

# SlicePilot: Demystifying Network Slice Placement in Heterogeneous Cloud Infrastructures

Ioannis Panitsas<sup>\*§</sup>, Tolga O. Atalay<sup>§</sup>, Dragoslav Stojadinovic<sup>§</sup>, Angelos Stavrou<sup>†§</sup>, Leandros Tassiulas<sup>\*</sup>

<sup>\*</sup>Department of Electrical and Computer Engineering, Yale University, New Haven, CT, USA

<sup>†</sup>Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA

<sup>§</sup>A2 Labs, LLC, Arlington, VA, USA

**Abstract**—Cellular networks are comprised of software-based entities, with main functions encapsulated as Virtual Network Functions (VNFs) deployed on Commercial-off-the-Shelf (COTS) hardware. As a key enabler of 5G, network slicing offers logically isolated Quality of Service (QoS) for diverse use cases. With the transition to cloud-native infrastructures, optimizing network slice placement across multi-cloud environments remains challenging due to heterogeneous resource capabilities and varying slice-specific demands. This paper presents *SlicePilot*, a modular framework that enables autonomous and near-optimal VNF placement using a disaggregated Multi-Agent Reinforcement Learning (MARL) approach. *SlicePilot* collects real-world traffic profiles to estimate resource needs for each slice type. These estimates guide a MARL-based scheduler that minimizes deployment costs while satisfying QoS constraints. We evaluate *SlicePilot* on a multi-cloud testbed and demonstrate a 19x speed-up over combinatorial optimization methods, while keeping deployment costs within 7.8% of the optimal. Although *SlicePilot* results in 2.42x more QoS violations under high-load conditions, this trade-off is offset by faster decision-making and reduced computational overhead. Overall, *SlicePilot* delivers a scalable, cost-efficient solution for network slice placement, making it suitable for real-time deployments where responsiveness and efficiency are critical.

## I. INTRODUCTION

Next-generation cellular networks are evolving to support various use cases, each with distinct Quality of Service (QoS) requirements. To accommodate this evolving diversity, “network slicing” has emerged as a key enabler, facilitating the creation of virtually isolated network segments over a shared physical infrastructure to support different use cases like URLLC (Ultra-Reliable Low-Latency Communication), eMBB (Enhanced Mobile Broadband), and mMTC (massive Machine Type Communications) [1], [2].

The advent of network slicing is facilitated by the adoption of foundational technologies such as Network Function Virtualization (NFV) and Software Defined Networking (SDN) [3], [4]. Leveraging NFV and SDN, the proprietary Physical Network Functions (PNFs) that made up the legacy 4G Radio Access Network (RAN) and Core Network (CN), are now deployed as Virtual Network Functions (VNFs) on Commercial-off-the-Shelf (COTS) hardware in the form of finer-grained microservices. A network slice is ultimately formed by service chaining a set of VNFs to create an isolated network segment that extends from the RAN to the CN [2].

To capitalize on this software-driven approach, network operators are partnering with cloud providers to advance cloud-based 5G deployments [5], [6]. This shift enables the dynamic

instantiation and migration of network slice VNFs across a geographically distributed cloud infrastructure. By leveraging this flexibility, operators can tailor deployment strategies to accommodate the diverse Service Level Agreements (SLAs) of different use cases. Specifically, operators can strategically position latency-sensitive VNFs closer to the RAN at edge locations [7], [8], despite higher computational costs, while hosting less critical functions more cost-effectively in regional or central clouds [2], [9], [10]. Modern multi-cloud infrastructures provide a wide range of computing, networking, and storage capabilities. With infrastructure heterogeneity, coupled with diverse service requirements in terms of latency, bandwidth, and reliability, strategic VNF placement decisions become essential for optimizing resource use and supporting varied application demands in next-generation networks.

**Motivation.** The problem of network slice placement is widely recognized in the literature as a variant of the Virtual Network Embedding (VNE) problem, which is proven to be NP-hard [9]. Prior work has explored Integer Linear Programming (ILP)-based methods to achieve optimality [11]–[13], but these approaches do not scale and are computationally intensive, making them impractical for real-time decisions. Heuristic-based strategies have been proposed to improve efficiency, yet they often fail to achieve solutions close to the optimum, as they tend to get stuck in local optima [14], [15]. More recently, Reinforcement Learning (RL) has gained traction as a promising approach for optimal slice placement through trial and error procedures [16], [17]. However, existing RL solutions predominantly rely on monolithic agent implementations that struggle to capture resource and service constraints of different slice types [18]–[20], lack real-world validation [20], [21], and scale poorly in large deployments [19], [20].

To address the optimal VNF placement problem, this paper introduces **SlicePilot**, a Multi-Agent Reinforcement Learning (MARL)-based placement framework designed to minimize human intervention and enable automated, zero-touch network management. *SlicePilot* automates the deployment of network slice VNFs and disaggregated RAN components by leveraging slice-specific traffic profiles, resource demands, and infrastructure resource state. This information is then used to craft an optimal VNF placement strategy within a distributed cloud environment. Building on this foundation and practical insights, this work makes the following key contributions:

- We collect real-world traffic from our over-the-air 5G testbed, capturing diverse slice-specific patterns to construct realistic and representative traffic profiles.
- We analyze the resource demands of key user plane VNFs by replaying the captured traffic, enabling accurate resource estimation and slice placement optimization through enhancements to the open-source OpenAirInterface (OAI) 5G software stack [22].
- We design and implement a fully customized, disaggregated MARL scheduler that optimizes slice deployment in heterogeneous cloud infrastructures while jointly ensuring cost-efficiency and SLA compliance.
- We deploy *SlicePilot* in a Kubernetes-orchestrated cluster hosted on OpenStack, emulating a realistic multi-cloud infrastructure environment. Our evaluation demonstrates that *SlicePilot* delivers a 19x speed-up over traditional combinatorial optimization methods, achieves deployment costs within 7.8% of the optimal solution, and tolerates 2.42x more SLA violations, highlighting its ability to enable cost-efficient and scalable slice deployments.

The remainder of the paper is organized as follows. Section II provides background on 5G architecture and network slicing. Section III introduces the *SlicePilot* framework along with its components. Section IV outlines the experimental setup used for real-world validation while Section V presents the evaluation results, followed by a discussion of related work in Section VI. Finally, Section VII concludes our work.

## II. BACKGROUND

**5G System Architecture.** A 5G cellular system, as illustrated in Fig. 1, consists of three main components: the RAN, which provides over-the-air connectivity between User Equipment (UE) and the network; the CN, which manages session control, mobility, and interconnectivity with external Data Networks (DNs); and the Transport Network (TN), responsible for carrying traffic between RAN and CN elements.

**Radio Access Network:** Unlike previous generations, the 5G RAN embraces a disaggregated virtualized architecture that enhances efficiency and scalability by allowing network operators to dynamically allocate resources based on service demands [23]. This is achieved by virtualizing baseband processing functions, enabling flexible deployment on cloud infrastructures. The RAN is composed of three main units: the Radio Unit (RU), which handles radio transmission, reception, and antenna processing; the Distributed Unit (DU), responsible for lower-layer processing such as error correction and resource allocation; and the Centralized Unit (CU), which manages higher-layer functions [23].

**Core Network:** The 5G CN is designed with a service-based, cloud-native architecture to ensure modularity, scalability, and support for network slicing [24]. It is composed of several control plane VNFs that operate on a consumer-producer model, communicating with each other via standardized REST APIs. At the heart of the 5G Core is the Access and Mobility Management Function (AMF), which acts as the communication hub between the RAN, UE, and other

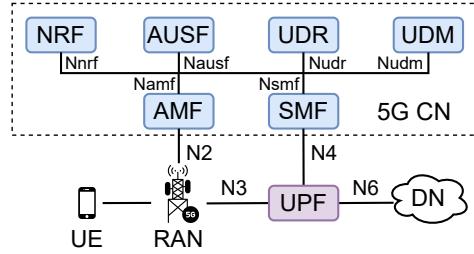


Fig. 1: 5G system architecture.

VNFs, providing essential signaling and mobility management services. The Authentication Server Function (AUSF) provides authentication credentials to the AMF, working in conjunction with the Unified Data Management (UDM) and Unified Data Repository (UDR) to implement the 5G Authentication and Key Agreement (AKA) procedure. Within the CN, the Session Management Function (SMF) and User Plane Function (UPF) serve as the control and user plane anchors, respectively. The UPF tunnels user traffic toward an external DN node, which represents real-world application servers. Finally, the Network Repository Function (NRF) facilitates initial discovery and communication among VNFs.

**Network Slicing.** A network slice is an end-to-end logical network instance formed by service chaining a set of VNFs, including both control and user plane components, to meet specific application requirements. Each VNF has distinct resource requirements, and is deployed across a virtualized infrastructure. While network slicing enables unprecedented flexibility by allowing operators to create isolated, end-to-end virtual networks, it also introduces complex challenges in orchestration, resource allocation, and slice placement optimization [2]. Several standardization efforts have been proposed to define and manage the lifecycle of network slices. Among them, the ETSI NFV architecture provides a comprehensive framework for deploying VNFs, and automating slice operations [25]. At its core, the Network Function Virtualization Orchestrator (NFVO) coordinates slice instantiation, scaling, and lifecycle management. The Virtualized Infrastructure Manager (VIM) oversees computing, storage, and networking resources, translating NFVO decisions into infrastructure-level actions. Beneath these layers, the Network Functions Virtualization Infrastructure (NFVI) unifies physical and virtual resources across edge, distributed, and central cloud domains, providing the foundation for hosting network slices.

## III. SLICEPILOT FRAMEWORK

*SlicePilot* is a modular framework designed to optimize network slice placement through a multi-stage approach. It consists of three functional blocks: (1) the User Plane Traffic Capturing Block (UPTCB), which collects and profiles over-the-air user plane traffic patterns for various slice types; (2) the User Plane VNF Stressing Block (UPVSB), which estimates VNF resource demands across different slice types; and (3) the Slice Placement Block (SPB), which optimizes slice deploy-

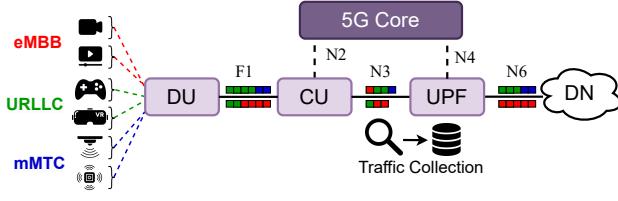


Fig. 2: UPTCB: Collecting and profiling over-the-air slice-specific user plane traffic for realistic replay and analysis.

ment by leveraging slice-specialized RL agents to minimize network operator costs while ensuring SLA compliance.

#### A. User Plane Traffic Capturing Block

The UPTCB is the first building block of the *SlicePilot* framework, designed to collect, preprocess, and profile over-the-air user plane traffic, enabling realistic slice performance analysis and optimization. Unlike synthetic traffic generators such as iPerf3 [26], which offer controlled traffic generation but lack configurability for complex application-specific patterns, UPTCB builds a structured repository of preprocessed flows aligned with the three primary 5G slice types: eMBB for bandwidth-intensive applications, URLLC for ultra-low latency services, and mMTC for massive IoT deployments. By replaying these traffic profiles through the user plane path, as illustrated in Figure 2, UPTCB enables accurate VNF resource estimation under real-world conditions, supporting informed slice placement and scaling. For interested readers, details on the traffic collection methodology are provided in Section §IV.

To illustrate the diversity of traffic patterns captured by UPTCB, Figure 4 presents three representative traces—one for each 5G slice type—highlighting both packet rate and inter-arrival time. The first pair of plots illustrates traffic from an eMBB slice (video streaming via YouTube), which exhibits periodic high-throughput bursts and longer inter-packet intervals, with a strong downlink dominance. The second pair corresponds to URLLC traffic (online gaming via Fortnite), where smaller payloads and shorter inter-arrival times reflect the low-latency, interactive nature of the application. The final pair represents mMTC traffic from a connected IoT sensor, characterized by infrequent transmissions, small payload sizes, and low sustained load. These distinct traffic patterns underscore the importance of slice-specific traffic profiling, as each slice type exhibits unique characteristics that demand different computational and network resources. Accurate profiling enables precise resource estimation and supports informed, adaptive, and efficient placement and scaling decisions.

#### B. User Plane VNF Stressing Block

The UPVSB is the second component of the *SlicePilot* framework, responsible for accurately estimating the resource requirements of each user plane VNF under different slice configurations and operational conditions. By applying controlled stress across varying load levels and slice types, UPVSB evaluates resource consumption and performance impact, providing

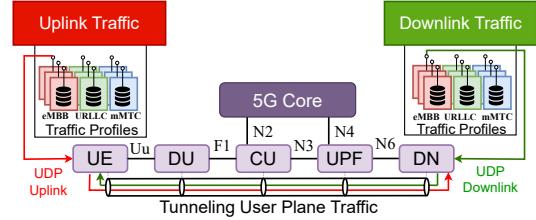


Fig. 3: UPVSB: Replay slice-specific traffic through the user plane for VNF resource profiling.

critical insights for optimal VNF placement, capacity provisioning, and efficient network orchestration. As illustrated in Figure 3, UPVSB replays traffic profiles collected by UPTCB through the user plane path to systematically quantify per-slice resource demands, measuring the resulting computational and storage footprint across VNFs. This process produces a lookup table that captures upper bounds on resource usage as a function of slice type, service type, and the number of active user sessions, supporting rapid runtime decisions. UPVSB is implemented on top of the OAI 5G software stack, and further implementation details are provided in Section §IV.

To showcase UPVSB’s resource estimation methodology, we examine CPU utilization variations across slice types as the number of active user connections steadily increases over time. As shown in Figure 5, we evaluate resource utilization across user plane VNFs, including the DU, CU, and UPF, for three different slices—one from each category: eMBB, URLLC, and mMTC. CPU utilization scales sharply with user load for the eMBB slice, which supports a video streaming application. With 200 active connections, DU utilization rises to 31.67%, CU utilization to 31.04%, and UPF utilization peaks at 41.48%, reflecting the processing overhead induced by high-throughput data plane traffic. Even at a lower load of 50 active connections, DU, CU, and UPF utilization already reach 11.01%, 9.18%, and 14.15%, respectively, emphasizing the significant resource demands of high-bandwidth services. In contrast, the URLLC slice, customized for an online gaming application (Fortnite), maintains significantly lower CPU utilization. With 200 active connections, DU utilization remains at just 2.93%, while CU and UPF utilization reach only 1.19% and 2.71%, respectively. This reflects the small packet sizes and stringent latency constraints of real-time applications, which require rapid but lightweight processing rather than heavy data forwarding. The mMTC slice, tailored for IoT sensor connectivity, exhibits a more gradual increase in CPU utilization as user connections grow. When the number of connected devices reaches 200, DU utilization is 2.77%, CU utilization reaches 1.82%, and UPF utilization remains at 2.82%, demonstrating a steady yet lower resource footprint compared to eMBB. Even at the lowest tested load of 10 connections, all user plane VNFs maintain utilization below 1.5%, reinforcing the low computational demand of IoT-based communication. Based on these insights, the next step is determining how to strategically place VNFs to efficiently accommodate diverse traffic demands while ensuring optimal network performance,

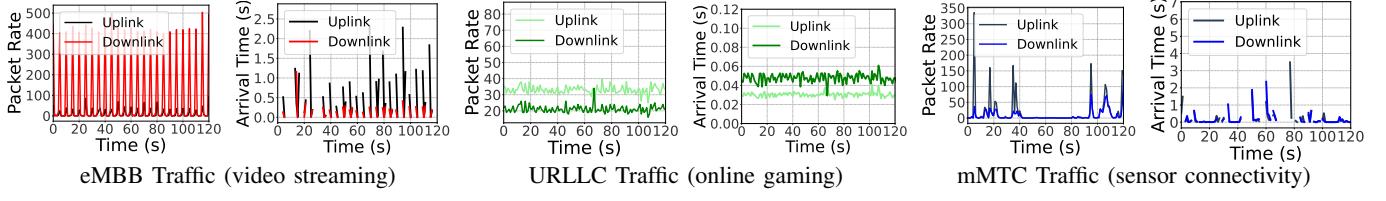


Fig. 4: Traffic profiling of eMBB, URLLC, and mMTC slices, illustrating packet rate (left) and inter-arrival time (right).

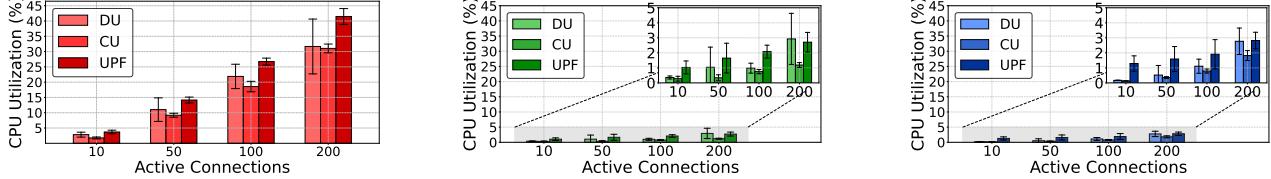


Fig. 5: User plane VNF resource usage for eMBB (red), URLLC (green), and mMTC (blue) slices under increasing user load.

resource utilization, and cost-effectiveness.

### C. Slice Placement Block

The final block of the *SlicePilot* framework is a fully customized MARL scheduler, designed to optimize slice deployment within a heterogeneous multi-cloud infrastructure. It employs a disaggregated agent design to accelerate convergence, improve stability, and simplify decision-making by assigning each agent to manage slice requests of the same type. This specialization reduces the state-action space each agent needs to explore, making slice placement more efficient and scalable compared to monolithic single-agent approaches [17], [27], [28]. Operating within the NFVO, the scheduler computes placement decisions and relays them to the VIM for slice deployment and instantiation within the NFVI. The overall architecture of the SPB is illustrated in Figure 6.

**Optimization Framework.** We represent each network slice as an undirected graph  $G = (V, L)$ , where  $V$  denotes the set of VNFs composing the slice, and  $L$  represents the set of links interconnecting them. To distinguish between functional roles, the set  $V$  is further decomposed into two subsets:  $V_u$ , which contains the user plane VNFs, and  $V_c$ , which contains the control plane VNFs, such that  $V = V_u \cup V_c$  and  $V_u \cap V_c = \emptyset$ . Each VNF  $v \in V$  has resource demands in terms of computation and storage, denoted by  $r_v = (r_v^{\text{cpu}}, r_v^{\text{mem}})$ . The total resource capacity of the NFVI is distributed across  $M$  different cloud infrastructures, where each infrastructure  $m \in M$  has a total resource capacity  $C_m = (C_m^{\text{cpu}}, C_m^{\text{mem}})$ . VNFs collocated in the same infrastructure  $m$  experience negligible link latency, while VNFs placed in different infrastructures ( $m \neq m'$ ) experience a transmission and propagation latency denoted as  $D_{m,m'}$ . Finally, each cloud infrastructure  $m$  incurs a cost associated with resource usage, denoted by  $\lambda_m$ , charged hourly per baseline unit of computational and storage resources.

To model the slice placement problem, we introduce a binary decision variable  $x_{v,m}$ , where  $x_{v,m} = 1$  indicates that VNF  $v$  is placed on infrastructure node  $m$ , and  $x_{v,m} = 0$

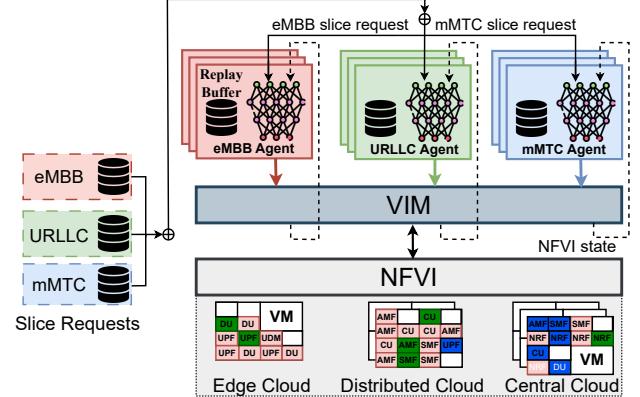


Fig. 6: High-level overview of the SPB block.

otherwise. To ensure feasibility, we introduce the following constraints:

First, each VNF must be placed in one cloud infrastructure:

$$\sum_{m \in M} x_{v,m} = 1, \quad \forall v \in V \quad (1)$$

Second, the total computational demand of VNFs assigned to cloud infrastructure  $m$  must not exceed its capacity:

$$\sum_{v \in V_{\text{total}}} r_v^{\text{cpu}} \cdot x_{v,m} \leq C_m^{\text{cpu}}, \quad \forall m \in M \quad (2)$$

Third, the total storage resources requested by VNFs in a cloud infrastructure  $m$  must not exceed its storage capacity:

$$\sum_{v \in V_{\text{total}}} r_v^{\text{mem}} \cdot x_{v,m} \leq C_m^{\text{mem}}, \quad \forall m \in M \quad (3)$$

Fourth, depending on the slice type  $z \in Z$ , the scheduler needs to activate only the relevant slice-specific constraints from a set of available constraints. While several new constraints can be introduced, we consider two general cases applicable to most slice types: one constraint for placing the VNFs to ensure the end-to-end user plane latency, denoted as  $D_z$ ,

is satisfied —where this latency represents the total delay experienced as traffic traverses through all user plane VNFs before reaching the application servers—, and an additional constraint for consolidating VNFs within the same cloud infrastructure to minimize inter-cloud communication overhead, which is particularly beneficial for slices that generate high volumes of signaling traffic. The user plane latency and VNF consolidation constraints are detailed in Equations (4) and (5), respectively. We focus on these two constraints as they reflect the current enforcement capabilities of existing 5G open-source software stacks such as OAI, where more complex or fine-grained placement policies cannot yet be fully realized.

$$\sum_{v,v+1 \in V_z} \sum_{m \in M} \sum_{m' \in M} D_{m,m'}^{\text{link}} \cdot x_{v,m} \cdot x_{v+1,m'} \leq D_z, \quad (4)$$

$$x_{v,m} = x_{v',m}, \quad \forall v, v' \in V_z, \quad \forall z \in Z, \quad \forall m \in M \quad (5)$$

Based on this formulation, the objective of the scheduler is to optimally place the VNFs of each slice to minimize the total costs associated with reserving computational and storage resources in the heterogeneous multi-cloud infrastructure:

$$\begin{aligned} \min_x \quad & \sum_{v \in V} \sum_{m \in M} \lambda_m \cdot r_v^{\text{cpu}} \cdot r_v^{\text{mem}} \cdot x_{v,m} \\ \text{s.t.} \quad & \text{constraints: (1) - (5)} \\ & x_{v,m} \in \{0, 1\}, \quad \forall v \in V, \quad \forall m \in M \end{aligned} \quad (6)$$

The optimization problem formulated in (6) is an integer linear programming problem and a variant of the VNE problem (NP-hard). Given this complexity, obtaining an optimal solution using conventional optimization methods is computationally intractable. Therefore, to approximate the solution we adopt a learning-based heuristic approach to efficiently derive near-optimal solutions while reducing computational overhead.

**Slice-Wise Problem Decomposition.** We decompose the optimization problem described in (6) into a set of smaller optimization problems, each corresponding to a specific slice type  $z \in Z$ . These sub-problems can be solved independently based on slice request arrival times, i.e., the first slice request is processed first, followed by subsequent requests in their respective order of arrival. We model each sub-problem as a Markov Decision Process (MDP) [29], viewing it as a sequential decision-making process where VNFs are placed one at a time until all required VNFs are deployed. Each MDP state  $s_t \in S$  is described as follows:

$$s_t = [\hat{\mathbf{C}}(t), Q_z(t), \mathbf{T}_z(t), \mathbf{L}_z(t)] \quad (7)$$

where  $\hat{\mathbf{C}}(t) = (\hat{C}_1(t), \hat{C}_2(t), \dots, \hat{C}_m(t)) \in \mathbb{R}^{1 \times 2m}$  is a row vector representing the currently available computational and storage resources in each cloud infrastructure  $m$ ,  $Q_z(t) \in \mathbb{R}$  is a scalar denoting the total number of queued slices of type  $z$  at time step  $t$ ,  $\mathbf{T}_z(t) = (\sum_{q \in Q_z} \sum_{v \in V} r_v^{\text{cpu}}, \sum_{q \in Q_z} \sum_{v \in V} r_v^{\text{mem}}) \in \mathbb{R}^{1 \times 2}$  is a row vector describing the total aggregated resource demand of all queued slices of type  $z$  at time step  $t$ , and  $\mathbf{L}_z(t) = (L_1(t), L_2(t), \dots, L_{Q_z}(t)) \in \mathbb{R}^{1 \times Q_z}$  is a row vector containing the SLA-related parameters for all slices of type  $z$ . The

action selection process follows a common structure across all MDPs, where the action space  $a_t \in A$  is represented as a one-hot vector:

$$a_t = [a_1(t), a_2(t), \dots, a_m(t)] \in \{0, 1\}^m \quad (8)$$

where  $a_m(t) \in \{0, 1\}$  denotes the binary placement decision for VNF  $v$  on cloud infrastructure  $m$  at time step  $t$ . Lastly, each MDP has its own reward function  $R : S \times A \rightarrow \mathbb{R}$ , to evaluate the effectiveness of the chosen actions in achieving the optimization objective while satisfying the given constraints. Three types of rewards are assigned in each MDP:

(1) A reward  $r_1 \in \mathbb{R}$  is assigned during the placement of each VNF to penalize resource constraint violations:

$$r_1(s_t, a_t) = \begin{cases} \delta_1, & \text{if } a_t \text{ violates resource constraints} \\ 0, & \text{otherwise.} \end{cases} \quad (9)$$

where  $\delta_1$  is a negative scalar ( $\delta_1 < 0$ ).

(2) A reward  $r_2 \in \mathbb{R}$  is assigned during the placement of each VNF to encourage cost-efficient decisions:

$$r_2(s_t, a_t) = \boldsymbol{\delta}_2 \cdot \mathbf{I}(a_t) \quad (10)$$

where  $\boldsymbol{\delta}_2 = [\delta_2(1), \delta_2(2), \dots, \delta_2(M)]$  is a predefined row vector that assigns a cost-efficiency reward  $\delta_2(m)$  to each selected infrastructure  $m$ , and  $\mathbf{I}(a_t)$  is a one-hot column vector indicating the selected infrastructure  $m$  at time step  $t$ .

(3) A reward  $r_3 \in \mathbb{R}$  is given at the end of the slice placement process for each slice to ensure compliance with SLAs:

$$r_3(s_t, a_t) = \begin{cases} \delta_3, & \text{if the slice's SLAs are met,} \\ 0, & \text{otherwise.} \end{cases} \quad (11)$$

where  $\delta_3$  is a positive scalar ( $\delta_3 > 0$ ). Therefore, the total reward  $R_t(s_t, a_t)$  at each time step is the sum of the individual rewards assigned during the slice placement process:

$$R_t(s_t, a_t) = r_1(s_t, a_t) + r_2(s_t, a_t) + r_3(s_t, a_t). \quad (12)$$

**Learning Slice-Wise Placement Policies.** To solve each formulated MDP, we employ RL techniques to determine the optimal sequence of actions that maximize the expected cumulative rewards. Specifically, for each MDP, the objective is to learn a dedicated policy—mapping states to actions—that maximizes the accumulated time-discounted rewards, defined as  $\pi^* = \arg \max_{\pi} \mathbb{E} [\sum_{t=0}^{\infty} \gamma^t R_t(s_t, a_t)]$ , where  $\gamma \in (0, 1]$  is the discount factor regulating the importance of future rewards. Each MDP is handled by a corresponding agent, which interacts with its environment by observing states, selecting actions, and receiving feedback in the form of rewards. Through this continuous interaction, each agent gradually refines its strategy, ultimately converging to an optimal policy tailored to its slice-specific placement problem.

A key challenge in this formulation is how agents retain and utilize past experiences. Classical RL methods often rely on tabular approaches to store and update experiences, but these become impractical in large state-action spaces due to limited generalization [30]. In our case, the state-action space per agent grows exponentially as  $\mathcal{O}(|V| \cdot |M| \cdot |Q_z| \cdot |M|^{|V|})$ ,

necessitating scalable learning techniques for efficient generalization. Function approximation has been explored to address this issue [31], with Deep Neural Networks (DNNs) emerging as a promising solution due to their success in recent applications [27], [28]. Motivated by this, we employ DNNs as value function approximators within our agents to cope with the high-dimensional state-action space.

**Deep Q-Learning for Value Function Approximation.** To enable efficient decision-making in each agent, we adopt *Deep Q-Learning* (DQL), an RL approach that integrates Q-learning with DNNs to generalize over high-dimensional state-action spaces [32], [33]. The goal is to learn a Q-function that maps each state-action pair  $(s, a)$  to the expected cumulative reward obtained by taking action  $a$  in state  $s$  and following a policy thereafter, formally defined as  $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ .

The optimal Q-function, denoted as  $Q^*$ , satisfies the Bellman equation, which recursively relates the value of a state-action pair to the immediate reward and the discounted value of the next state:

$$Q^*(s, a) = \mathbb{E}_{s'} \left[ r(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right], \quad (13)$$

where  $s'$  is the next state,  $a'$  is the next action, and the expectation is taken over the transition distribution  $P(s'|s, a)$ . To approximate  $Q^*(s, a)$ , we use a DNN parameterized by  $\theta$ , denoted as  $Q_\theta(s, a)$ . The agent iteratively updates this network through a value iteration process that refines its reward estimates over time [32]. This is achieved by minimizing the difference between the predicted and target Q-values using a mean squared error loss:

$$L(\theta) = \mathbb{E} [(y - Q_\theta(s, a))^2], \quad (14)$$

where the target value  $y$  is computed as:

$$y = r(s, a, s') + \gamma \max_{a'} Q_\theta(s', a'). \quad (15)$$

To optimize  $Q_\theta(s, a)$ , the parameters  $\theta$  are updated via gradient descent:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta), \quad (16)$$

where  $\alpha$  is the learning rate and  $\nabla_\theta L(\theta)$  denotes the gradient of the loss with respect to the network parameters. Unlike supervised learning, the target in DQL depends on the same network being trained, which can lead to instability during learning. To address this, we apply the following enhancements to improve training stability and sample efficiency:

- 1) **Target Network.** We maintain a delayed version of the Q-function,  $Q_{\theta'}(s, a)$  [33], to compute the target value more reliably. The target in (15) is modified as:

$$y = r(s, a, s') + \gamma \max_{a'} Q_{\theta'}(s', a'), \quad (17)$$

and the target network is updated every  $C$  steps by copying weights from the current network.

- 2) **Double Q-Learning.** To mitigate the overestimation bias caused by using the same Q-function for both action selection and evaluation, we use Double Q-learning [34].

The action that maximizes the Q-value is selected using the current network:

$$a^* = \arg \max_{a'} Q_\theta(s', a'), \quad (18)$$

while the corresponding Q-value is evaluated using the target network:

$$y = r(s, a, s') + \gamma Q_{\theta'}(s', a^*). \quad (19)$$

- 3) **Experience Replay.** To improve sample efficiency and reduce temporal correlations, each agent maintains an experience replay buffer  $\mathcal{I}$  that stores past transitions  $(s, a, r, s')$ . Instead of updating the network using only the most recent experience, we sample a mini-batch of  $B$  transitions from  $\mathcal{I}$  during each training iteration. The buffer holds up to  $N$  transitions and enables more stable and data-efficient learning.

#### IV. EXPERIMENTATION ENVIRONMENT

In this section, we present the experimental setup used to realize the functional blocks of *SlicePilot*.

**User Plane Traffic Collection.** For the traffic collection campaign, we leveraged our in-lab over-the-air 5G standalone testbed, which operated in the n78 TDD band with a 40 MHz bandwidth and utilized a USRP B210 SDR as the RF front end, as illustrated in Figure 7. The baseband processing and core network VNFs were deployed as Docker containers using the latest OAI release [22]. A Google Pixel 7 mobile device served as the primary UE for most network scenarios. Additionally, to connect external devices, we tethered the Pixel to an OpenWRT router acting as a gateway to our 5G network. We considered two external devices: an AR/VR headset (Microsoft HoloLens 2) and a Govee WiFi Thermometer Hygrometer H5179 smart humidity and temperature IoT sensor.

Building on this setup, we conducted a series of traffic collection experiments targeting three slice types: URLLC, eMBB, and mMTC. For each application scenario, we launched the base station (gNB), verified its connection to the core network, and connected the UE by disabling airplane mode. Once the SMF assigned an IP address, we captured traffic on the gNB host using *tcpdump* [35], filtering by the UE's IP to isolate user-plane packets and exclude control signaling. Each scenario was recorded for 4 hours to ensure sufficient traffic volume and diversity. The application use cases were grouped by slice type: URLLC included VoIP (Messenger, WhatsApp), online gaming (Fortnite, PUBG), web browsing, and AR/VR (Microsoft Dynamics 365 Remote Assist); eMBB included media streaming and downloads (YouTube, TikTok), social media (Facebook, Instagram), and video conferencing (Messenger, Skype, Zoom); mMTC focused on IoT traffic.

**Large-Scale VNF Stress Testing.** To enable large-scale slice instantiation and traffic replay, we replaced the physical USRP and devices with OAI's emulated RF front ends and NR-UEs. For each network slice, we instantiated dedicated user plane VNFs and configured static routes between them to ensure correct forwarding across the data path. We also customized the NR-UE and traffic server containers to replay only the

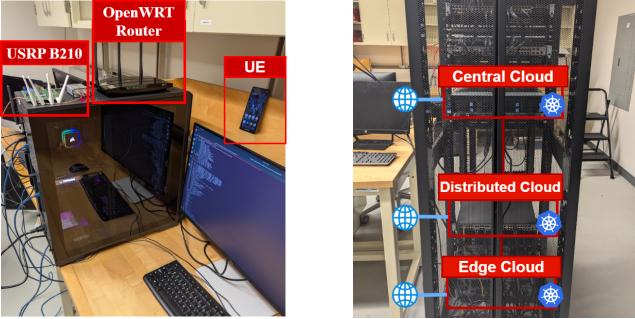


Fig. 7: Experimental setup: (left) Over-the-air 5G testbed, (right) heterogeneous multi-cloud infrastructure.

relevant traffic profiles corresponding to each slice type. To support concurrent high-volume traffic flows, we implemented a lightweight UDP server within each VNF, binding it to multiple ports to handle parallel connections from emulated devices. During each experiment, we replayed the collected traffic traces through the instantiated slices, ensuring that each slice carried its own application-specific load. Throughout the experiments, we monitored VNF-level resource consumption metrics—including CPU and RAM usage—under varying connection loads to analyze the relationship between traffic intensity and compute demand. All experiments were conducted on a commercial-grade server equipped with an AMD EPYC 7352 24-core processor and 128 GB of DDR4 RAM.

**Multi-Cloud Infrastructure and Scheduler Deployment.** To evaluate our scheduler in a real deployment, we set up a multi-cloud infrastructure with OpenStack as the VIM to manage computing, storage, and networking resources. A Kubernetes cluster was deployed on top of OpenStack to serve as a simplified NFVO, orchestrating network slices and handling their lifecycle. The MARL scheduler was integrated within the NFVO, interacting with the environment by receiving state updates from the Kubernetes Metrics Server and issuing placement decisions via the Kubernetes API Server. As illustrated in Figure 7, the infrastructure was deployed across three COTS servers, each equipped with an AMD EPYC 7352 24-core processor, 128 GB DDR4 RAM, and four NVIDIA RTX A5000 GPUs. These servers served as computing nodes representing the edge, distributed, and central cloud infrastructures. Since all servers were physically colocated within the same rack, kernel modifications were applied to introduce latencies, accurately emulating the latency characteristics of a geographically distributed infrastructure.

## V. EVALUATION RESULTS

### A. Configurations and Settings

**Infrastructure Configurations.** We consider  $M = 3$  heterogeneous cloud infrastructures—edge, distributed, and central—and set their resource capacities as follows:  $C_0 = (16 \text{ CPU cores}, 16 \text{ GiB RAM})$ ,  $C_1 = (32 \text{ CPU cores}, 32 \text{ GiB RAM})$ , and  $C_2 = (64 \text{ CPU cores}, 64 \text{ GiB RAM})$ , respectively. Each infrastructure hosts virtual machines instantiated using predefined flavors: (1 CPU core, 1 GiB RAM), (2 CPU cores, 2 GiB RAM), and (4 CPU cores, 4 GiB RAM). We select cost

values to reflect the hourly charge for on-demand reservation of 1 CPU core and 1 GiB RAM, set as  $\lambda_0 = 0.010 \text{ \$/h}$  at the edge,  $\lambda_1 = 0.005 \text{ \$/h}$  at the distributed cloud, and  $\lambda_2 = 0.001 \text{ \$/h}$  at the central cloud, following commercial pricing trends [36]. Inter-cloud latencies are modeled as normal distributions, following values reported in [37]: edge-to-distributed latency  $D_{0,1}^{\text{link}} \sim \mathcal{N}(0.5, 0.1^2) \text{ ms}$ , and distributed-to-central  $D_{1,2}^{\text{link}} \sim \mathcal{N}(20, 1^2) \text{ ms}$ . Finally, latency to external networks depends on UPF placement—5 ms at the edge, 7.5 ms in the distributed cloud, and 10 ms in the central cloud.

**Slice Configurations.** We represent each slice with dedicated VNFs, including NRF, UDR/UDM/AUSF (U/U/A), AMF, SMF, UPF, CU, and DU. As a proof of concept, we assume uniform resource demands across slices using the default values provided by OAI. NRF is assigned  $r_0 = (0.15 \text{ CPU cores}, 0.128 \text{ GiB RAM})$ , UDR/UDM/AUSF requires  $r_1 = (0.65 \text{ CPU cores}, 0.896 \text{ GiB RAM})$ . AMF and SMF are allocated  $r_3 = (0.25 \text{ CPU cores}, 0.256 \text{ GiB RAM})$  and  $r_4 = (0.25 \text{ CPU cores}, 0.256 \text{ GiB RAM})$ , respectively. Similarly, UPF and CU are assigned  $r_5 = (0.5 \text{ CPU cores}, 0.512 \text{ GiB RAM})$  and  $r_6 = (0.5 \text{ CPU cores}, 0.512 \text{ GiB RAM})$ , respectively, while the DU requires  $r_7 = (3 \text{ CPU cores}, 2 \text{ GiB RAM})$ . Finally, we define latency constraints for each slice type. For URLLC slices, we set the end-to-end user plane latency requirement to  $D_{\text{URLLC}} < 10 \text{ ms}$ , while for eMBB and mMTC slices,  $D_{\text{eMBB}} < 20 \text{ ms}$  and  $D_{\text{mMTC}} < 50 \text{ ms}$ , respectively.

**Training and RL Configurations.** All *SlicePilot* agents are trained using a shared configuration, and the reported values are obtained through hyperparameter optimization: batch size  $B = 32$ , replay buffer size  $\mathcal{I} = 20,000$ , number of episodes  $E = 50,000$ , and synchronization interval  $C = 1000$ . An exploration decay of  $\lambda_\epsilon = 25,000$  is used for all agents. Each agent employs the ReLU activation function, with three hidden layers of size 128 for eMBB and URLLC, and 256 for mMTC. The learning rate is set to 0.05 for eMBB, 0.01 for URLLC, and 0.005 for mMTC, while the discount factor  $\gamma$  remains 0.01 across all agents. The reward function parameters are set as follows:  $\delta_1 = -100$ ,  $\delta_2 = [1, 2, 4]$ , and  $\delta_3 = 15, 20$ , and 10 for eMBB, URLLC, and mMTC, respectively.

### B. Placement Algorithms

We compare *SlicePilot*'s disaggregated scheduler against a set of baseline placement algorithms:

**1) ILP Algorithm.** Solves the optimization problem in (6) to obtain optimal placement but is computationally expensive. We use it as an *upper bound* on placement performance. A simplified version is employed by OSM MANO [38].

**2) Cost-Aware Algorithm.** A heuristic that places slices in the lowest-cost cloud tiers until resources are exhausted.

**3) Performance-Aware Algorithm.** A heuristic that places slices on infrastructures closer to users, shifting to higher-tier clouds as local resources deplete.

**4) Random Algorithm.** A heuristic that assigns slice placements arbitrarily, without optimization criteria.

**5) Load Balancing Algorithm (Kubernetes Scheduler).** The default Kubernetes scheduler, which places VNFs based on

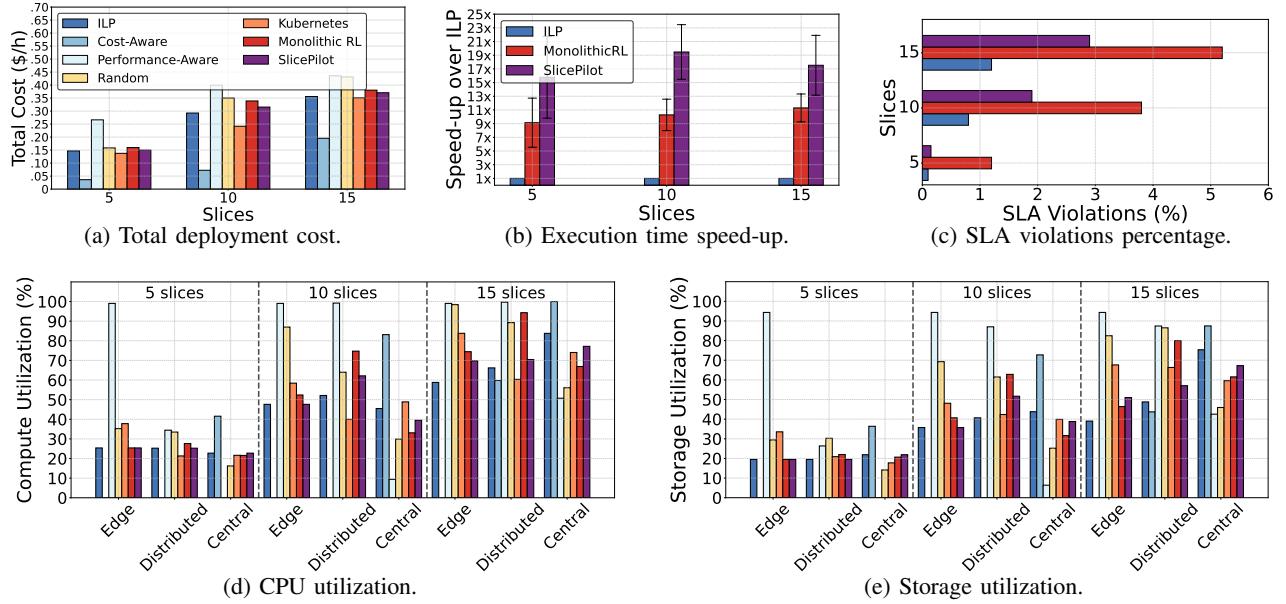


Fig. 8: Performance evaluation of *SlicePilot* and baseline placement algorithms in terms of (a) deployment cost, (b) execution time speed-up, (c) SLA violation rate, (d) CPU utilization, and (e) storage utilization.

resource availability to balance utilization across the multi-cloud infrastructure [39].

**6) Monolithic RL Algorithm:** A single-agent RL approach for all slice placement decisions. It uses the same training setup as *SlicePilot*, with a 3-layer DNN (128 neurons/layer), ReLU activation, learning rate  $\alpha = 0.05$ , and discount factor  $\gamma = 0.01$ , selected via hyperparameter optimization.

While Algorithms 2)–5) enable fast placement decisions, they ignore slice-specific SLAs. Thus, we evaluate Algorithms 1), 6), and *SlicePilot* based on execution time and SLA compliance, and use Algorithms 2)–5) to assess resource utilization and deployment cost.

### C. Experimental Results

We evaluate *SlicePilot* based on computational efficiency, cost, resource utilization, and SLA compliance. To test adaptability under varying slice requests, we consider deployments of 5, 10, and 15 slices, corresponding to 25%, 50%, and 75% of total infrastructure capacity. Slice requests—URLLC, eMBB, and mMTC—are sampled per scenario from a multinomial distribution with probabilities 0.2, 0.3, and 0.5, respectively, to ensure diversity in slice types. Each scenario is executed multiple times, and the reported results reflect the average performance across all trials.

**Cost Efficiency.** We evaluate the hourly deployment cost (\$/h) of network slices across placement strategies. As shown in Figure 8(a), *SlicePilot* consistently achieves near-optimal cost, closely matching the ILP baseline. At 25%, 50%, and 75% load, *SlicePilot*'s costs are only 2.04%, 7.8%, and 4.1% higher than ILP, respectively. In contrast, the Monolithic RL incurs notably higher costs due to centralized decision-making, with up to 15.8% overhead at medium load. Heuristic strategies exhibit a trade-off: the Cost-Aware algorithm minimizes cost

but suffers from high SLA violations, while Performance-Aware prioritizes SLAs at a significantly higher cost—up to 81% above ILP. Load Balancing offers moderate savings but lacks SLA-awareness, and Random placement performs the worst overall. Notably, deployment costs do not scale proportionally with the number of slices. This is primarily due to the heterogeneous nature of cloud resources, and the pricing differences across cloud tiers.

**Computational Efficiency.** Second, we evaluate the computational efficiency of different slice placement strategies relative to ILP, defined as the time required to compute placement decisions upon slice request arrival. Figure 8(b) presents the speed-up achieved by Monolithic RL and *SlicePilot* compared to ILP for different number of slice requests. As expected, ILP exhibits the highest execution time due to its exhaustive search process, making it computationally prohibitive for large-scale slice deployments. In contrast, Monolithic RL achieves a speed-up of approximately 9x for 5 slices, increasing to 11x for 10 slices and 10x for 15 slices. *SlicePilot* further improves computational efficiency, demonstrating a speed-up of approximately 16x for 5 slices, 19x for 10 slices, and 17x for 15 slices. This reduction in execution time is due to the agents' learned ability to apply optimal placement decisions, thereby avoiding brute-force combinatorial searches during inference. Overall, *SlicePilot* achieves up to 73% faster execution than Monolithic RL and maintains an order-of-magnitude improvement over ILP, making it a practical and scalable solution for real-time slice placement. However, *SlicePilot* does not achieve a strict  $\times 3$  improvement over Monolithic RL due to the uneven distribution of slices among its schedulers, resulting from the multinomial sampling, coupled with variations in the DNN architecture of each agent.

**SLA Compliance.** Third, we assess how different slice placement strategies uphold compliance with predefined SLAs. As shown in Figure 8(c), SLA violation levels vary significantly across strategies. ILP achieves the highest level of compliance, with SLA violations remaining at only 0.1%, 0.8%, and 1.2% for deployments of 5, 10, and 15 slices, respectively. This is expected, as ILP enforces SLA constraints during slice placement decisions. *SlicePilot* demonstrates a strong balance between SLA compliance and deployment efficiency, achieving significantly fewer violations than Monolithic RL while closely approximating ILP. Across deployments, *SlicePilot* maintains violations at 0.15%, 1.9%, and 2.9% for 5, 10, and 15 slices, respectively. Compared to ILP, *SlicePilot* incurs 1.5x times more violations for small deployments, 2.375x times more for medium deployments, and 2.42x times more for large deployments. Nevertheless, *SlicePilot* achieves substantially better SLA compliance than Monolithic RL. In contrast, Monolithic RL exhibits a significantly higher percentage of SLA violations due to its centralized decision-making process. SLA violations reach 1.2%, 3.8%, and 5.2% for deployments of 5, 10, and 15 slices, respectively. Finally, while Load Balancing (Kubernetes) offers lower cost, as illustrated in Figure 8(a), its SLA compliance remains capped at 80%, making it less viable than RL-based strategies and, for that reason, it is not included in Figure 8(c).

**Multi-Cloud Resource Footprint and Utilization.** Finally, we analyze how different slice placement strategies allocate compute and storage resources across the edge, distributed, and central cloud infrastructures as the number of slices increases. As illustrated in Figures 8(d),(e) resource utilization patterns differ significantly based on the placement strategy, particularly in how VNFs are distributed across cloud tiers.

*SlicePilot* demonstrates balanced resource utilization across all cloud layers. For 5 slices, it distributes 25.4% of CPU load and 19.5% of RAM usage across the edge and distributed clouds while utilizing 22.7% CPU and 21.9% RAM in the central cloud. As demand increases, *SlicePilot* efficiently scales resource allocation, reaching 47.6% CPU and 35.7% RAM utilization across edge and distributed tiers for 10 slices, while maintaining 39.5% CPU and 38.8% RAM in the central cloud. At 15 slices, *SlicePilot* sustains 69.8% CPU and 51.0% RAM in edge and distributed clouds, offloading from the central cloud, which reaches 77.2% CPU and 67.2% RAM. These results indicate that *SlicePilot* effectively balances workload distribution, reducing over-reliance on expensive edge cloud resources. In contrast, ILP achieves the lowest cost but heavily loads the central cloud, reaching 83.8% CPU and 75.3% RAM utilization for 15 slices, reflecting its tendency to exhaust low-cost resources first. Compared to *SlicePilot*, Monolithic RL exhibits greater centralization, with CPU usage in the central cloud rising from 21.6% (5 slices) to 66.9% (15 slices), and RAM reaching 61.5%. Among heuristic strategies, Cost-Aware minimizes edge and distributed usage, saturating the central cloud CPU at 99.9% for 15 slices. Performance-Aware favors edge placement, pushing edge CPU to 99.0% while underutilizing the central cloud. Load Balancing (Kubernetes)

distributes load more evenly, with central cloud usage at 74.4% CPU and 61.5% RAM. The Random strategy results in unpredictable slice placement, leading to 56.1% CPU and 45.9% RAM utilization in the central cloud.

## VI. RELATED WORK

The problem of joint resource allocation and slice placement has been extensively studied and is often modeled as a variant of the virtual network embedding problem [9]. Several surveys address slice lifecycle management and placement optimization [1], [40], covering techniques for resource allocation, performance guarantees, and network efficiency. Optimization-based methods provide optimal decisions under latency, isolation, and resource constraints [11], [12], but are computationally prohibitive for large-scale deployments. Heuristic approaches reduce complexity [13], [15] but struggle with dynamic network conditions and lack real-world validation. More recently, reinforcement learning has been used to improve placement scalability and adaptability [16], [18]–[20], though most studies remain limited to simulation settings, hindering their applicability in real 5G systems.

Despite 5G-compliant testbeds supporting large-scale slice deployment [21], [41], [42], an optimization engine for intelligent slice placement remains absent. Existing efforts focus primarily on scaling slices rather than optimizing their placement [43], [44]. For example, [43] introduces a slice-as-a-service framework and evaluates deployments on local testbeds with cloud cost estimations, but without optimizing placement. Similarly, [44] explores slice scaling across various topologies and use cases, yet lacks mechanisms for intelligent placement. While these studies validate slice deployment and scaling, they overlook optimization under real-world constraints. To our knowledge, no existing framework jointly addresses slice placement, traffic profiling, resource demand estimation, and RL-based optimization in a real heterogeneous multi-cloud 5G setup—highlighting a key gap in the literature.

## VII. CONCLUSION

This paper presents *SlicePilot*, a comprehensive framework for intelligent slice placement in heterogeneous multi-cloud infrastructures. Built on three functional blocks, *SlicePilot* seamlessly integrates (i) the User Plane Traffic Capturing Block to profile and replicate real-world 5G traffic patterns, (ii) the User Plane VNF Stressing Block to accurately estimate resource demands across slices, and (iii) the Slice Placement Block, a disaggregated multi-agent RL scheduler that automates VNF deployment while minimizing costs and ensuring SLA compliance. Through extensive experimentation on a self-managed multi-cloud testbed, *SlicePilot* delivers near-optimal placement efficiency, achieving deployment costs within 7.8% of ILP, 19x faster execution, and comparable SLA compliance, with only 2.42x more violations under high-load conditions. Unlike heuristic approaches, *SlicePilot* intelligently balances workload distribution, preventing costly edge overutilization while avoiding central cloud congestion, ensuring efficient, scalable, cost-effective deployments.

## REFERENCES

- [1] A. Kaloxylos, "A survey and an analysis of network slicing in 5g networks," *IEEE Communications Standards Magazine*, vol. 2, no. 1, pp. 60–65, 2018.
- [2] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5g: Survey and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 94–100, 2017.
- [3] European Telecommunications Standards Institute, "Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action," 2012. [Online]. Available: [https://portal.etsi.org/NFV/NFV\\_White\\_Paper.pdf](https://portal.etsi.org/NFV/NFV_White_Paper.pdf)
- [4] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks," 2012. [Online]. Available: <https://opennetworking.org/wp-content/uploads/2013/05/ONF-WhitePaper-SDN-and-NFV.pdf>
- [5] Ericsson, "5G Core (5gc) network: Get to the core of 5G," Jan 2025. [Online]. Available: <https://www.ericsson.com/en/core-network/5g-core>
- [6] Nokia, "5G Core on Hybrid Cloud," Jun 2022. [Online]. Available: <https://www.nokia.com/core-networks/5g-core/hybrid-cloud/>
- [7] A. W. Services, "5G Edge Computing Infrastructure – AWS Wavelength Features," Mar 2025. [Online]. Available: <https://aws.amazon.com/wavelength/features/>
- [8] ———, "Low latency, data residency and migration - aws local zones," Mar 2025. [Online]. Available: <https://aws.amazon.com/about-aws/global-infrastructure/localzones/?p=ngi&loc=3>
- [9] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer, and X. Hesselbach, "Virtual network embedding: A survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 1888–1906, 2013.
- [10] W. Attaoui, E. Sabir, H. Elbiaze, and M. Guizani, "Vnf and cnf placement in 5g: Recent advances and future trends," *IEEE Transactions on Network and Service Management*, vol. 20, no. 4, pp. 4698–4733, 2023.
- [11] Q. Zhang, F. Liu, and C. Zeng, "Adaptive interference-aware vnf placement for service-customized 5g network slices," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 2449–2457.
- [12] D. Harutyunyan, R. Behravesh, and N. Slamnik-Kriještorac, "Cost-efficient placement and scaling of 5g core network and mec-enabled application vnfs," in *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021, pp. 241–249.
- [13] T. Gao, X. Li, Y. Wu, W. Zou, S. Huang, M. Tornatore, and B. Mukherjee, "Cost-efficient vnf placement and scheduling in public cloud networks," *IEEE Transactions on Communications*, vol. 68, no. 8, pp. 4946–4959, 2020.
- [14] D. Harutyunyan, R. Fedrizzi, N. Shahriar, R. Boutaba, and R. Riggio, "Orchestrating end-to-end slices in 5g networks," in *2019 15th International Conference on Network and Service Management (CNSM)*, 2019, pp. 1–9.
- [15] J. Yusupov, A. Ksentini, G. Marchetto, and R. Sisto, "Multi-objective function splitting and placement of network slices in 5g mobile networks," in *2018 IEEE Conference on Standards for Communications and Networking (CSCN)*, 2018, pp. 1–6.
- [16] J. S. Pujol Roig, D. M. Gutierrez-Estevez, and D. Gündüz, "Management and orchestration of virtual network functions via deep reinforcement learning," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 2, pp. 304–317, 2020.
- [17] J. Pei, P. Hong, M. Pan, J. Liu, and J. Zhou, "Optimal vnf placement via deep reinforcement learning in sdn/nfv-enabled networks," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 2, pp. 263–278, 2020.
- [18] J. J. Alves Esteves, A. Boubendir, F. Guillemain, and P. Sens, "A heuristically assisted deep reinforcement learning approach for network slice placement," *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 4794–4806, 2022.
- [19] F. Rezazadeh, H. Chergui, L. Alonso, and C. Verikoukis, "Continuous multi-objective zero-touch network slicing via twin delayed ddpg and openai gym," in *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*, 2020, pp. 1–6.
- [20] A. Pasquali, V. Lomonaco, D. Bacci, and F. Paganelli, "Deep reinforcement learning for network slice placement and the deepnetslice toolkit," in *2024 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, 2024, pp. 76–81.
- [21] H. A. Kholidy, A. Karam, J. H. Reed, and Y. Elazzazi, "An experimental 5g testbed for secure network slicing evaluation," in *2022 IEEE Future Networks World Forum (FNWF)*, 2022.
- [22] N. Nikaein, M. K. Marina, S. Manickam, R. Knopp, C. Bonnet, and C. V. Imer, "Openairinterface: A flexible platform for 5g research," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 33–38, 2014.
- [23] M. Polese, L. Bonati, S. D'oro, S. Basagni, and T. Melodia, "Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 2, pp. 1376–1411, 2023.
- [24] 3GPP, "System architecture for the 5g system (5gs)," 3rd Generation Partnership Project (3GPP), Tech. Rep. TS 23.501, 2018. [Online]. Available: [https://www.etsi.org/deliver/etsi\\_ts/123500\\_123599/123501/15.03.00\\_60/ts\\_123501v150300p.pdf](https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/15.03.00_60/ts_123501v150300p.pdf)
- [25] ETSI, "Network functions virtualisation (nfv) release 3; evolution and ecosystem; report on network slicing support with etsi nfv architecture framework," European Telecommunications Standards Institute, Tech. Rep. ETSI GR NFV-EVE 012 V3.1.1, 2017. [Online]. Available: [https://www.etsi.org/deliver/etsi\\_gr/NFV-EVE/001\\_099/012/03.01.01\\_60/gr\\_nfv-eve012v030101p.pdf](https://www.etsi.org/deliver/etsi_gr/NFV-EVE/001_099/012/03.01.01_60/gr_nfv-eve012v030101p.pdf)
- [26] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, K. Prabhu, M. Ashley, A. Brown, A. Jaišle, S. Sahani, B. Simpson, and B. Tierney, "iperf3: A tcp, udp, and sctp network bandwidth measurement tool," 2025. [Online]. Available: <https://software.es.net/iperf/>
- [27] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev, "Reinforcement learning for combinatorial optimization: A survey," *arXiv preprint arXiv:2003.03600*, 2020.
- [28] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.
- [29] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.
- [30] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018. [Online]. Available: <http://incompleteideas.net/book/the-book-2nd.html>
- [31] D. Bertsekas and J. Tsitsiklis, "Neuro-dynamic programming: an overview," in *Proceedings of 1995 34th IEEE Conference on Decision and Control*, vol. 1, 1995, pp. 560–564 vol.1.
- [32] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
- [33] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.
- [34] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI)*, 2016.
- [35] The Tcpdump Group, "Tcpdump: A powerful command-line packet analyzer," <https://www.tcpdump.org/>.
- [36] A. W. Services, "Amazon ec2 on-demand pricing," 2025. [Online]. Available: <https://aws.amazon.com/ec2/pricing/on-demand/>
- [37] T. O. Atalay, D. Stojadinovic, A. Famili, A. Stavrou, and H. Wang, "A first look at 5g core deployments on public cloud: Performance evaluation of control and user planes," *arXiv preprint arXiv:2312.04833*, 2023. [Online]. Available: <https://arxiv.org/abs/2312.04833>
- [38] European Telecommunications Standards Institute, "Open Source MANO (OSM) User Guide," 2025. [Online]. Available: <https://osm.etsi.org/docs/user-guide/>
- [39] K. Documentation, "kube-scheduler," 2025. [Online]. Available: <https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/>
- [40] K. Abbas, T. A. Khan, M. Afaf, and W.-C. Song, "Network slice life-cycle management for 5g mobile networks: An intent-based networking approach," *IEEE Access*, vol. 9, pp. 80 128–80 146, 2021.
- [41] A. Esmaeil, K. Kralevska, and D. Gligoroski, "A cloud-based sdn/nfv testbed for end-to-end network slicing in 4g/5g," in *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, 2020, pp. 29–35.
- [42] C.-W. Liao, F. J. Lin, and Y. Sato, "Evaluating nfv-enabled network slicing for 5g core," in *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2020, pp. 401–404.
- [43] T. O. Atalay, D. Stojadinovic, A. Famili, A. Stavrou, and H. Wang, "Network-slice-as-a-service deployment cost assessment in an end-to-end 5g testbed," in *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, 2022, pp. 2056–2061.
- [44] T. O. Atalay, D. Stojadinovic, A. Stavrou, and H. Wang, "Scaling network slices with a 5g testbed: A resource consumption study," in *2022 IEEE Wireless Communications and Networking Conference (WCNC)*, 2022, pp. 2649–2654.