \quad (27)$$

In the Actor step, the policy network parameters are updated. Following the previous assumption of estimating the Advantage function through TD-error, the policy update is computed in this manner:

$$\theta_{t+1}^i = \theta_t^i + \beta_\theta \cdot \tilde{\delta}_t^i \cdot \nabla_{\theta^i} log \pi_{\theta_t^i}^i(s_t, a_t^i) \quad (28)$$

where $\nabla_\theta \log \pi_\theta(s,a)$ is the *score function* for policy π used to compute a sampled measure of the gradient of the expected return with respect to its parameters, β_θ is the actor learning rate and $\tilde{\delta}_t^i$ is the globally-averaged TD-error. In fact, we cannot reuse the state-value TD-error for the policy update, because it samples the locally defined reward. The globally averaged TD-error is used to encourage joint actions that transition to better states and are rewarded more than the average. It is evaluated in the following way:

$$\tilde{\delta}_t^i = \bar{R}_t(\lambda_t^i) - \mu_t^i + V_{t+1}(v_t^i) - V_t(v_t^i) \quad (29)$$

where $\bar{R}_t(\lambda_t^i)$ represents the output of the reward estimator at time step t.

Finally, once both actor and critic steps are terminated, the consensus step is performed over the shared parameters $\tilde{\lambda}, \tilde{\mu}, \tilde{v}$:

$$\mu_{t+1}^i = \sum_{j \in N} cn_t(i,j) \cdot \tilde{\mu}_t^j \tag{30}$$

$$v_{t+1}^i = \sum_{j \in N} cn_t(i,j) \cdot \tilde{v}_t^j \tag{31}$$

In the consensus step, all agents wait to receive updates from every active agent before proceeding to the next time step$t$. This phase occurs only during training, so while waiting for all updates is not critical, it may prolong the overall training time.

The consensus step enables building a decentralized critic and reward estimator, which are fundamental to finding the optimal joint policy. Obtaining such a decentralized critic model and reward estimator requires exchanging a scalar value $\tilde{\mu}$ and the weights of two deep neural networks for each agent. This process, though it occurs only during training, demands significant bandwidth, as agents need to exchange large amounts of data frequently. To mitigate this communication overhead, we propose pruning neural network weights prior to transmission. Specifically, the agents identify and remove the weights with the lowest L1-norm, as they contribute the least to the output, and transmit the remaining weights with a bitmask so the receiver can reconstruct their positions. This reduces the size of the data to be transferred, minimizing also the required bandwidth. Importantly, pruning is applied only to the weights shared between agents, while each agent's local model remains unaffected, thereby having minimal impact on performance, as locally learned information is preserved. To limit the loss in the contribution from other agents, we impose that the agents preserve the first layer of the neural networks, as it conveys critical information [48], while pruning the subsequent ones. Additionally, since early pruning during training leads to significant performance degradation [49], it is applied after 5 epochs, *i.e.,* communication rounds. We apply Local Unstructured Pruning (LUP) [50] to each layer, excluding the first one, and adopt a dynamic rate: each agent initially prunes 30% of the weights of the selected layers, but linearly increases the rate over training epochs up to 70%. Once training stops, all obtained neural networks are used in inference mode without further weight exchange.

VI. Evaluation

In this section, we assess the performance and robustness of our solution in our own emulator and over a large-scale virtual network testbed. We compare EAGLE to both non-learning-based and DRL-based autoscaling solutions. To assess the generalization, we provide a visual representation of the graph embedding result. Then, we demonstrate the resiliency of our system over both data plane and control plane failures. Lastly, we measure the overhead of our solution.

A. Settings

1) Training Setup: We implement the data plane and control plane over Mininet [19], a well-known network emulator, to create realistic networks. We use OVS switches featuring OpenFlow for the data plane and Ryu [51] for the controllers. The network used during experiments is divided into three subnets by our community detection algorithm, with each managed by a controller that interacts with the others for collaborative learning. Each agent's Actor, Critic, and Reward estimator are multilayer perceptrons with shared hidden layers of sizes 256, 128, and 64, differing only in their input dimensions and final output layer. A pre-training phase is conducted to build the Reward estimator, which is later used only in inference mode. Learning rates for Actor, Critic, and Reward estimator are $\beta_\theta = 0.0001$, $\beta_v = 0.001$, and $\beta_\lambda = 0.001$, respectively. The weight parameter α is set to 0.5 and the time step τ to 5s. This time step was chosen empirically as a good trade-off since a high frequency can introduce substantial overhead and excessively frequent adjustments, leading to synchronization issues, while an excessively low frequency can hinder the network's ability to address issues promptly. The partitioning variables for the training network are $\rho = 0.01$, $\epsilon = 2$, and $\gamma = 6$.[1]

During training, hosts replay real-world packet traces, producing localized hotspots that can temporarily exceed link capacities. Agents initially explore the action space using random or partially randomized scaling actions. When switches are activated at appropriate times-i.e., as congestion begins to emerge-controllers update routing decisions in Mininet to leverage newly available paths created by additional switches and links. This leads to improved throughput, which is reflected in a higher reward. Conversely, deactivating unnecessary resources is incentivized by the reward function, which penalizes energy consumption and encourages more efficient configurations. Each agent periodically computes gradients and updates its local network parameters to maximize long-term cumulative reward, thereby learning the control policy. For our training phase based on curriculum learning, we established 5 Training Environments (TrEnvs) with incremental difficulty. Prior work utilizing RL models, e.g., [43], has shown the promise of curriculum learning, including faster convergence, higher asymptotic performance, and better generalization. Since our solution needs to respond to congestion while simultaneously deallocating unnecessary resources, we included congestion conditions and periods of underutilization in the domains to learn. We manually set up these TrEnv for our training, whereby TrEnv 1 denotes no network congestion, TrEnv 5 represents congestion on all three subnets, and all TrEnv in between denote congestion on specific subnets. We replicate traffic via traces publicly available on Netresec [52], which offers a wide list of traffic packet captures with non-regular patterns. In particular, we use 5GB TCP connections by Digital Corpora and the Megalodon Challenge trace.

2) Benchmarks: We compared the performance of our solution against five benchmarks: *(i) DeepConf*, a related DRL-based solution to automate data center networks [12], *(ii) SmartFCT*, a flow consolidation schema [10], *(iii) Mystique* an alternative MARL network management schema [13], *(iv) CollaQ*, a general collaborative MARL algorithm

[1]The code is available at https://github.com/dahara98/EAGLE

[53], *(v) CATE*, a recent heuristic aiming for carbon-aware routing [22].

DeepConf outputs link activation states based on the network topology and traffic matrix received at each step. Its objective is to maximize the amount of data transferred while minimizing flow duration. *SmartFCT* analyzes the distribution of flows across multiple links and outputs a flow ratio. The objective is defined through a reward function that penalizes the number of shut-down links while also considering violations of flow completion time (FCT). Despite having similar goals, we employ a multi-agent approach rather than a centralized single agent.

Mystique deploys multiple agents that operate independently. Their input consists of link utilization, and they output link activation to optimize quality of experience (QoE), fairness, and network cost management. In addition to the distinct reward design, we differentiate by utilizing collaborative agents. Moreover, our solution abstracts the network status through an embedding whose scale is independent of specific network characteristics. *CollaQ* algorithm deploys two Deep Q-networks (DQN), where one is used to evaluate the contribution of agents singularly, and the other evaluates the collective impact on the Q-value function. These two terms are then combined to compute an average agent contribution during training updates. Since it is an RL algorithm rather than a network scaling solution, we adopt it on top of our framework to compare our collaborative algorithm against it. *CATE* heuristic ranks links based on their carbon cost and verifies if a viable routing solution exists after excluding specific links at each step to deactivate them. The training time varies across frameworks, amounting to 341 minutes for EAGLE, 110 minutes for CollaQ, 112 minutes for DeepConf, 68 minutes for Mystique, and 71 minutes for SmartFCT.

Similar to [54], which studies the generability of RL models, we design experiments in the evaluation to test how EAGLE can be transferred across different topologies and network status distributions. We summarize in Table III the settings that determine the *source domain* used for training, as well as four target topologies and four target traffic conditions. The rationale behind this choice is detailed later in Section VI-C.

All experiments were repeated 10 times for each topology, traffic trace, and controller configuration, and results are reported using 90% confidence intervals.

B. Evaluation Metrics

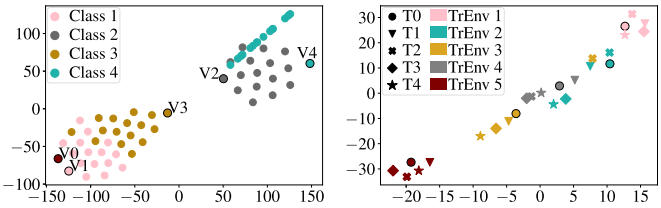
While the reward function is typically employed to assess the performance of RL models, the different solutions used in our comparison use different reward functions. As such, for a fair comparison, we base our evaluation on meaningful metrics such as Flow Completion Time (FCT), a key performance metric for network congestion [55], and Power Saving (PS). For clarity, in what follows, we report the FCT gain and PS compared to a network management schema that does not employ any reaction or predictive logic (*baseline*), using these equations:

$$FCTgain = 1 - \frac{\text{FCT of solution}}{\text{FCT baseline}} \tag{32}$$

TABLE III
GENERALIZATION EXPERIMENTS SETUP

Network	#Nodes	#Links	Key Differences
Source $V0$	13	16	
$V1$	13	16	Random network with $V0$ node degrees
$V2$	13	18	Random network with more links
$V3$	16	22	New nodes and links expand $V0$
$V4$	149	386	Large-scale networks

Setting	Source	Target traffic			
	$T0$	$T1$	$T2$	$T3$	$T4$
Traffic capture	Corpora	Corp.	Megalodon	Corp.	Corp.
Network utilization	default	↓	-	↑	↑↑



(a) Different Topologies (b) Different Traffic Conditions

Fig. 4. Visual representation of MARL input. Similar networks and traffic patterns result in embeddings close in the space.

$$PS = 1 - \frac{\text{power consumed by solution}}{\text{power consumed baseline}} \tag{33}$$

To summarize these two properties, we introduce an *efficiency score* formulating the objective to minimize the flow completion time while using the minimum amount of necessary resources:

$$\text{Efficiency score} = \frac{\text{FCTgain norm.}}{1 - \text{PS}}, \tag{34}$$

where to simplify and make this formula more practical and easy to understand, we normalized the FCT gain w.r.t. the maximum value measured among all the solutions. Among solutions with the same FCT gain, those achieving the same performance with fewer active resources obtain a higher score. This is a relative efficiency indicator that should be interpreted together with FCT gain and power saving.

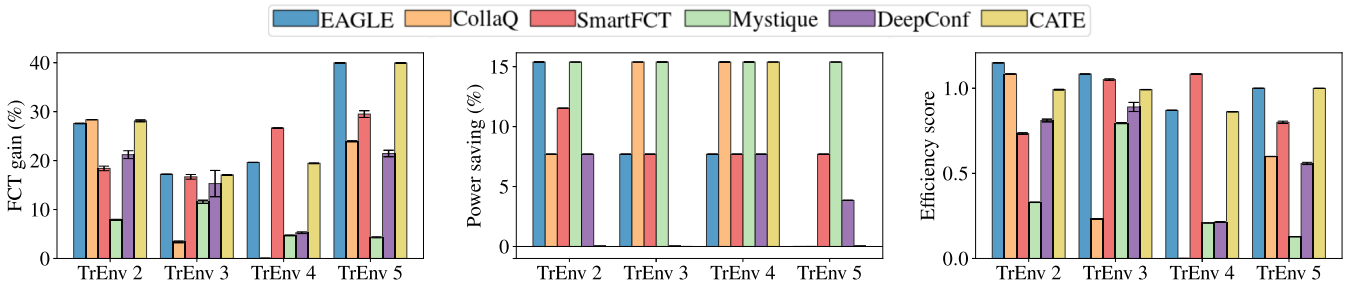
For an effective comparison against *CATE* algorithm, we measure the carbon emissions of our solution as in [22]:

$$C_{tot} = \sum_{v}^{V}(rate_v \cdot P_{dyn_v} \cdot ci_v) + \sum_{l=(u,v)}^{E} \omega(ci_u + ci_v) \tag{35}$$

where ci is the carbon intensity and represents the amount of carbon released for each unit of energy consumed, ω is the incremental power per active port.

C. Graph Embedding Impact

Since the state distribution is the key for RL generalization (a trained model acts similarly over similarly distributed data), we now study the impact in the state representation when varying network settings, such as network size or traffic pattern. We provide a visual representation of the neural network inputs in Fig. 4a as a simple and effective way to visualize that similar-structured networks produce similar embeddings.



(a) FCT Gain w.r.t. traditional networks (b) Power Saving w.r.t. traditional networks (c) Efficiency score comparison

Fig. 5. Our algorithm has the best performance w.r.t. FCT and energy saving across different network conditions.

After the subnets are embedded with Feather Graph with parameters $g = 1$ and $e = 1$ (see Sec. IV-C), the embeddings are concatenated into one array, which is further reduced to a 2-dimensional space with t-SNE [56], a non-linear algorithm that plots objects in a 2D plane based on their similarity.[2]

Starting from the original topology (Source $V0$), the new set of networks is characterized by the following: *(i) Class 1,* randomly generated networks with the same degree distribution, *(ii) Class 2,* randomly generated network with the same number of nodes but additional links, *(iii) Class 3,* expanded netowrks with additional nodes and links, *(iv) Class 4*, large-scale networks. The target topologies from each Class used in the experiments are represented with bordered markers. It is evident that the degree distribution is the most impactful characteristic for embedding similarity: the topologies closest to the source topology have the same degree distribution. As we look farther away, the degree distribution is partially different (in the case of expanded networks) and then totally different for bigger networks. For closer networks, the relative distance is given by a different weight distribution, as the weight of links is fundamental to computing the Feather Graph embedding.

The same technique has been used to produce Fig. 4b, where we compare the embedding of different traffic patterns over $V0$. The traffic observed during training ($T0$) is derived from the Corpora trace. During the capture, network utilization fluctuates over time, so we extrapolate different time intervals of the same trace to create unseen settings. In $T1$, the traffic intensity is lower, while it increases in $T3$ and reaches a heavy load in $T4$. Additionally, we use a different traffic capture for $T2$. Each colored group represented in the picture includes the embedding produced by the source (bordered points) and unseen traffic in established TrEnvs. Note that TrEnv 1 represents a uniformly distributed, uncongested traffic scenario, while TrEnv 5 induces congestion across all three subnets. Intermediate environments (TrEnv 2–TrEnv 4) generate hot spots only in selected subnets. The figure shows a clear trend: the produced embedding varies as network throughput increases. TrEnv 1 and TrEnv 5 are far apart, with intermediate states in the middle. The similarity of embeddings for similar conditions is an important result, since our model can detect congestion in the input and apply similar actions in such occurrences. Given this outcome, we select as target domains four topologies at different areas of Fig. 4a, and four traffic patterns along the main line of Fig. 4b to have a variance of conditions during our tests.

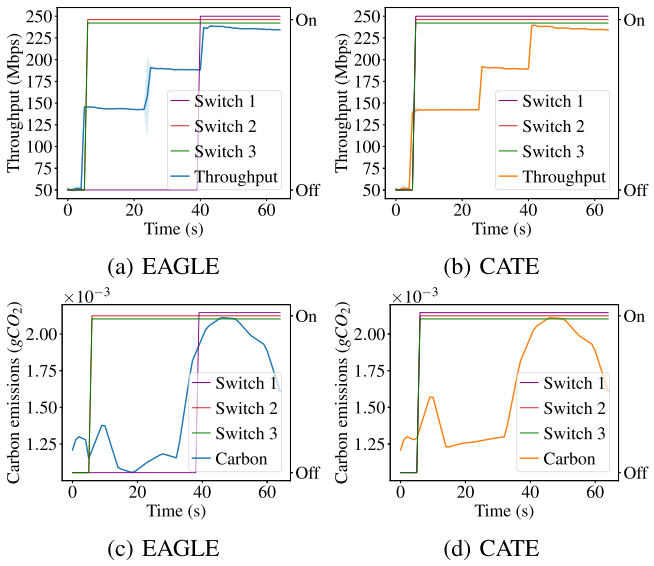
[2]Note that embedding plots miss axis labels because they don't directly represent a specific feature of the original data.

D. Performance Over Different Network Conditions

In Fig. 5, we report the performance when running different autoscaling schemes on the source topology $V0$ and source traffic $T0$, *i.e.,* the topology and traffic patterns used to train the RL algorithms. In particular, we can observe the performance in terms of (a) FCT gain, (b) power saving, and (c) efficiency score of the trained algorithms over the same conditions (TrEnvs) used during the curriculum learning procedure. While from Fig. 5a, we see that SmartFCT can tackle congestion, it does not properly reduce power consumption (Fig. 5b and Fig. 5c). Similarly, solutions like Mystique can efficiently save more power (Fig. 5b), but at the cost of higher FCT (Fig. 5a and Fig. 5c). Solutions employing a centralized controller, such as DeepConf and SmartFCT, exhibit quicker decision-making times because the global information is directly available. However, the switch-to-controller latency escalates on large-scale networks. On the other hand, our solution reduces latency in collecting metrics and injecting rules while also limiting communication overhead through the exchange of compressed or pruned information. Despite this, EAGLE consistently performs well across all conditions (TrEnvs), providing a good trade-off between network performance and energy savings, yielding at least a score of 0.87 in its worst case.

The graph embedding's role in capturing and representing the intricate relationships within the network is crucial for the algorithm's robust performance across scenarios, fostering a more comprehensive understanding of the dynamic structure. In addition, these results validate our lightweight consensus updates against individual decision-making (e.g., Mystique) and alternative collaborative MARL approaches (e.g., Collaq), which produce markedly lower scores across all conditions.

The carbon-aware heuristic shows satisfactory performance in terms of throughput, but sometimes struggles to turn off unnecessary resources due to its greedy approach. A detailed comparison is reported in Fig. 6, where the carbon intensity of nodes corresponds to regional data acquired by [57]. As shown in Fig. 6(a-b), EAGLE can produce the same throughput as



(a) EAGLE (b) CATE

(c) EAGLE (d) CATE

Fig. 6. (a-b) Evolution of throughput (left side) and decisions for three monitored switches (right side). (c-d) Evolution of carbon emissions (left side) and decisions (right side).

CATE, but with switch 1 off, along with associated links, in the first 40 seconds. The observed deviation occurs because CATE extracts links from a cost-based sorted list and, while scanning them iteratively, tests whether they can be removed. This greedy approach is fast but can lead to suboptimalities. Consequently, it can compromise its overall effectiveness and lead to higher carbon emissions over the same period compared to our approach (Fig. 6(c-d)).

E. Generalization Over Unseen Topologies and Traffic

We now test the generalization of EAGLE and compare it against a subset of the benchmarks. Specifically, since we are evaluating the generalization capabilities of machine learning models, we do not assess CATE, as it is a heuristic method, and generalization does not apply to it. Given that SmartFCT and DeepConf are both learning-based, single-agent approaches, we choose to compare EAGLE with SmartFCT, which performed better in the initial set of experiments (Fig. 5c). For these experiments, we use the eight target domains mentioned in Table III. The target topologies and traffic patterns, although unseen during training, span a range of distributional distances from the source domain. Topologies $v1 - v3$ are generated by controlled modifications of the source topology, while $v4$ (GtsCe) is a structurally distinct real-world large-scale network unrelated to the source topology. Fig. 7 shows the efficiency score when mutating the network topologies. Since some of the algorithms rely on network-specific features for the state input size, we need to re-train these models when needed and mark them with hatched bars in the graphs. We can observe that EAGLE provides a consistent score above 1 in Fig. 7a and Fig. 7c, which represent the most similar topologies to the source one, as shown in Fig 4a. As for topologies $V2$ and $V4$, shown in Fig. 7b and Fig. 7d, our algorithm reports performance that is mostly comparable to the CollaQ results, which, however, were specifically retrained on these topologies. For EAGLE we considered a *zero-shot generalization*, as re-training a model for each new environment would be both time and power-consuming. To further study this aspect, we report in Fig. 9 the reward evolution over episodes in topology $V2$, where dotted lines denote the training phase and solid lines the testing episodes. We measured all algorithms using the same reward function as in EAGLE to ensure comparable values, and we plot the cumulative reward normalized by $(r - r_{min})/(r_{max} - r_{min})$. These models are not the ones used for performance experiments, where original reward functions are deployed. This graph clearly shows how our solution can zero-shot generalize to unseen domains without retraining, saving time and resources.

Fig. 8 shows the same set of experiments produced over different traffic patterns reproduced in the source topology. We observe that EAGLE performs best in most experiments and is the only framework whose score never drops below 0.9, maintaining stable performance across a variety of unseen scenarios.

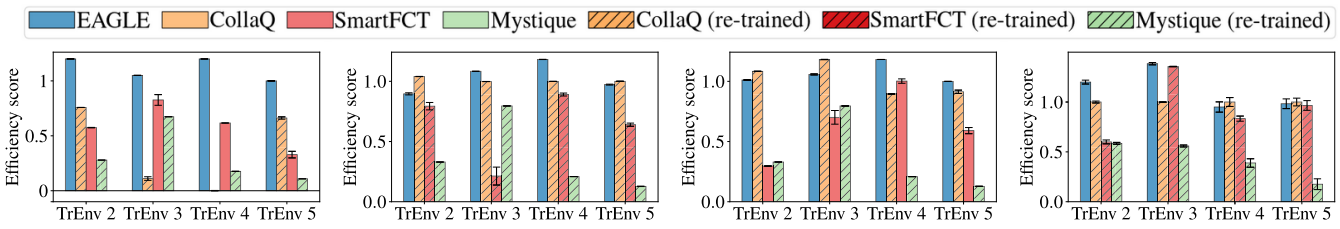
F. Links and Agents Failure Reaction

In this set of experiments, we test EAGLE robustness over link failure. The tests were performed by increasing the number of link failures over the same emulated traffic and network. Fig. 10a shows the reward distribution when increasing the number of failed links. We can see that the more links fail, the average reward (orange line) is maintained almost constant, with only a little variation in the data variance (ticks). A similar outcome is observed for packet delay (Fig. 10b), where our re-routing strategy can react immediately to failures, thus reducing their impact on performance.

The second set of experiments tests the robustness of the framework over agent failure. When a controller fails, the other controllers are in charge of handling the switches previously assigned to the failing agent. This functionality of EAGLE allows continuous monitoring and management of the network. However, to enable this feature, our MARL model, designed as a network of time-varying agents, must also respond to failures during the consensus phase. Results of the experiment are presented in Fig. 10c. We compare the efficiency score for this scenario with that of the optimal scenario with no failures. The lower scores on some tests are due to reaction time and model alignment. However, such a score reduction is limited, and EAGLE shows resilience, maintaining a score above 0.9.

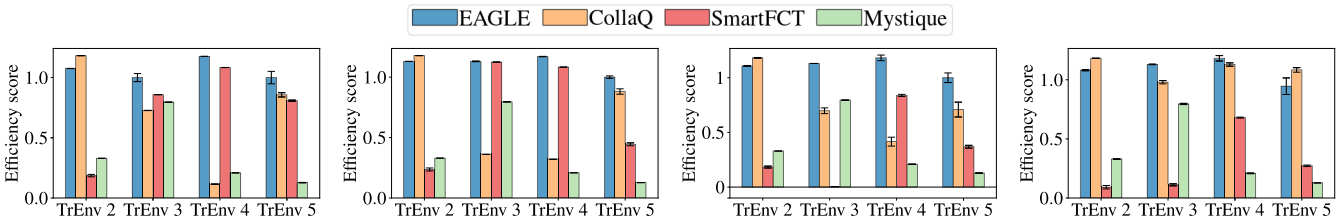
G. Experiments Over the Fabric Testbed

To demonstrate the viability of our approach and assess its performance over real-world networks, we test EAGLE over the FABRIC testbed [20]. We first measure the efficiency score of our model over an unseen network topology (NSFNET), comparing the two testbeds (Fig. 11a). Across all the experiments, EAGLE maintains a stable performance in both cases, with a score always above 0.9. The slight differences in some tests are due to the background traffic that is present in FABRIC since physical links are shared among users. Despite this unpredictable link utilization influence, the comparable scores suggest that (1) results on Mininet are realistic and



(a) Topology $v1$ (b) Topology $v2$ (c) Topology $v3$ (d) Topology $v4$

Fig. 7. Efficiency score over unseen topologies. Our model outperforms other algorithms in generalizing over new topologies in multiple network conditions. It also achieves similar performances w.r.t. re-trained models (hatched bar).



(a) Traffic pattern $t1$ (b) Traffic pattern $t2$ (c) Traffic pattern $t3$ (d) Traffic pattern $t4$

Fig. 8. Efficiency score over unseen traffic settings. Our model has the best performance in generalizing over unseen traffic in different network conditions.

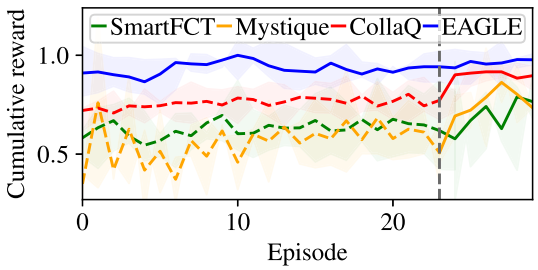


Fig. 9. Cumulative reward during training (dashed line) and testing (solid line) over topology $v2$. Our model achieves the highest reward even without the need to be re-trained, showing *zero-shot generalization*.

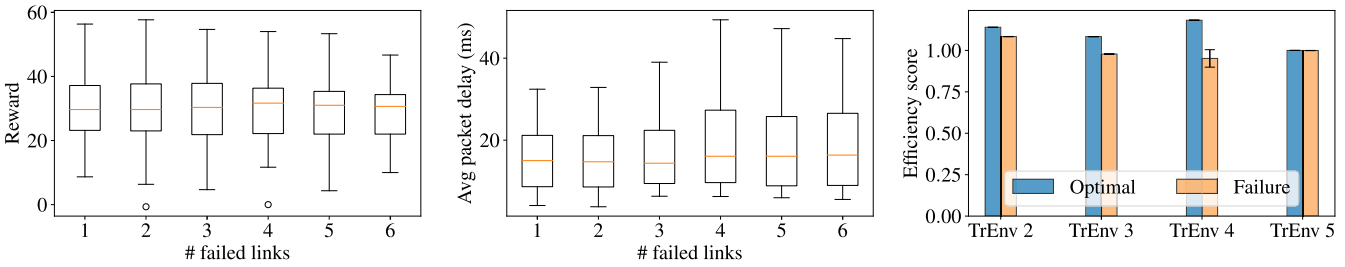
valid for drawing conclusions, and (2) the model learned over Mininet can generalize even to diverse testbeds. Since such a network used for these experiments represents a new domain, we then test the generalization capabilities of our approach. Fig. 11b shows the efficiency score comparison among state-of-the-art solutions and EAGLE in FABRIC, where we need to re-train all other models given the different-sized topology. EAGLE has a consistent score close to or above 1.0, although this topology and traffic data were never explored during training. Overall, we can state that EAGLE can zero-shot generalize and reach a similar or better performance than other algorithms specifically trained on that environment.

H. System Overhead

We now evaluate the overhead of our solution in terms of action latency. Measurements are repeated 25 times. At each step of our control loop, our system first performs inference to generate a decision. This decision is then communicated to the controller, which updates the network accordingly. If the configuration remains unchanged from the previous step, no additional overhead is incurred. However, if there are changes, certain interfaces and OVS switches may be enabled or disabled, and new routing rules may be installed. As shown in Fig. 12, the inference phase exhibits an average latency of approximately 100 ms, with values ranging from roughly 10 ms to peaks around 220 ms, based on input complexity. The overhead associated with interface management is negligible, as both activation and deactivation times for interfaces remain very low. In contrast, reconfiguring OVS introduces a higher and asymmetric cost. While the average shutdown time for OVS operations is around 50 ms, the activation process takes notably longer, with an average latency of close to 200 ms. Regarding routing, we measure the time required by the controller to push the OF message through the network, excluding subsequent controller-to-switch propagation, and the average is approximately 180 ms. Overall, these results indicate that the system introduces negligible overhead when no changes are made. However, the costs associated with reconfiguration are primarily driven by OVS updates and routing rule installation. Despite these factors, the total latency remains under one second, confirming the practical feasibility of this approach for near-real-time control scenarios. Note that the routing latency reported here does not account for controller-to-switch propagation delay, which is expected to add a modest additional cost depending on the underlying network topology and controller placement.

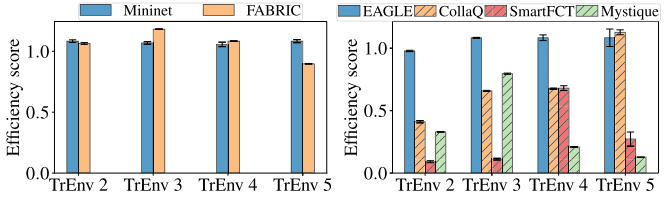
I. Sensitivity Analysis

We study the impact of key hyperparameters, such as the path length g and the number of evaluation points e, on embedding quality (see Section IV-C). A greater g leads to a higher order of neighborhoods for each node of the



(a) Reward distribution (b) Latency distribution (c) Agent-failure vs. optimal efficiency score

Fig. 10. Our MARL framework is robust to network nodes and agent failures. Even in challenging scenarios, the model maintains a stable performance.



(a) FABRIC vs. Mininet (b) Benchmark tests in FABRIC

Fig. 11. Production results (FABRIC) are comparable to Mininet emulation. Our zero-shot model achieves comparable performance w.r.t. re-trained (hatched bar) baselines.

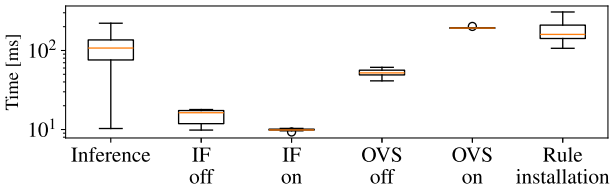
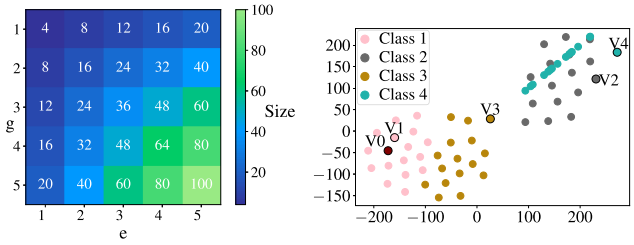


Fig. 12. The action latency is given by inference when no changes are made, while reconfiguration costs mainly arise from OVS updates and routing rule installations.



(a) Parameters effect on state size (b) Embedding with g=2,e=1

Fig. 13. Sensitivity analysis over FeatherGraph parameters.

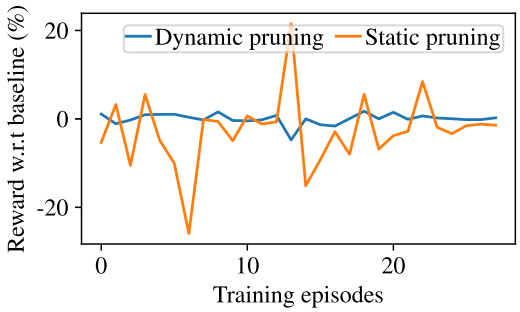


Fig. 14. Dynamic pruning reduces the communication overhead with limited loss of performance.

embedding, while increasing e allows the capture of more fine-grained information, so higher values generally enhance the capacity to gather more information; however, it also carries the risk of overfitting and has an effect on the embedding size, as reported in Fig. 13a, which shows the size at varying g and e. This size has an impact on the training time of the RL model, since the embeddings of the subnets are later concatenated and used as input for the model. As such, we seek a trade-off between accuracy and the model's input size via cross-validation and by visually representing the embedding, as we did in Fig. 4a. In particular, we empirically study the effect on the embedding at increasing g and e values. As shown in Fig. 13b, setting $g = 2$ and $e = 1$ yields local embeddings of size 8 and a final input of size 24. Recalling how Fig. 4a was produced with $g = 1$ and $e = 1$, we can observe little changes, leading to the consequence that increasing the values beyond a certain threshold does not consistently yield improved results or provide additional information. In fact, we experimentally observed that as the size grows, an increasing number of cells do not convey information because their values don't change across TrEnvs. In conclusion, we set $g = 1$ and $e = 1$, producing a final embedding with a fixed size of 12.

Finally, we show the impact of deploying our dynamic pruning technique in this collaborative protocol against a static pruning rate. Fig. 14 shows the reward collected during training by the two approaches w.r.t. the training without pruning, used as the baseline. The static approach prunes 70% of the neural network's weights, while the dynamic approach starts at 30% and linearly increases to 70%. While the static method shows heavily degraded performance at the beginning, with only occasional gains over time, the dynamic approach maintains a reward very close to the baseline, resulting in reduced overhead without consistent performance loss.

VII. Conclusion

In this paper, we presented EAGLE, a solution that allows autonomously managing softwarized and virtualized networks using a multi-agent reinforcement learning approach. Multiple agents can control the subnet while sharing network information and model knowledge with one another to create a more reactive and intelligent management plane. Our proposed model can also generalize to other network domains, *i.e.,* scenarios differing by the number of links and traffic patterns, given the state space design and the training procedure. Our results demonstrated the validity of this approach,

confirming how EAGLE can jointly minimize the FCT and reduce power consumption in a variety of contexts (including large and real-world networks) without the need to be re-trained. Nevertheless, some limitations remain. In particular, the effectiveness of the proposed approach relies on the availability of accurate and timely network telemetry. In practical deployments, imperfect observability or delayed measurements may lead to suboptimal or unstable decisions. Moreover, while generalization results are encouraging, further evaluation on a broader set of structurally diverse networks would help to thoroughly characterize the limits of generalization.

REFERENCES

[1] M. E. Kanakis, R. Khalili, and L. Wang, "Machine learning for computer systems and networking: A survey," *ACM Comput. Surveys*, vol. 55, no. 4, pp. 1–36, Apr. 2023.

[2] D. Monaco, A. Sacco, and G. Marchetto, "CROWN: Cross-attention reinforcement learning for O-RAN wireless networks," *Comput. Netw.*, vol. 282, Jun. 2026, Art. no. 112276.

[3] M. N. A. H. Khan, Y. Liu, H. Alipour, and S. Singh, "Modeling the autoscaling operations in cloud with time series data," in *Proc. IEEE 34th Symp. Reliable Distrib. Syst. Workshop (SRDSW)*, Sep. 2015, pp. 7–12.

[4] S. Bhagavathiperumal and M. Goyal, "Workload analysis of cloud resources using time series and machine learning prediction," in *Proc. IEEE Asia–Pacific Conf. Comput. Sci. Data Eng. (CSDE)*, Dec. 2019, pp. 1–8.

[5] V. Tadakamalla and D. A. Menasce, "Model-driven elasticity control for multi-server queues under traffic surges in cloud environments," in *Proc. IEEE Int. Conf. Autonomic Comput. (ICAC)*, Sep. 2018, pp. 157–162.

[6] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Trans. Auto. Adapt. Syst.*, vol. 3, no. 1, pp. 1–39, Mar. 2008.

[7] A. Ashraf, B. Byholm, and I. Porres, "CRAMP: Cost-efficient resource allocation for multiple web applications with proactive scaling," in *4th IEEE Int. Conf. Cloud Comput. Technol. Sci. Proc.*, Dec. 2012, pp. 581–586.

[8] S. Gong, X. Zhu, R. Zhang, H. Zhao, and C. Guo, "An intelligent resource management solution for hospital information system based on cloud computing platform," *IEEE Trans. Rel.*, vol. 72, no. 1, pp. 329–342, Mar. 2023.

[9] Y. G. Kim and C.-J. Wu, "AutoScale: Energy efficiency optimization for stochastic edge inference using reinforcement learning," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 1082–1096.

[10] P. Sun, Z. Guo, S. Liu, J. Lan, J. Wang, and Y. Hu, "SmartFCT: Improving power-efficiency for data center networks with deep reinforcement learning," *Comput. Netw.*, vol. 179, Oct. 2020, Art. no. 107255.

[11] L. Pappone, A. Sacco, and F. Esposito, "Mutant: Learning congestion control from existing protocols via online reinforcement learning," in *Proc. 22nd USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2025, pp. 1507–1522.

[12] S. Salman, C. Streiffer, H. Chen, T. Benson, and A. Kadav, "DeepConf: Automating data center network topologies management with machine learning," in *Proc. Workshop Netw. Meets AI ML (NetAI)*, 2018, pp. 8–14.

[13] A. Sacco, M. Flocco, F. Esposito, and G. Marchetto, "Supporting sustainable virtual network mutations with mystique," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 3, pp. 2714–2727, Sep. 2021.

[14] D. Monaco, A. Sacco, E. Alberti, G. Marchetto, and F. Esposito, "A collaborative and distributed learning-based solution to autonomously plan computer networks," in *Proc. 26th Conf. Innov. Clouds, Internet Netw. Workshops (ICIN)*, Monaco, Mar. 2023, pp. 1–5.

[15] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 473–478, 2012.

[16] K. Zhou, Z. Liu, Y. Qiao, T. Xiang, and C. C. Loy, "Domain generalization: A survey," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 4, pp. 4396–4415, Apr. 2023.

[17] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, "Quantifying generalization in reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, May 2019, pp. 1282–1289.

[18] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *J. Stat. Mechanics, Theory Exp.*, vol. 2008, no. 10, Oct. 2008, Art. no. P10008.

[19] R. L. S. de Oliveira, C. M. Schweitzer, A. A. Shinoda, and L. Rodrigues Prete, "Using mininet for emulation and prototyping software-defined networks," in *Proc. IEEE Colombian Conf. Commun. Comput. (COLCOM)*, Colombia, Jun. 2014, pp. 1–6.

[20] I. Baldin et al., "FABRIC: A national-scale programmable experimental network infrastructure," *IEEE Internet Comput.*, vol. 23, no. 6, pp. 38–47, Nov. 2019.

[21] F. Abuzaid et al., "Contracting wide-area network topologies to solve flow problems quickly," in *Proc. 18th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2021, pp. 175–200.

[22] S. El-Zahr, P. Gunning, and N. Zilberman, "Exploring the benefits of carbon-aware routing," in *Proc. ACM Netw.*, vol. 1, 2023, pp. 1–24.

[23] D. Coudert, A. Kodjo, and T. K. Phan, "Robust energy-aware routing with redundancy elimination," *Comput. Operations Res.*, vol. 64, pp. 71–85, Dec. 2015.

[24] H. Arabnejad et al., "An auto-scaling cloud controller using fuzzy Q-learning-implementation in openstack," in *Proc. Service-Oriented Cloud Comput.*, 2016, pp. 152–167.

[25] A. Sacco, F. Esposito, and G. Marchetto, "RoPE: An architecture for adaptive data-driven routing prediction at the edge," *IEEE Trans. Netw. Service Manage.*, vol. 17, no. 2, pp. 986–999, Jun. 2020.

[26] V. R. Messias et al., "Combining time series prediction models using genetic algorithm to autoscaling web applications hosted in the cloud infrastructure," *Neural Comput. Appl.*, vol. 27, pp. 2383–2406, Nov. 2016.

[27] M. Cheng, J. Li, and S. Nazarian, "DRL-cloud: Deep reinforcement learning-based resource provisioning and task scheduling for cloud service providers," in *Proc. 23rd Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2018, pp. 129–134.

[28] S. M. R. Nouri, H. Li, S. Venugopal, W. Guo, M. He, and W. Tian, "Autonomic decentralized elasticity based on a reinforcement learning controller for cloud applications," *Future Gener. Comput. Syst.*, vol. 94, pp. 765–780, May 2019.

[29] N. Belhaj, D. Belaid, and H. Mukhtar, "Framework for building self-adaptive component applications based on reinforcement learning," in *Proc. IEEE Int. Conf. Services Comput. (SCC)*, Jul. 2018, pp. 17–24.

[30] D. Heo, D. Lee, H.-G. Kim, S. Park, and H. Choi, "Reinforcement learning of graph neural networks for service function chaining in computer network management," in *Proc. 23rd Asia–Pacific Netw. Operations Manage. Symp. (APNOMS)*, Sep. 2022, pp. 1–6.

[31] W. Huang, B. Yuan, S. Wang, J. Zhang, J. Li, and X. Zhang, "A generic intelligent routing method using deep reinforcement learning with graph neural networks," *IET Commun.*, vol. 16, no. 19, pp. 2343–2351, Dec. 2022.

[32] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio, "RouteNet: Leveraging graph neural networks for network modeling and optimization in SDN," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 10, pp. 2260–2270, Oct. 2020.

[33] B. Rozemberczki and R. Sarkar, "Characteristic functions on graphs: Birds of a feather, from statistical descriptors to parametric models," in *Proc. 29th ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2020, pp. 1325–1334.

[34] W. Wei et al., "GRL-PS: Graph embedding-based DRL approach for adaptive path selection," *IEEE Trans. Netw. Service Manage.*, vol. 20, no. 3, pp. 2639–2651, Sep. 2023.

[35] E. Casalicchio and L. Silvestri, "Autonomic management of cloud-based systems: The service provider perspective," in *Proc. 27th Int. Symp. Comput. Inf. Sci.*, 2013, pp. 39–47.

[36] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi, "Integrated and autonomic cloud resource scaling," in *Proc. IEEE Netw. Operations Manage. Symp.*, Apr. 2012, pp. 1327–1334.

[37] N. Feamster and J. Rexford, "Why (and how) networks should run themselves," 2017, *arXiv:1710.11583*.

[38] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Proc. 2nd USENIX Workshop Hot Topics Manage. Internet, Cloud, Enterprise Netw. Services* (Hot-ICE), 2012, p. 10.

[39] L. Guo-Hui and G. Xue, "K-center and K-median problems in graded distances," *Theor. Comput. Sci.*, vol. 207, no. 1, pp. 181–192, Oct. 1998.

[40] J. Peters and S. Schaal, "Natural actor-critic," *Neurocomputing*, vol. 71, nos. 7–9, pp. 1180–1190, Mar. 2008.

[41] W. Wang, V. W. Zheng, H. Yu, and C. Miao, "A survey of zero-shot learning: Settings, methods, and applications," *ACM Trans. Intell. Syst. Technol.*, vol. 10, no. 2, pp. 1–37, Mar. 2019.

[42] H. Cai, V. W. Zheng, and K. C.-C. Chang, "A comprehensive survey of graph embedding: Problems, techniques, and applications," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 9, pp. 1616–1637, Sep. 2018.
[43] Z. Xia, Y. Zhou, F. Y. Yan, and J. Jiang, "GeNet: Automatic curriculum generation for learning adaptation in networking," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 397–413.
[44] B. Hammer and K. Gersmann, "A note on the universal approximation capability of support vector machines," *Neural Process. Lett.*, vol. 17, pp. 43–53, Feb. 2003.
[45] E. Schulz, M. Speekenbrink, and A. Krause, "A tutorial on Gaussian process regression: Modelling, exploring, and exploiting functions," *J. Math. Psychol.*, vol. 85, pp. 1–16, Aug. 2018.
[46] J. Bouvrie and B. Hamzi, "Kernel methods for the approximation of nonlinear systems," *SIAM J. Control Optim.*, vol. 55, no. 4, pp. 2460–2492, Jan. 2017.
[47] B. C. Csáji et al., "Approximation with artificial neural networks," *Fac. Sci., Etvs Lornd Univ., Hung.*, vol. 24, no. 48, p. 7, 2001.
[48] C. Zhang, S. Bengio, and Y. Singer, "Are all layers created equal?" *J. Mach. Learn. Res.*, vol. 23, no. 67, pp. 1–28, 2022.
[49] M. Shen, P. Molchanov, H. Yin, and J. M. Alvarez, "When to prune? A policy towards early structural pruning," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2022, pp. 12247–12256.
[50] A. K. Mishra and M. Chakraborty, "Does local pruning offer task-specific models to learn effectively?" in *Proc. Student Res. Workshop Associated RANLP*, 2021, pp. 118–125.
[51] Ryu SDN Framework. Accessed: Jul. 25, 2024. [Online]. Available: https://ryu-sdn.org/
[52] *Network Forensics and Network Security Monitoring*. Accessed: Jul. 25, 2024. [Online]. Available: https://www.netresec.com/
[53] T. Zhang, "Multi-agent collaboration via reward attribution decomposition," 2020, *arXiv:2010.08531*.
[54] T. Dong et al., "Generative adversarial network-based transfer reinforcement learning for routing with prior knowledge," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 2, pp. 1673–1689, Jun. 2021.
[55] N. Dukkipati and N. McKeown, "Why flow-completion time is the right metric for congestion control," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 59–62, Jan. 2006.
[56] H. Zhou, F. Wang, and P. Tao, "T-distributed stochastic neighbor embedding method with the least information loss for macromolecular simulations," *J. Chem. Theory Comput.*, vol. 14, no. 11, pp. 5499–5510, Nov. 2018.
[57] Carbon Intensity Methodology Accessed: Apr. 15, 2024. [Online]. Available: https://www.carbonintensity.org.uk/

Doriana Monaco (Student Member, IEEE) received the M.Sc. degree in computer engineering from the Politecnico di Torino, in 2022, where she is currently pursuing the Ph.D. degree. Her research interests include computer networks monitoring and management, distributed learning applications for software-defined networks (SDN), and cloud-computing solutions.

Alessio Sacco (Member, IEEE) received the Ph.D. degree in computer engineering from the Politecnico di Torino, Italy, in 2022. He is currently an Assistant Professor with the Politecnico di Torino. His research interests include architecture and protocols for networks management and how ML/AI algorithms can be used in conjunction, implementation, and design of cloud computing applications, and new networking paradigms for AI workloads.

Flavio Esposito (Member, IEEE) received the Ph.D. degree in computer science from Boston University, USA. He is currently an Associate Professor with the Department of Computer Science, Saint Louis University, where he is also a Research Institute Fellow and a Graduate Coordinator. His research interests include edge computing, networks management, networks virtualization, cyber-physical systems, and applied artificial intelligence. His work has been supported by multiple awards, mostly from the National Science Foundation.

Guido Marchetto (Senior Member, IEEE) received the Ph.D. degree in computer engineering from the Politecnico di Torino, in 2008. He is currently a Full Professor with the Department of Control and Computer Engineering, Politecnico di Torino. His research topics cover distributed systems and formal verification of systems and protocols. His interests also include networks protocols and networks architectures.

Open Access funding provided by 'Politecnico di Torino' within the CRUI CARE Agreement