Master's Thesis

# Topology-Aware Service Function Chain Scheduling using DRL

Eui-Dong Jeong (정 의 동)

Department of Computer Science and Engineering

Pohang University of Science and Technology

2024

# 토폴리지 정보 기반의 심층강화학습을 활용한 Service Function Chain 스케줄링

# Topology-Aware Service Function Chain Scheduling using DRL

# Topology-Aware Service Function Chain Scheduling using DRL

by

Eui-Dong Jeong
Department of Computer Science and Engineering
Pohang University of Science and Technology

A thesis submitted to the faculty of the Pohang University of Science and Technology in partial fulfillment of the requirements for the degree of in the Computer Science and Engineering

Pohang, Korea
06. 14. 2024
Approved by
James Won-Ki Hong (Signature)
Academic advisor

# Topology-Aware Service Function Chain Scheduling using DRL

Eui-Dong Jeong

The undersigned have examined this thesis and hereby certify that it is worthy of acceptance for a master's degree from POSTECH

06. 14. 2024

| | | |
|---|---|---|
| Committee Chair | James Won-Ki Hong | (Seal) |
| Member | Young-Joo Suh | (Seal) |
| Member | Inseok Hwang | (Seal) |

MCSE 정 의 동 Eui-Dong Jeong
20222751
Topology-Aware Service Function Chain Scheduling using DRL.
토폴리지 정보 기반의 심층강화학습을 활용한 Service Function Chain 스케줄링.
Department of Computer Science and Engineering , 2024, 51p
Advisor : James Won-Ki Hong.
Text in English.

# ABSTRACT

Advancements in technology have led to increased demands on network infrastructure. To manage this flexibly and effectively, concepts such as Network Function Virtualization (NFV) and Service Function Chain (SFC) have emerged. However, without efficient operation, these can lead to high costs in terms of power consumption, service availability, Quality of Service (QoS). Previous studies have attempted to address these challenges using Deep Reinforcement Learning (DRL) and Graph Neural Networks (GNN). Nevertheless, they have not clearly defined the relationships between nodes in clusters and containers in services, nor proposed effective methods for embedding these relationships, resulting in limitations for efficient scheduling. Therefore, we propose a system that defines these relationships in the form of a graph and embeds them effectively. Moreover, previous research was limited to simulations

during the validation phase and did not involve actual network traffic collection, which restricted the ability to guarantee direct performance. In this study, we built a system to apply this in a real environment and, through actual performance validation, were able to reduce latency and power consumption by 5% while maintaining meaningful service availability.

# Contents

# List of Figures

# I. Introduction

With the advancement of technologies such as video streaming, metaverse, and deep learning, new requirements for networks have emerged. These requirements need to be met alongside the existing legacy demands while also catering to user-specific needs like security, high throughput. To achieve this, installing and operating new hardware devices is not cost-effective in terms of CAPEX and OPEX, and it also makes it difficult to provide flexible services to customers. Hence, the concepts of Network Function Virtualization (NFV) [2] and Service Function Chain (SFC) [3] have been introduced.

NFV is the concept of virtualizing network functions as software that can run on commercial computing devices, eliminating the need for physical installation of new hardware. Additionally, SFC involves grouping and managing these network functions and service functions in a flexible and efficient manner by chaining them together. Through these approaches, we have gained flexibility, but the complexity in terms of distributing and operating these functions has significantly increased. Therefore, to handle this complexity, a system-wide orchestrator is needed, and by applying appropriate SFCs to traffic, efficient service delivery becomes possible. However, there are several limitations in their operation. Specifically, it is challenging to simultaneously reduce power consumption, improve Quality of Service (QoS), and enhance service availability. To address this, various studies have conducted container-level scheduling, which means determining where to deploy containers in a multi-node cluster environment. Nevertheless, there are still limitations in operating this effectively.

Notably, there are three main limitations: first, it is difficult to validate the feasibility in a real operational environment; second, the relationships between nodes, services, and associated containers in the form of graphs are not adequately represented; and third, the problem's complexity converges to NP-Hard, making it unsolvable as

the number of nodes and containers increases. Thus, attempting to resolve this with existing rule-based methods presents challenges.

In response, this thesis sets up the actual operational environment using Kubernetes, overcoming the first limitation of not considering the real operational environment through the implementation and experimentation of SFC. And, we proposes a monitoring system that simultaneously collects the information and relationships of nodes, services, and associated containers represented in graphs. We also use Graph Neural Network (GNN) [4] architecture to process this kind of graph data. Finally, it presents a Deep Reinforcement Learning (DRL) [5] algorithm for effective container scheduling based on this architecture. In this theis, we utilized off-policy DRL algorithm like DQN [6], TD3 [7], SAC [8]. And for fast convergence, we also apply the Multi-Agent DRL (MADRL) [9] concept. As a result, we achieved a performance improvement in terms of latency and power consumption of 5% compared to the existing kube-scheduler and the previous study, NetMARKS [10], and made the entire experimental environment and development code publicly available to serve as a baseline for future research.

# II. Background

## 2.1 Network Function Virtualization (NFV)

Traditional hardware-based network management has a critical issue of lacking flexibility. Additionally, the operation of such systems is inefficient and hard to automate, posing a risk of human error. To address these problems, Network Function Virtualization (NFV) [2] was proposed, which involves changing network functions from hardware to software and managing them through commercial server equipment. This makes it easier to create, delete, and update functions compared to hardware devices. The software-based network functions are referred to as Virtualized Network Functions (VNFs) [11], and they are deployed and managed in units of virtual machines or containers, aiding in the design of flexible network structures. However, VNFs alone do not adequately address topology dependency, configuration complexity, and elastic service delivery [12].

To solve these issues, the concept of Service Function Chain (SFC) [3] was introduced. SFC helps manage and operate VNFs by grouping them into chains. This approach facilitates the identification of dependencies between services and reduces configuration complexity [3]. With SFC, we could easily build and manage the relationships among various network functions. However, SFC alone struggles to handle complex functions such as ensuring system stability, guaranteeing QoS and availability. Therefore, an orchestrator is essential for managing and operating SFCs. Accordingly, commercial services such as OpenStack [13] and Kubernetes [14] are being used as platforms for SFC management to operate container or VM-based systems.

## 2.2 Kubernetes

With the advent of microservices architecture, the deployment and management environment of applications has become increasingly complex. Microservices enhance flexibility and collaboration by decomposing the functions of monolithic applications into multiple services, but this also results in an increase in management points. Therefore, the need for tools to efficiently manage this complexity became apparent, leading to the emergence of Kubernetes [14]. Kubernetes provides various container orchestration features such as load balancing, deployment, scaling, and rolling updates to manage microservices deployed in containers. It focuses on simplifying the automatic deployment, scaling, and operation of applications.

Kubernetes aims to optimize the deployment environment of containerized applications by structuring each element of the application into units called **pods**. A pod includes containers and all necessary resources for their operation, such as networks, volumes, and even sidecar containers that support the main container. Kubernetes defines the pod as the smallest unit of orchestration.

In Kubernetes, the **deployment** unit is used to manage workloads by facilitating easy scaling and volume control of pods. This allows for easy control of replicas of specific pods. Additionally, the **service** unit provides service discovery and load balancing for traffic directed at pods. In this thesis, each service function is defined as a pair of Kubernetes service and deployment unit. Consequently, our scheduling algorithm is performed in an environment where Pods supporting various services are distributed across multiple nodes in a Kubernetes cluster.

By default, Kubernetes includes a built-in scheduler called **kube-scheduler**, which determines the appropriate node for pod placement when a pod installation request occurs. The kube-scheduler performs pod scheduling through various processes as illustrated in Figure 2.1. Among these, filtering and scoring play an important role. Filtering identifies nodes where scheduling is not possible, and scoring selects the optimal node. To execute these processes, the kube-scheduler references resource re-

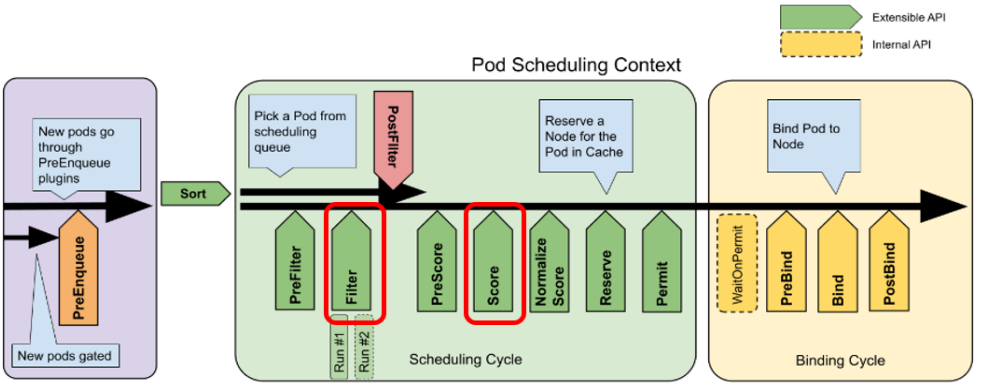


Figure 2.1: Kubernetes Scheduling Process [1]

quests, limits, and usage metrics from the nodes, and pods. However, the resource information is limited to CPU, memory, and disk. Therefore, it lacks consideration for factors such as power consumption and network usage. This is because the goal of Kubernetes is to maximize availability rather than optimize service performance. For example, Kubernetes includes a configuration that distributes the deployment of replicas for the same service to as many nodes as possible. Nonetheless, Kubernetes provides manuals for defining custom resources, modifying scheduling, or adding plugins. In practice, several telecommunications companies (Huawei, Nokia, Ericsson, etc.) discuss and utilize Kubernetes for the deployment and operation of network functions [15].

In this research, Kubernetes is used to provide a stable and automated environment for managing the operation of SFCs. So, we can focus on researching and implementing a SFC scheduler that considers power consumption and QoS when performing SFC operations.

## 2.3 SFC Scheduling

Currently, NFV and SFC are deployed in the form of virtual machines or containers. While this has enabled flexible network operations, it has also introduced the challenge of increased management complexity. To meet the diverse requirements of various services such as video services, IoT, and deep learning, multiple network functions are needed. However, there are limitations in efficiently deploying and managing these functions. Research has been conducted on NFV Orchestrators (NFVO) to address these limitations. There are examples of building such NFV environments using OpenStack and Kubernetes [16, 17].

One of the goals of these orchestrators is to deploy network functions in the optimal node location within a cluster, based on the characteristics of VNFs and SFCs, to ensure power consumption, QoS, and service availability. Much research has been conducted on aspects such as optimizing SFC deployment (see Chapter III). However, as the number of nodes and the services to be managed increase, there are limitations in performing these tasks promptly. Because, SFC scheduling can be represented as a Knapsack Problem (KP) [18], which is an optimization problem where the goal is to distribute items with given values into a limited-capacity bag to maximize the total value. This problem itself is an NP-Hard problem, and SFC scheduling that has more properties compared with KP is much more complex. First, it has multi-objective nature, involving power consumption, QoS, and availability. Also, the evaluation criteria must consider various resources such as CPU, memory, bandwidth, and disk I/O, making it a multi-dimensional problem. Moreover, instead of optimizing placement for a single node, it requires considering the optimal placement across multiple nodes, thus becoming a Multiple-Objective Multiple-Dimensional Multiple-Knapsack Problem (MOMDMKP). Solving this problem through exhaustive search would involve the following complexity.

$$O(N^P) \begin{cases} N \text{ is number of nodes} \\ P \text{ is number of pods} \end{cases} \tag{2.1}$$

Additionally, real-world environments have an important limitation. It is the time. The time required to install new network features, the time it takes for those features to start booting, traffic handling, and metric impact all impact the time needed to determine optimal placement. Therefore, we have to wait for this amount of time to evaluate the algorithm. Moreover, network traffic is constantly changing. These limitations make the design of optimization algorithm more complex.

## 2.4 Graph Neural Network (GNN)

Graph Neural Network (GNN) [4] utilizes neural networks to perform inference and analysis on data structured as graphs. The core of GNN is to effectively transfer the characteristics of vertices and edges in a graph into an embedding space, leveraging these embeddings to restructure and solve problems. (In this thesis and Kubernetes context, computing servers in a cluster are referred to as nodes, so to avoid confusion, graph nodes are referred to as vertices.) For instance, GNN can aggregate information to vertices by embedding all necessary details into the vertex or by gathering information via edges. This allows us to gain insights and solutions through vertex embeddings. Generally, GNN algorithms can be expressed by Equation 2.2. To obtain the embedded result of a specific vertex ($\mathbf{x}'_i$), the vertex ($\mathbf{x}_i$), its adjacent vertices ($\mathbf{x}_j \in \mathcal{N}(i)$), and the connecting edges ($e_{j,i}$) are processed using a specific algorithm ($\phi_\Theta$), aggregated ($\bigoplus$), and then processed again ($\gamma_\Theta$).

$$\mathbf{x}'_i = \gamma_\Theta \left( \mathbf{x}_i, \bigoplus_{j \in \mathcal{N}(i)} \phi_\Theta(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{j,i}) \right) \tag{2.2}$$

Graph Convolutional Networks (GCN) [19] are influenced by Convolutional Neural Networks (CNN). They perform matrix multiplication with the same parameters

applied to neighboring vertices to update the value of a vertex, aggregating these values to re-embed the vertex. This method maximizes data efficiency through parameter sharing rather than performing embeddings separately for each node, mitigating issues like overfitting. Additionally, GCN applies penalties to vertices with many neighbors (high degree), reducing bias towards specific vertices and improving overall performance.

$$\mathbf{x}'_i = \mathbf{\Theta}^\top \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j \tag{2.3}$$

Graph SAmple and aggreGatE (GraphSAGE) [20] simplifies large-scale computations by sampling neighboring vertices instead of considering all features of vertices and edges. The simplest method involves calculating the mean of neighboring vertices and using only this for embedding. This approach is easy to implement and advantageous for efficiently evaluating performance.

$$\mathbf{x}'_i = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \cdot \text{mean}_{j \in \mathcal{N}(i)} \mathbf{x}_j \tag{2.4}$$

Graph Attention Network (GAT) [21] addresses the limitation of previous models that performed unweighted aggregation by treating all neighboring vertices as equally important. GAT assigns higher weights to more important vertices using an attention algorithm, significantly improving performance. The parallel processing capability of the attention algorithm also enhances performance and offers flexible configuration. The structure can be formulated as follows.

$$\mathbf{x}'_i = \alpha_{i,i} \mathbf{\Theta}_s \mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j} \mathbf{\Theta}_t \mathbf{x}_j \tag{2.5}$$

$$\alpha_{i,j} = \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{a}_s^\top \mathbf{\Theta}_s \mathbf{x}_i + \mathbf{a}_t^\top \mathbf{\Theta}_t \mathbf{x}_j\right)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\text{LeakyReLU}\left(\mathbf{a}_s^\top \mathbf{\Theta}_s \mathbf{x}_i + \mathbf{a}_t^\top \mathbf{\Theta}_t \mathbf{x}_k\right)\right)} \tag{2.6}$$

Building upon GAT, a new perspective was introduced in a more advanced direction [22]. It pointed out that calculating independent attention scores for each vertex

could lead to unstable learning due to high variance, and integrated weights to ensure more stable convergence. This architecture is beneficial for stable learning.

$$\mathbf{x}_i' = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{i,j} \mathbf{\Theta}_t \mathbf{x}_j \tag{2.7}$$

$$\boldsymbol{\alpha}_{i,j} = \frac{\exp\left(\mathbf{a}^\top \text{LeakyReLU}\left(\mathbf{\Theta}_s \mathbf{x}_i + \mathbf{\Theta}_t \mathbf{x}_j\right)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\mathbf{a}^\top \text{LeakyReLU}\left(\mathbf{\Theta}_s \mathbf{x}_i + \mathbf{\Theta}_t \mathbf{x}_k\right)\right)} \tag{2.8}$$

In this thesis, the models were constructed and applied to experiments using GraphSAGE, and GAT for embedding node, service, and associated pod metrics.

## 2.5 Deep Reinforcement Learning (DRL)

Traditional rule-based methods have taken a method of replacing expert knowledge with a rule in a specific state and applying it to the system. However, there are limitations to the method due to three problems. First, it is because system complexity is increasing. As there are more matters to consider depending on the problem to be solved, a simple rule cannot optimize it. Second, it is an insufficient generalization performance. If the problem can be simplified through generalization, the problem can be easily solved, but generalization is becoming impossible as the problem becomes more complex. Finally, it relies on expertise. Relying on a specific expert's rule not only makes flexible design impossible, but also causes more cost and system-specific gaps. Therefore, reinforcement learning has emerged to solve this problem.

Reinforcement learning involves an agent interacting with an environment to learn the optimal policy. The agent observes the state, selects an action, and receives a reward. This process is modeled as a Markov Decision Process (MDP), where the reinforcement learning agent aims to learn a policy that maximizes cumulative rewards within the modeled system.

MDP is based on the Markov assumption, which states that the future state at time t+1 is independent of all previous states, given the current state at time t. In other

words, it simplifies the prediction of the next state by considering only the immediate prior state. For a first-order Markov assumption, this can be expressed mathematically as shown in Equation 2.9.

$$P(S_{t+1}|S_t, S_{t-1}, \cdots, S_1, S_0) = P(S_{t+1}|S_t) \tag{2.9}$$

If the prior state sufficiently captures information from earlier states, reinforcement learning can abstract complex problems more easily. In such a model, the optimal action based on the reward in each state is defined as the policy, overcoming the limitations of rule-based methods.

Although MDP simplifies the real environment into a lightweight model, it can still be excessively large. Furthermore, there are temporal and spatial limitations in collecting data to cover all possible scenarios. Deep Reinforcement Learning (DRL) [5] addresses this by using deep learning to estimate policies or values. In our system, data is represented as graphs, and as the number of nodes, pods, and services increases, complexity grows exponentially. Therefore, we propose using DRL for estimation.

DRL initially focused on estimating values in the Q-Learning process and later evolved towards learning policies. Various algorithms have been developed as a result. The problem we aim to solve—optimal scheduling in a cluster—faces limitations in data collection during the learning process. Hence, this study prefers off-policy learning algorithms that enhance data efficiency. Specifically, we conducted experiments on Twin Delayed Deep Deterministic Policy Gradient (TD3)[7] and Soft Actor Critic (SAC)[8], both of which evolved from Deep Deterministic Policy Gradient (DDPG) [23].

Deep Deterministic Policy Gradient (DDPG) combines Deterministic Policy Gradient and Deep Q Network. It enhances learning stability by separating the target network and behavior network in the traditional Actor-Critic method and applying an off-policy algorithm based on replay memory.

Twin Delayed Deep Deterministic Policy Gradient (TD3) was proposed to address biases and stability issues in the Q value learning process of DDPG. TD3 in-

troduces Q value clipping, delays actor updates, and increases the frequency of critic updates. As an off-policy algorithm, TD3 exhibits high data efficiency and stable performance across various environments through multiple stabilization techniques.

Soft Actor Critic (SAC) overcomes the limitations of deterministic policies in DDPG and TD3 by applying a stochastic policy while using maximum entropy method to construct the objective function, ensuring sufficient exploration. SAC excels in complex tasks, broadening the search space for optimal policies. It maintains high data efficiency as an off-policy algorithm and shows strong performance in tasks where exploration is crucial.

This thesis implements and experiments with TD3 and SAC algorithms.

# III. Related Work

This study aims to develop a system and algorithm for scheduling SFCs managed in a Kubernetes cluster while considering power consumption, QoS, and service availability simultaneously. Given the NP-Hard nature of this problem, we intend to integrate GNN and DRL to approximate optimal solutions efficiently.

Previous research has attempted to address similar issues. The first group of studies proposed **utilizing network variables** in the scheduling process. Łukasz et al. [10] demonstrated that considering network traffic for SFC scheduling in Kubernetes cluster could result in better QoS, especially in terms of latency. They collected bandwidth information between network functions and enhanced performance by adding a plugin to the kube-scheduler. Their experiments were conducted in an actual cluster environment, lending high reliability to their findings. However, their research only used bandwidth to determine service dependencies, failing to utilize other metrics effectively and not addressing power consumption.

Yu et al. [24] aimed to minimize SFC delay in heterogeneous edge networks by optimizing scheduling with bandwidth and latency as network variables. They proposed the JOS algorithm, which proved effective in simulation environments but did not consider CPU or memory usage.

Angelo Marchese and Orazio Tomarchio [25] proposed a scheduling approach for the Cloud-to-Edge continuum, emphasizing the application workflow's DAG structure and considering network bandwidth and cost. Their algorithm, implemented in Kubernetes, outperformed the kube-scheduler in response time. However, the algorithm was limited in reflecting complex network usage by merely integrating traffic amounts per node.

Zeyuan et al. [26] categorized applications into latency-sensitive, compute-intensive, and data-intensive types, applying different scheduling policies for each. Their ap-

proach, implemented in Kubernetes, showed better performance than the kube-scheduler in computing, I/O, and overall system environment. However, they did not account for service dependencies and faced limitations in complex applications involving multiple categories.

Michael et al. [27] proposed optimal placement of IoT applications in edge environments by considering a combination of physical, operational, and network parameters. Using CPU, memory, latency, jitter, and packet loss for scheduling, they demonstrated better resource scheduling than the kube-scheduler and showed stable cluster temperature averages at specific points. All these studies demonstrated that incorporating network variables leads to more efficient latency management and power consideration compared to the kube-scheduler, but none provided a comprehensive optimization of complex scheduling algorithms.

Several studies have applied **reinforcement learning to address these complex scheduling problems**. John Rothman and Javad Chamanara [28] used Double Deep Q-Network (DDQN) with Prioritized Experience Replay (PER) Memory to maximize resource usage and energy efficiency. They structured their reinforcement learning model using CPU, memory, disk, and bandwidth information, with average fragmentation score as the reward. Their approach, integrating multiple policies into the kube-scheduler, proved effective in real Kubernetes environments, though it did not consider dependencies between nodes, services, and associated containers.

Yamming et al. [29] applied DRL for reliable SFC scheduling in multi-domain networks, focusing on maximizing service reliability using CPU, memory, disk, and bandwidth. However, their optimization did not consider power consumtion or QoS, and standard DRL struggled to identify dependencies among nodes, services, and associated containers.

Lei et al. [30] used DRL to optimize reliability and resource usage in SFC scheduling, embedding SFC information and dependencies with CNN. Their DQN-based approach outperformed rule-based algorithms but did not use resource information in state input and was only validated in simulation.

Research combining network variables and reinforcement learning to improve state estimation using **GNN** has shown promise. Siyu et al. [31] aimed to optimize power efficiency and resource allocation in SFC scheduling using GNN and DRL. They critiqued previous works for interpreting SFCs as linear graphs, proposing effective embedding with GNN and demonstrating superior performance. However, they did not validate the approach in real environments and focused only on service dependencies.

Meilin et al. [32] applied GNN-based DRL to optimize E2E latency in satellite-terrestrial networks, proposing a GCN-based A3C architecture. They concentrated on latency-related network information, showing high E2E latency performance but lacked consideration for CPU, memory, and disk usage and focused only on service dependencies.

Chengfeng et al. [33] proposed a GNN and RL-based scheduler for collaborative scheduling across multiple edges. Using a simple GNN and DQN, they successfully minimized error rates and latency but did not effectively represent dependencies among nodes, services, and associated containers and were only validated in simulation.

As shown in Figure 3.1, these studies have proposed algorithms to optimize container scheduling for various applications, demonstrating that using network variables can enhance performance in power saving, QoS, or service availability. They also showed that DRL could effectively operate in container scheduling environments and that GNN could improve performance by considering service dependencies. However, attempts to structure graphs using heterogeneous information about nodes, services, and associated containers are limited, and reinforcement learning algorithms still face challenges. Additionally, these studies have been restrictive in sharing their cluster environments and algorithms, limiting improvements.

In this thesis, we propose a structure that embeds relationships among network and system elements, effectively embedding them through GNN. We conduct experiments with various reinforcement learning algorithms based on this embedding and

make the entire process publicly available on GitHub [34], hoping to aid future research.

| Paper | Year | Network Variable | RL | GNN | Application | Orchestration tool | Environment | Variables (Input) | Metrics (Evaluation) | Code Accessibility |
|---|---|---|---|---|---|---|---|---|---|---|
| **[10]** | 2021 | O | X | X | SFC | K8s | Private Cluster (5 nodes) | Number of bytes | Response time, bandwidth | X |
| **[24]** | 2021 | O | X | X | SFC | X | Simulation | Bandwidth, Computing (do not specified CPU, GPU or Memory) | Time Complexity, Latency | X |
| **[25]** | 2023 | O | X | X | DAG form microservice (Web service) | K8s | Private Cluster | CPU, Memory, bandwidth, network cost | latency | X |
| **[26]** | 2023 | O | X | X | 3 resource intensive service (CPU, Network, Disk) | K8s | Private Cluster (12 nodes) | CPU, Disk, latency | Latency, EPS, read speed, write speed, OPS | X |
| **[27]** | 2020 | O | X | X | Flask web application | K8s | 4 Raspberry Pi | CPU, Memory, latency, jitter, packet loss | Scheduling time, CPU temperature, Memory usage, latency | X |
| **[28]** | 2023 | O | O | X | 40 Pods (CPU-intensive and other type) | K8s | Private Cluster (15 nodes) | CPU, disk, Memory, bandwidth, latency | Latency, machine number | X |
| **[29]** | 2023 | O | O | X | SFC | X | CERNET2 (Simulation) | CPU, Memory, Storage, bandwidth | Deployment Success Ratio, Deployment Resource Cost | X |
| **[30]** | 2022 | O | O | △ (CNN) | SFC | X | Simulation | Completion time | Deployment success Rate, E2E latency | X |
| **[31]** | 2021 | O | O | O | SFC | X | Internet2 (12 nodes) | CPU, delay, Energy Consumption, bandwidth | Acception Rate, server num, E2E latency, | X |
| **[32]** | 2023 | O | O | O | SFC | X | Simulation | CPU, Memory, bandwidth, link distance, total VNF num | Latency, Acceptance Rate, Profit | X |
| **[33]** | 2024 | O | O | O | Factory-Application | X | Factory (Simulation) | Operation status, completion time, waiting time, r emaining time | Error rate, latency | X |
| **Ours** | 2024 | O | O | O | SFC | K8S | Private Cluster (4 nodes) | Bandwidth, Latency, CPU, Memory usage | Energy Consumption, E2E Latency, Service Availability | O |

Figure 3.1: Comparison of Related Work

# IV. Design

This study aims to implement an SFC scheduling algorithm that minimizes power consumption and service latency while ensuring service availability. As illustrated in Figure 4.1, the goal is to find the optimal location for a pod when a Kubernetes cluster user requests to add a replica of a specific service. This process is designed to be executed in nine steps:

1. The user sends a request to add a new pod.
2. The Kubernetes cluster requests the scheduler to find a node to place the pod.
3. The scheduler requests scores to DRL model implemented in our system. These scores will be assigned to nodes to find the optimal one for the pod placement.
4. The DRL model requests cluster metric information from the monitoring system to select the node.
5. The monitoring system collects the required information.
6. The monitoring metrics are delivered to the DRL model.
7. The DRL model selects the optimal node and provides the score to the Kubernetes scheduler.
8. The Kubernetes scheduler communicates this information to the Kubernetes cluster.
9. Finally, the Kubernetes cluster binds the pod to the selected node.

To achieve this, we developed two key components. First, we integrated and developed a total of six modules to build a monitoring system that can collect metrics

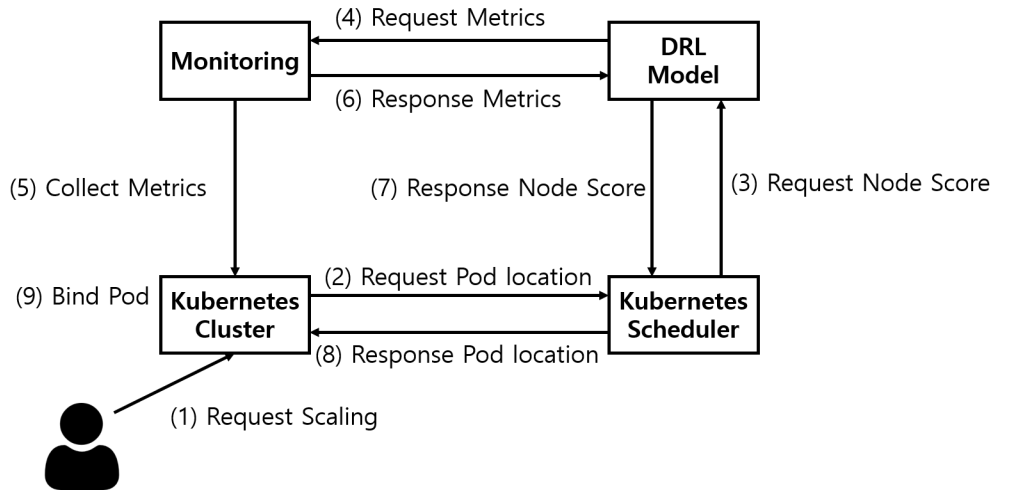


Figure 4.1: Overall System Architecture

from nodes, services, and associated pods. Second, we constructed a DRL model and a corresponding training environment. To efficiently utilize graph-structured data in this process, we employed a GNN-based feature extractor.

## 4.1 Monitoring System

To collect metric information in a Kubernetes environment, we used Prometheus [35]. Prometheus provides both a monitoring system and a time series database, allowing real-time monitoring of system information. It can collect information obtained by separate modules through dedicated Prometheus Exporters, providing an aggregated view of the data. As shown in Figure 4.2, we used or implemented a total of six modules to collect information on nodes, services, and associated pods.

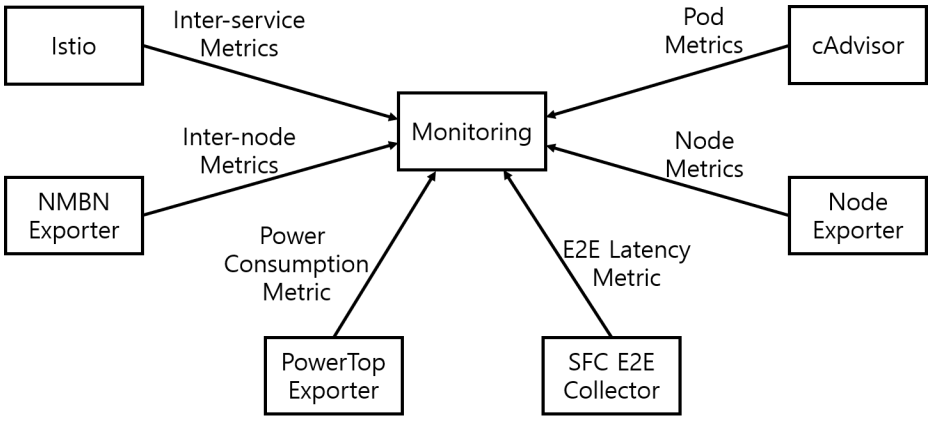


Figure 4.2: Monitoring Overall Architecture

### 4.1.1 Node Exporter

Node Exporter [36] is a tool developed by Prometheus that provides metrics for individual nodes. It compiles metrics collected using open-source tools such as netstat, hwmon, and meminfo at the node level. In Kubernetes, a **daemonSet** can be used to deploy a specific image-based pod on all nodes simultaneously, enabling the deployment of Node Exporter on all nodes and centralized management by Prometheus. In this study, we utilized metrics provided by Node Exporter, including CPU and memory usage, and receive and transmit bytes of nodes, using their one-minute averages (CPU, Memory) and cumulative values (receive, transimit bytes) (See Figure 4.3).

### 4.1.2 container Advisor (cAdvisor)

container Advisor (cAdvisor) [37] is an open-source tool developed by Google for monitoring the status of containers running in a cluster. cAdvisor runs on the host system, collecting container information from the control group (cgroup) data, accessible at the kernel level. In this study, we utilized metrics provided by cAdvisor, including one-minute averages (CPU, Memory) and cumulative values (receive, transimit bytes) of containers (See Figure 4.3). As our study uses pods as the minimum

unit, we estimated pod metric value based on the cumulative value of included containers. Moreover, since methods to collect metrics at the service level are limited, we estimated service-level values by summing the metrics of the pods constituting the service. Using cAdvisor metrics, we estimated the CPU and memory usage, and collected receive and transmit bytes at the pod and service levels.

### 4.1.3 Network Metric Between Nodes Exporter (NMBN Exporter)

The Network Metric Between Nodes Exporter (NMBN Exporter) is a monitoring system proposed in this thesis to monitor network traffic between nodes. Previous studies did not collect metrics to determine dependencies between nodes, but in this thesis, we implemented a system to measure latency using ICMP traffic-based ping [38] and to obtain receive and transmit bytes information between nodes using iptables [39]. The system collects communication information among all nodes in the cluster, as shown in Figure 4.5. In this study, we utilized one-minute cumulative values of receive bytes, transmit bytes, and ping latency between nodes provided by NMBN Exporter (See Figure 4.4).

Detailed implementation of the actual system can be accessed on Github [40].

### 4.1.4 Istio

Istio [41] leverages service mesh [42] architecture to provide network observability, traffic management, security, and policy enforcement. In this thesis, service functions were deployed based on the service mesh architecture and grouped into Kubernetes deployments. This allows for easy scaling, such as adding and removing replicas as needed. In such an environment, Istio observes network traffic between services and provides information on HTTP requests, responses, and duration. This information was used in the thesis to collect one-minute cumulative values of dependencies between services (See Figure 4.4).

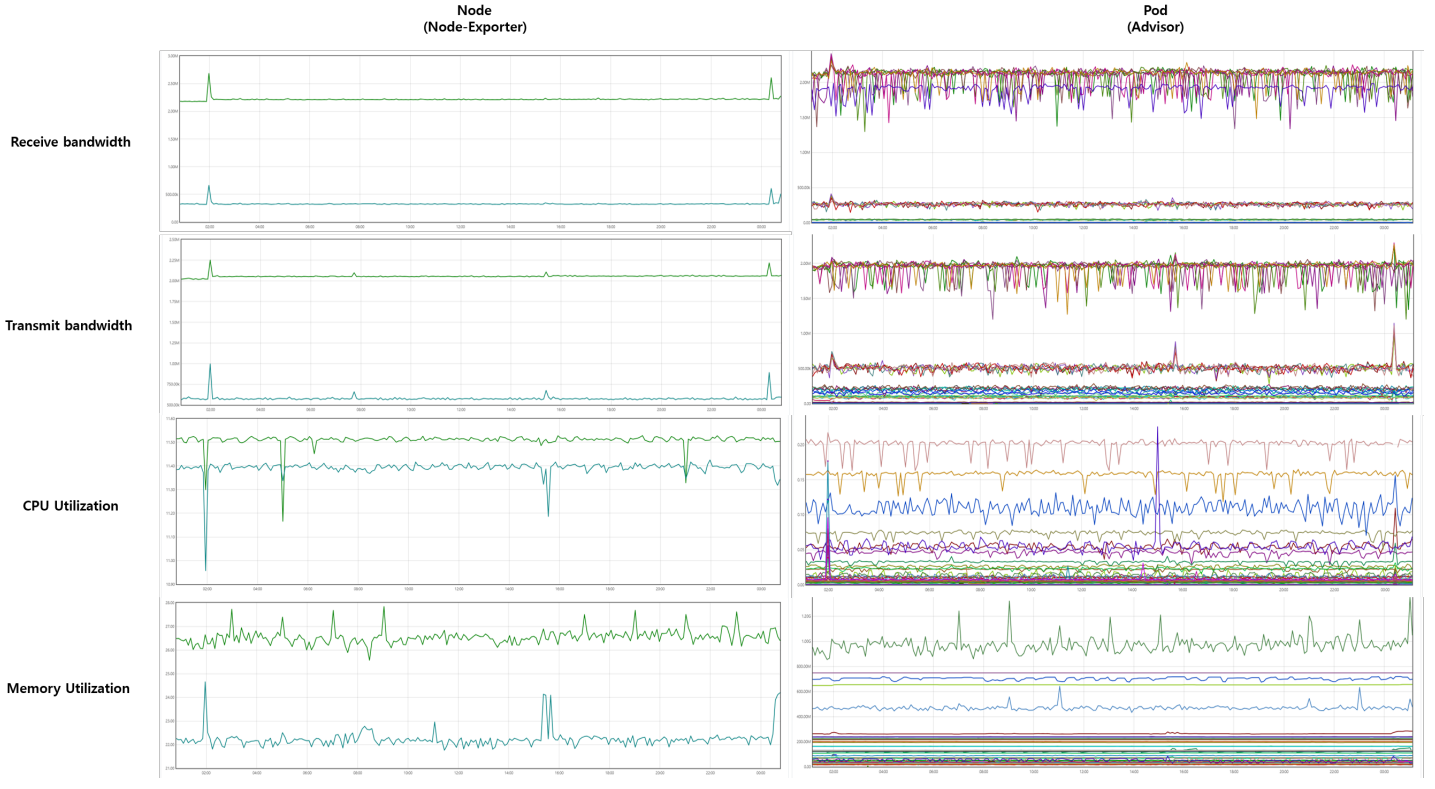


Figure 4.3: Node Exporter & cAdvisor Metrics

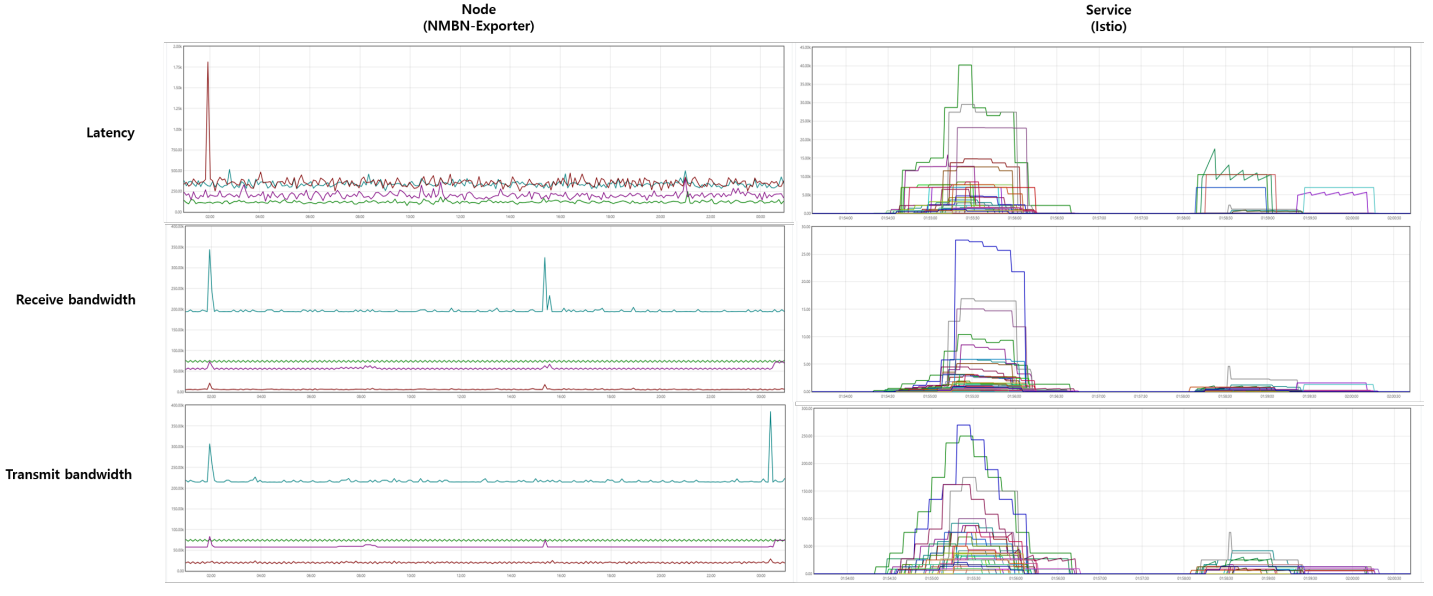


Figure 4.4: NMBN Exporter & Istio Metrics

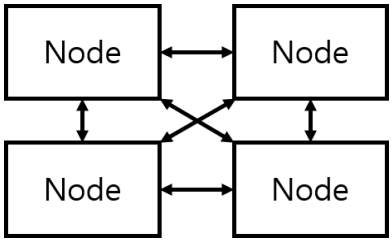


Figure 4.5: Dependencies between Nodes

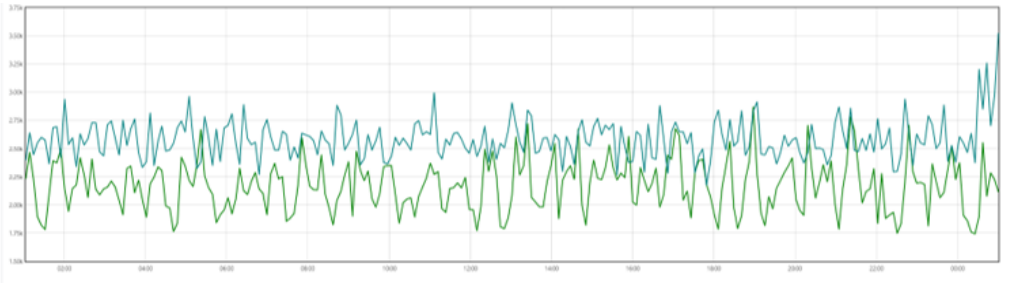

Figure 4.6: PowerTop Exporter Metrics

### 4.1.5 PowerTop Exporter

The PowerTop Exporter is implemented using the open-source PowerTop Monitoring [43], deployed on nodes via Kubernetes daemonSet. PowerTop Monitoring uses the PowerTOP [44] tool to calculate the predicted power consumption of a node. In this study, we utilized one-minute cumulative values of overall power consumption from the node provided by PowerTop Monitoring (See Figure 4.6).

### 4.1.6 SFC E2E Collector

The SFC E2E Collector is a monitoring system proposed in this thesis to send SFC requests and collect E2E latency metrics for these requests. As shown in Figure 4.7, SFC requests sent through this module are recorded in the system, and latency information can be retrieved. When a user defines an SFC path via the **/start** API request, the SFC E2E Collector transmits this information to the VNF service. Once

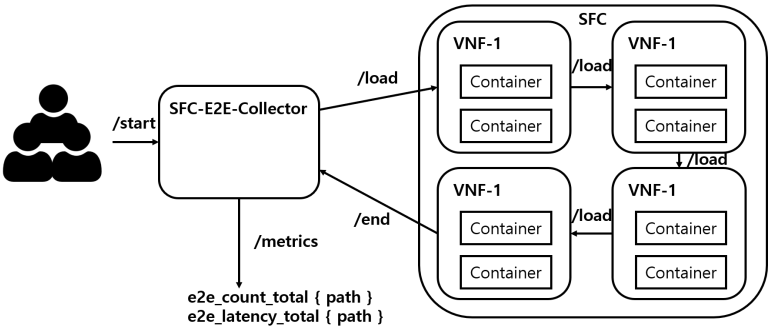


Figure 4.7: SFC E2E Collector Architecture

processing is complete, the latency is calculated and recorded in the database.

Detailed implementation of the actual system can be accessed on Github [45].

## 4.2 DRL Algorithm

We utilized DRL to find a method that minimizes power consumption and service latency while maximizing service availability, using data collected through a monitoring system. Figure 4.8 illustrates the overall structure of the DRL model. We employed GNN to embed the state information, structured as a graph, into vectors, which were then used to construct layers estimating the policy and action values for the DRL model, outputting node probabilities. For GNN algorithms, GAT was used when edge feature information (e.g., latency) was available, and GraphSAGE or GAT was used when there was no edge information. Linear layers were used to estimate the policy and value, and for the DRL algorithm, we opted for off-policy algorithms like TD3 and SAC to optimize sample efficiency. Detailed implementation of the actual system can be accessed on Github [46].

### 4.2.1 Markov Decision Process Modeling

To begin the training, the Kubernetes cluster environment, representing the real world, was modeled as a Markov Chain. Consequently, we defined the state, action,

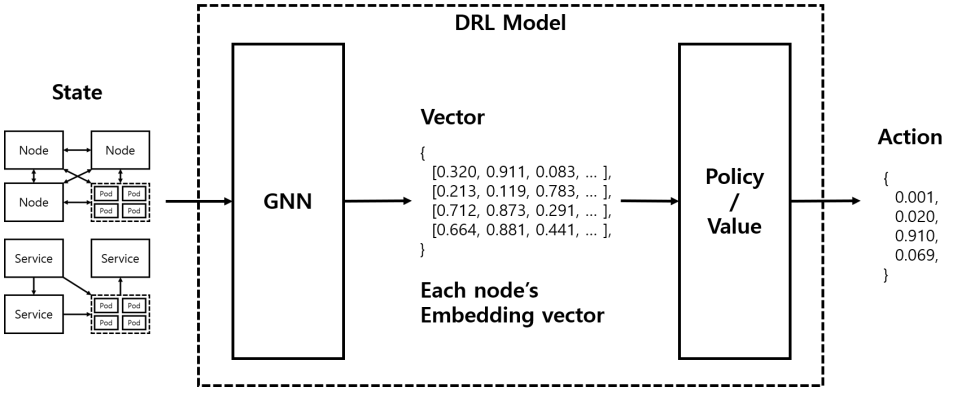


Figure 4.8: DRL Model Architecture

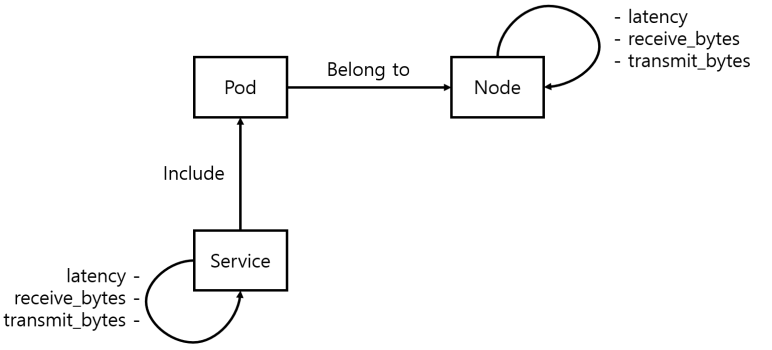


Figure 4.9: State Graph

and reward as follows:

- **state**: As shown in Figure 4.9, we structured the data as a graph. Vertices of nodes, pods, and services all have CPU, memory utilization, receive bytes, and transmit bytes information. Edges between nodes also gathering latency, receive bytes, and transmit bytes data. Edges between services collect the same information. Additionally, the graph's edges connect nodes with installed pods and services with contained pods, structuring the state as a graph.

$$
\begin{aligned}
&\textbf{State} = \{ \\
&\quad \text{Each Node's Information}, \\
&\quad \text{Each Pod's Information}, \\
&\quad \text{Each Service's Information}, \\
&\quad \text{Each inter Node edge's Information}, \\
&\quad \text{Each inter Service edge's Information}, \\
&\quad \text{Each Service to Pod edge's Information}, \\
&\quad \text{Each Pod to Node edge's Information}, \\
&\} \qquad (4.1)
\end{aligned}
$$

- **action**: This paper aims to customize the Kubernetes scheduler for optimal placement. Therefore, we calculate and output scores for each node.

$$\textbf{Action} = \{\text{Each Node's Score}\} \qquad (4.2)$$

- **reward**: Rewards are based on the average power consumption collected from each node, the average E2E latency occurring during SFC processing, and the average service availability. Power consumption and SFC E2E latency were directly measured from monitoring system, while service availability was based on the dispersion of failure points. The goal is to maximize service availability while minimizing power consumption and SFC E2E latency, structured as shown in Equation 4.3. Previous studies used weighted averages, requiring hyperparameter tuning, a limitation this study addresses by proposing a multiplication-based reward system. Service availability, stable between 0 and

1, becomes the numerator, while power consumption and SFC E2E latency, with wider value ranges, are the denominator. This structure required the use of a minus reward system.

$$\textbf{Reward} = -\frac{\text{PC}_{\text{avg}} \times \text{SEL}_{\text{avg}}}{\text{SA}_{\text{avg}}} \tag{4.3}$$

$\text{PC}_{\text{avg}}$ = Average Power Consumption

$\text{SEL}_{\text{avg}}$ = Average SFC E2E Latency

$\text{SA}_{\text{avg}}$ = Average Service Availability = $\frac{1}{|S|}\sum_{s \in S}\frac{|N_s|}{|N|}$

$N$ = Set of Nodes

$S$ = Set of Services

$N_s$ = Set of Nodes that include at least one pod belong to the Service $s$

### 4.2.2 DRL Model Detail

In our proposed system, the action space corresponds to the number of nodes, with the goal of obtaining a score for each node. Although we could use a single agent to obtain outputs for all nodes, this approach is inefficient for learning. As the number of nodes increases, more data is required during the learning process, making it difficult to apply the algorithm in a cluster environment where nodes change frequently. Therefore, as shown in Figure 4.10, we designed each agent to produce only one node's score. Each agent competes based on its score, and only the node with the highest score is selected. This approach mimics the effect of Multi-Agent Reinforcement Learning (MARL). Using this method, we can obtain the outputs of multiple agents corresponding to the number of nodes in a single inference of the DRL model. This allows us to acquire more samples even with fewer transitions. By leveraging this

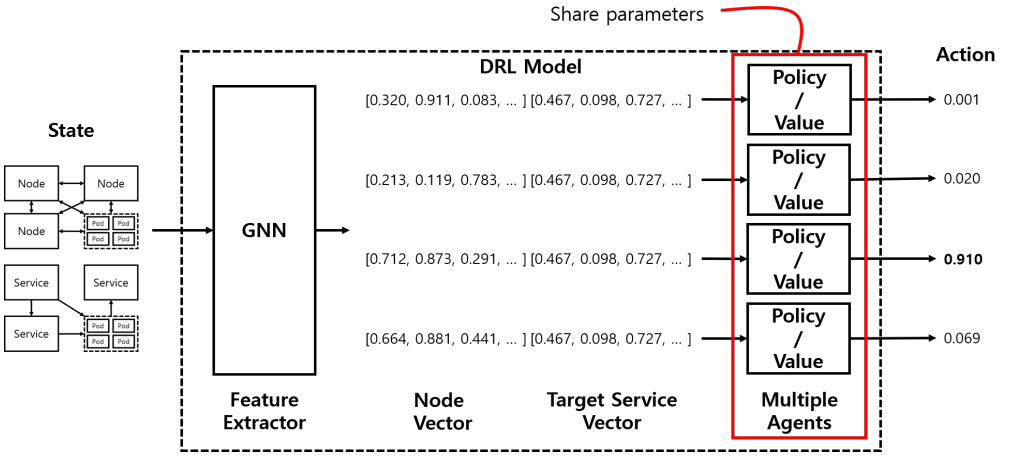


Figure 4.10: Multi-Agent DRL Architecture

in the DRL model's update process, we can use the number of samples obtained from a mini-batch multiplied by the number of nodes for training. This enhances learning stability and ensures efficient training.

Each agent's input is a combination of the embedding vector for each node, processed through a GNN, and the embedding of the service to which the new pod belongs. This combined input helps calculate a score that matches the service with the appropriate node.

### 4.2.3 Training

We trained the DRL model in the previously discussed MC modeling environment. The training environment was built based on actual Kubernetes environment data, not a simulation. To abstract this into OpenAI's gym [47] environment format, we implemented the **reset** and **step** methods as shown in Figures 4.11 (reset) and 4.12 (step).

- **reset**: Before creating an episode for training, necessary settings are configured. Initially, N services are registered in the Kubernetes cluster, with replicas cre-

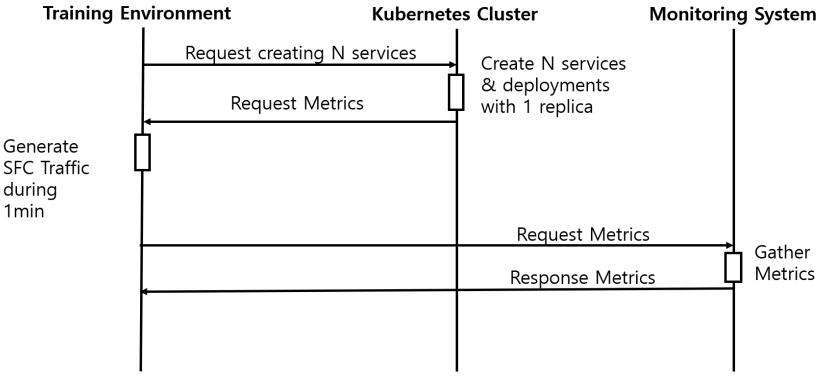


Figure 4.11: RL Environment Reset flow

ated for each. Background traffic is sent to apply basic system load, and metrics are collected, with the initial state observed and conveyed to the model. Since this system trains the model in a real Kubernetes environment, actual pod deployment and traffic generation are involved. In this training process, SFC paths must be configured, and with $N$ services, the number of possible SFC paths is theoretically $N!$. To limit the number of paths, the order of services was predefined, restricting paths to $2^N$ cases.

- **step**: The actual interaction and learning phase with the DRL model. Here, the scenario involves gradually increasing the number of replicas for a specific service randomly, then deploying the pods. The DRL model acquires state information from the training environment and outputs actions based on node selection. The environment executes these actions, collecting all metrics. Metric collection in a Kubernetes environment requires a waiting period to account for pod installation time and traffic processing, set at about one minute. Thus, each step lasts over a minute. This process is repeated until as many pods as $\#Nodes \times \#Services$ have been deployed, continuously rewarding the learning model based on power consumption, latency, and service availability.

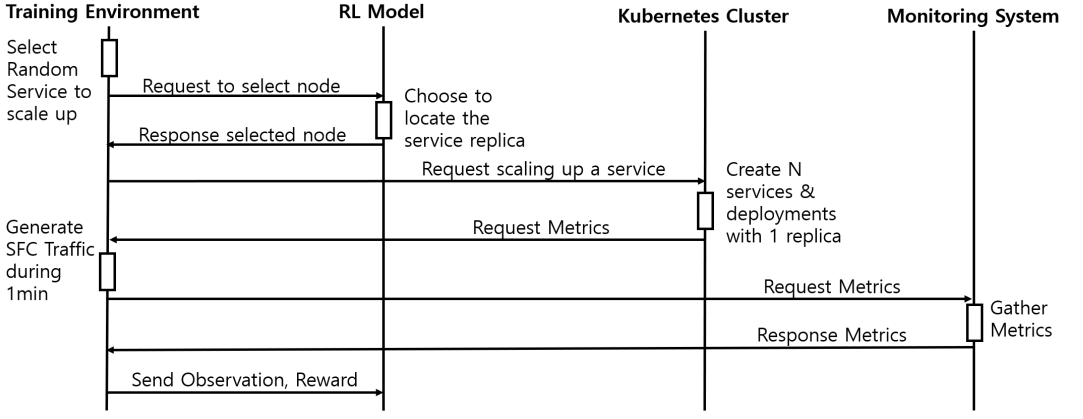


Figure 4.12: RL Environment Step flow

# V. Implementation

## 5.1 Enviornment

In this study, metrics were collected from an actual Kubernetes cluster environment, and these metrics were used to train the DRL model. Experiments were conducted on 4 hardware nodes based on the specifications shown in Table 5.1. One of the nodes was allocated as the master node, where VNFs could not be installed.

For the software corresponding to service functions, numerous publicly available functions exist, such as firewalls and Intrusion Detection Systems (IDS). However, collecting and modifying all of them posed challenges. Therefore, a program was developed to simulate VNFs. This program was designed to reflect CPU, memory, disk, and network load (see Subsection 5.1.1), and the experimental environment was set up to send various SFC requests based on this program. Table 5.2 details the specifications of the assumed services.

### 5.1.1 SFC Implementation

There are various types of service functions that can be used in a network. Common examples include firewalls, load balancers (LB), and Network Address Translation (NAT). Depending on the system requirements, other types of VNFs, such as Intrusion Detection Systems (IDS), TCP optimizers, and host ID injection, can also exist. These functions can be freely chained together, as shown in Figure 5.1, and they can

| **Name** | **CPU** | **Disk** | **Memory** |
|---|---|---|---|
| Dell PowerEdge R610 | 12 Core | 2 TB | 20 GiB |

Table 5.1: Hardware spec

| Name | CPU | Disk | Memory |
|---|---|---|---|
| Account | mid | mid | mid |
| Firewall | high | low | high |
| Host ID injection | high | high | low |
| IDS | high | high | high |
| NAT | high | low | low |
| Observer | low | low | low |
| Registry | low | high | high |
| Session | low | high | low |
| TCP optimizer | low | low | high |

Table 5.2: Software Load, detail load information on the Table 5.3 5.4 5.5

| | CPU_OPERATION_NUM | CPU_LIMIT(%) |
|---|---|---|
| High | 1000 | 30 |
| Middle | 250 | 30 |
| Low | 100 | 30 |

Table 5.3: Software CPU Load

| | MEM_OPERATION_NUM | MEM_BYTES(b) |
|---|---|---|
| High | 1000 | 100000 |
| Middle | 250 | 50000 |
| Low | 100 | 10000 |

Table 5.4: Software Memory Load

| | DISK_OPERATION_NUM | DISK_BYTES(b) |
|---|---|---|
| High | 1000 | 10000000 |
| Middle | 250 | 5000000 |
| Low | 100 | 2000000 |

Table 5.5: Software Disk Load

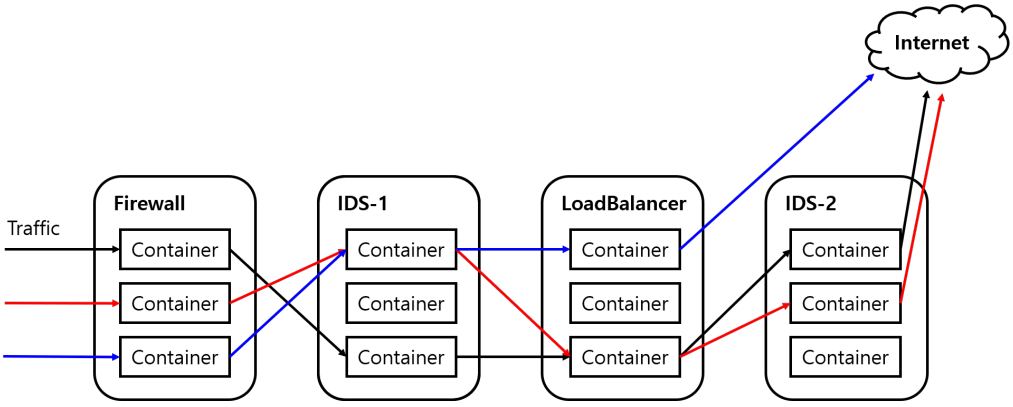


Figure 5.1: SFC Example

perform additional processing on traffic. However, implementing and experimenting with all of these functions is impractical. Hence, this study proposes a system to simulate these functions. Service functions use some computing and network resources to perform various processes on network traffic. Leveraging these characteristics, a system was devised where, upon receiving a request within an HTTP server, it performs pre-configured amounts of CPU, memory, and disk processing before forwarding the request to the next server. This system was inspired by the open-source project sfc-stress [48], which allows for setting CPU, memory, disk, and network workloads via HTTP requests. Each VNF performs its workload and then recursively forwards the traffic to the next target, according to the SFC configuration. Based on this approach, we propose a message-wrapping structure that specifies the next target in the SFC (see Figure 5.2).

Unlike sfc-stress, our implementation used stress-ng [49] to apply a wider variety of stress workloads, providing a more flexible design by allowing workload settings to be changed not only at initial execution but also during runtime. This implementation includes two APIs:

- **/load**: This API is called to apply load to the system, sending a message and a

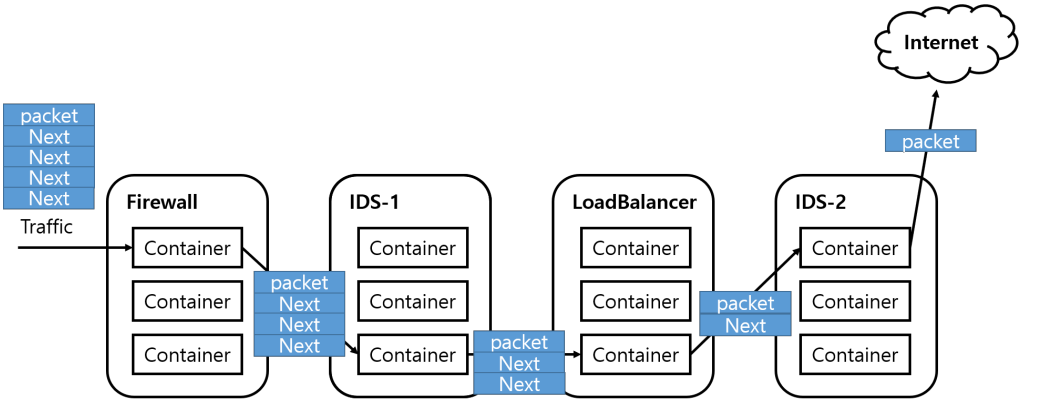


Figure 5.2: High Level Architecture of SFC

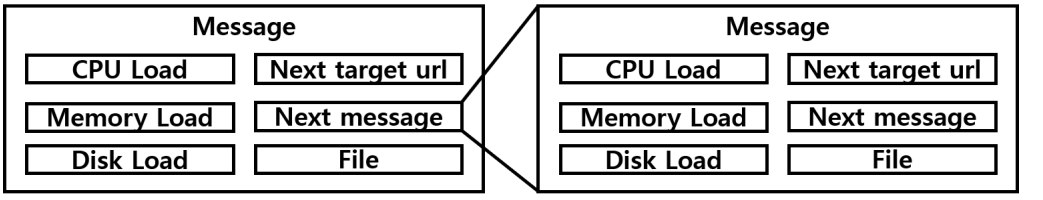


Figure 5.3: VNF Simulator Message

file. The file can be up to 10 MB in size, and the message specifies the amount of load to be applied to CPU, memory, and disk. The message also includes the target URL for the next destination and the message to be forwarded to the target. This is illustrated in Figure 5.3.

- **/config** : This API retrieves the current configuration settings. These settings can be initially configured via environment variables when the application is first launched. If no specific load is applied when the /load API is called, the workload is generated based on these default settings.

The detailed implementation of the actual system can be accessed on GitHub [50].

## 5.2 Baselines

To compare the proposed system's performance, we used two baseline models: the kube-scheduler, which is the default scheduler built into Kubernetes, and the scheduler presented in the NetMARKS [10] paper. We tried performance comparisons using these models.

- **Kube-scheduler**: This is the default package included in Kubernetes for pod scheduling. It considers CPU, memory usage, and availability to place pods. To ensure user autonomy, it supports actions such as favoring specific nodes or entirely blocking placements on certain nodes.
- **NetMARKS**: While the kube-scheduler accounts for CPU and memory usage in scheduling, it does not consider network variables. NetMARKS addresses this by implementing a plugin for the kube-scheduler that also takes network variables into account. The network metrics are based on information provided by Istio, making a service mesh structure essential.

## 5.3 Evaluation

For the experiment, we deployed a total of 30 pods across 4 nodes. Excluding the master node, 10 services could be evenly distributed across the remaining 3 nodes, intending to deploy one service per node. This process was repeated 10 times for each algorithm, and the average results were collected.

### 5.3.1 Performance compared with baselines

Compared to the previously mentioned kube-scheduler and NetMARKS, our methodology showed higher performance in terms of latency and power consumption. As shown in Figure 5.4, our approach reduced average latency and power consumption by 5% each compared to the kube-scheduler. This improvement was also evident

| | | Kube-Scheduler | NetMARKS | Ours |
|---|---|---|---|---|
| **Average Latency (ms)** | Min / Max | 2.56 / 7.49 | 2.57 / 7.96 | 2.46 / 9.05 |
| | Mean ± Std | 4.67 ±1.62 | 4.60 ± 1.76 | **4.44 ±1.76** |
| **Average Power Consumption (kJ)** | Min / Max | 2.29 / 2.88 | 2.23 / 2.92 | 2.22 / 2.80 |
| | Mean ± Std | 2.63 ±0.19 | 2.63 ± 0.19 | **2.50 ±0.17** |
| **Average Service Availability** | Min / Max | 0.40 / 1.0 | 0.40 / 0.97 | 0.35 / 0.68 |
| | Mean ± Std | **0.70 ± 0.18** | 0.69 ± 0.17 | 0.52 ± 0.10 |

Figure 5.4: Performance compared with baselines with ours (1)

when compared to NetMARKS. However, our model did not perform well in ensuring service availability. This issue arises from the tradeoff relationship between average latency and service availability. In our experiment, we deployed pods equal to the number of nodes multiplied by the number of services. To maximize service availability, it is best to deploy one service per node. Deviations from this optimal deployment inevitably result in reduced service availability. Figure 5.5 illustrates how kube-scheduler and NetMARKS emphasize service availability. Consequently, our algorithm achieved benefits in latency and power consumption without significantly compromising service availability. Although previous DRL-based algorithms showed higher performance in terms of power consumption or latency, they did not address the reduction in service availability. Our experimental results are significant in that we reduce latency and power consumption simultaneously while meaningfully maintaining service availability.

### 5.3.2 GNN algorithm comparison

We compared the performance of a feature extractor built using a simple Multi Layer Perceptron (MLP) with stacked Linear Layers against a GNN-based approach.

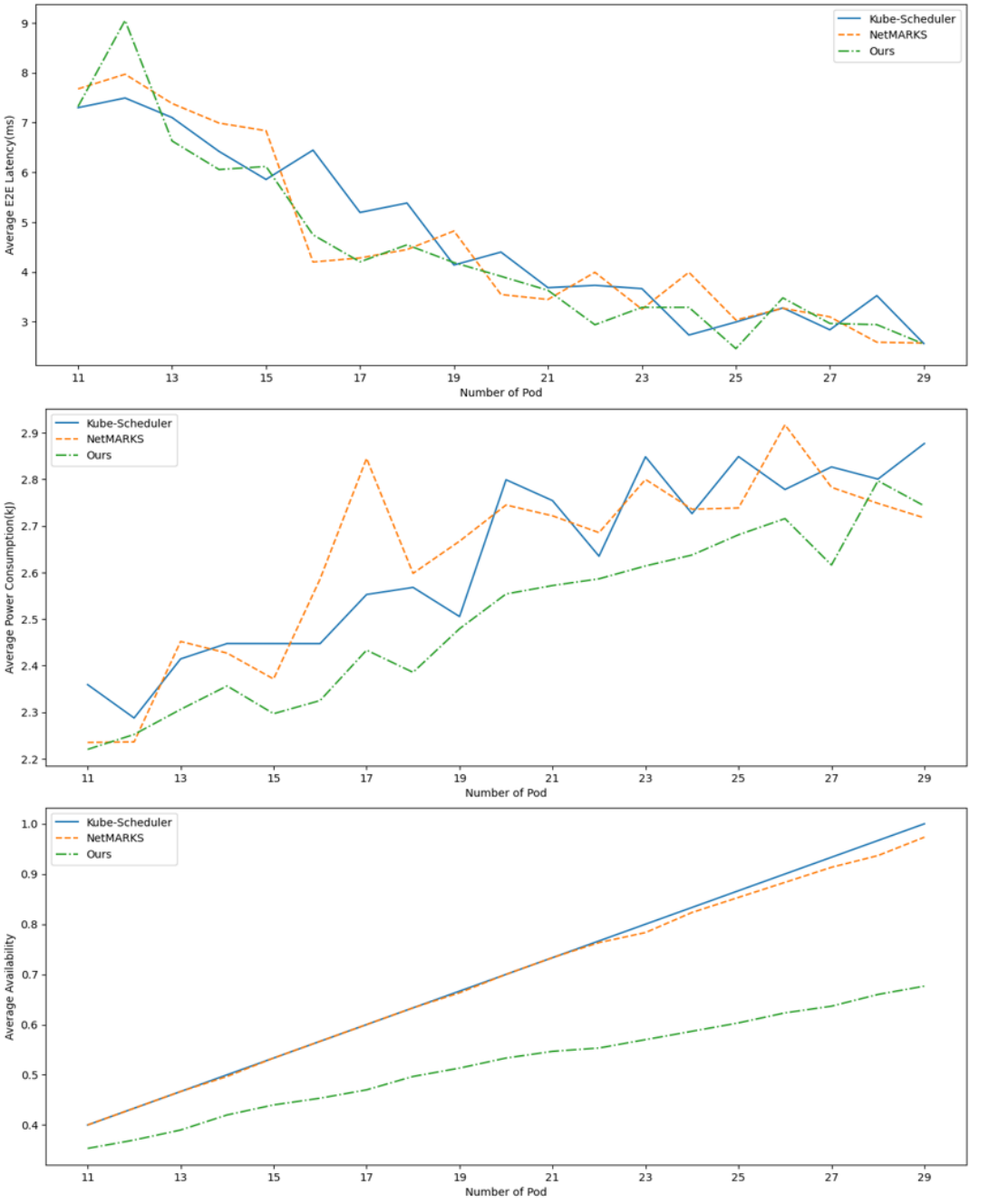


Figure 5.5: Performance compared with baselines with ours (2)

| | | MLP | GAT | GAT / Graph SAGE |
|---|---|---|---|---|
| **Average Latency (ms)** | Min / Max | 2.50 / 8.55 | 2.46 / 9.05 | 2.36 / 8.60 |
| | Mean ± Std | 4.56 ±1.80 | **4.44 ±1.76** | 4.75 ±1.93 |
| **Average Power Consumption (kJ)** | Min / Max | 2.26 / 2.90 | 2.22 / 2.80 | 2.24 / 2.92 |
| | Mean ± Std | 2.58 ±0.19 | **2.50 ±0.17** | 2.63 ±0.21 |
| **Average Service Availability** | Min / Max | 0.38 / 0.66 | 0.35 / 0.68 | 0.35 / 0.67 |
| | Mean ± Std | **0.53 ± 0.08** | 0.52 ± 0.10 | 0.53 ± 0.10 |

Figure 5.6: GNN algorithm comparison (1)

For GNN, first, we utilized GAT for the entire graph embedding process and, we trued Graph SAGE to edge when there were no attributes (pod-to-service edge, pod-to-node edge) and GAT for the rest (service-to-service edge, node-to-node edge). As shown in Figure 5.6, the approach using only GAT showed the best performance. While the MLP demonstrated better performance in service availability, all three algorithms showed similar figures in this aspect. Therefore, GAT is the most appropriate for embedding the features of this system. Additionally, the combination of GAT and Graph SAGE showed lower performance than MLP, likely due to significant information loss during the graph embedding process. Figure 5.7 indicates that the method using GAT consistently showed high performance throughout the entire deployment process.

### 5.3.3 DRL algorithm comparison

To identify the performance differences based on the chosen DRL algorithm, we compared the performance of off-policy algorithms using the same GNN feature extractor. As shown in Figure 5.8, TD3 exhibited the highest overall performance. DQN showed generally low performance due to its poor action inference capabilities. SAC performed well in scenarios where exploration is crucial, but since our experiment lim-

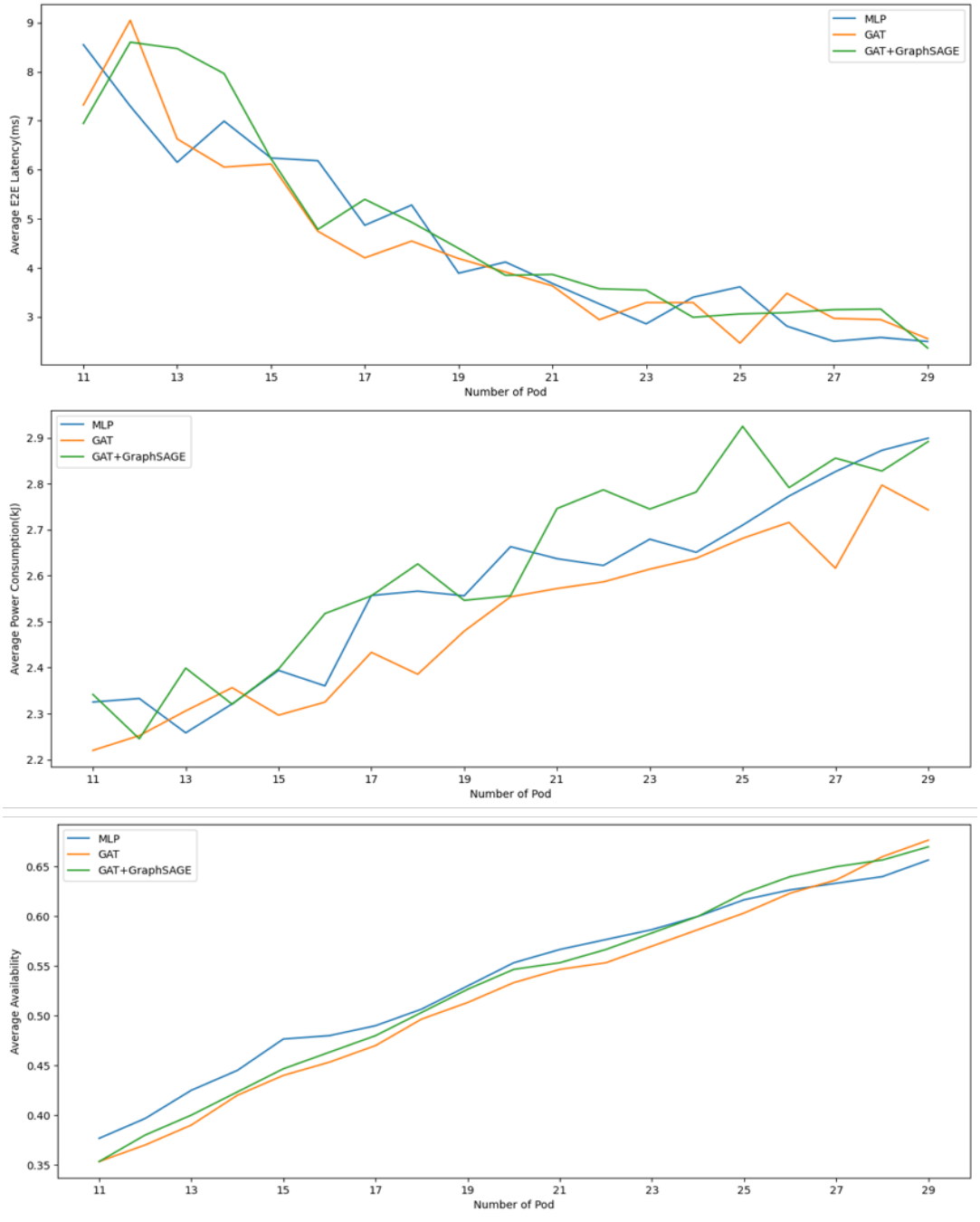


Figure 5.7: GNN algorithm comparison (2)

| | | DQN | TD3 | SAC |
|---|---|---|---|---|
| **Average Latency (ms)** | Min / Max | 4.86 / 13.39 | 3.63 / 12.05 | 4.65 / 11.86 |
| | Mean ± Std | 7.80 ± 2.21 | **7.29 ± 2.66** | 7.71 ± 2.19 |
| **Average Power Consumption (kJ)** | Min / Max | 2.58 / 3.09 | 2.37 / 3.00 | 2.45 / 3.25 |
| | Mean ± Std | 2.86 ± 0.15 | **2.76 ± 0.18** | 2.82 ± 0.19 |
| **Average Service Availability** | Min / Max | 0.35 / 0.66 | 0.36 / 0.71 | 0.35 / 0.66 |
| | Mean ± Std | 0.52 ± 0.09 | 0.52 ± 0.11 | **0.54 ± 0.09** |

Figure 5.8: DRL algorithm comparison (1)

ited the number of nodes to four, SAC could not leverage its exploratory advantage, resulting in lower performance than TD3. Figure 5.9 shows the performance metrics as the number of pods increases, indicating that while SAC performs well in the early stages, it falls behind TD3 in the later stages.

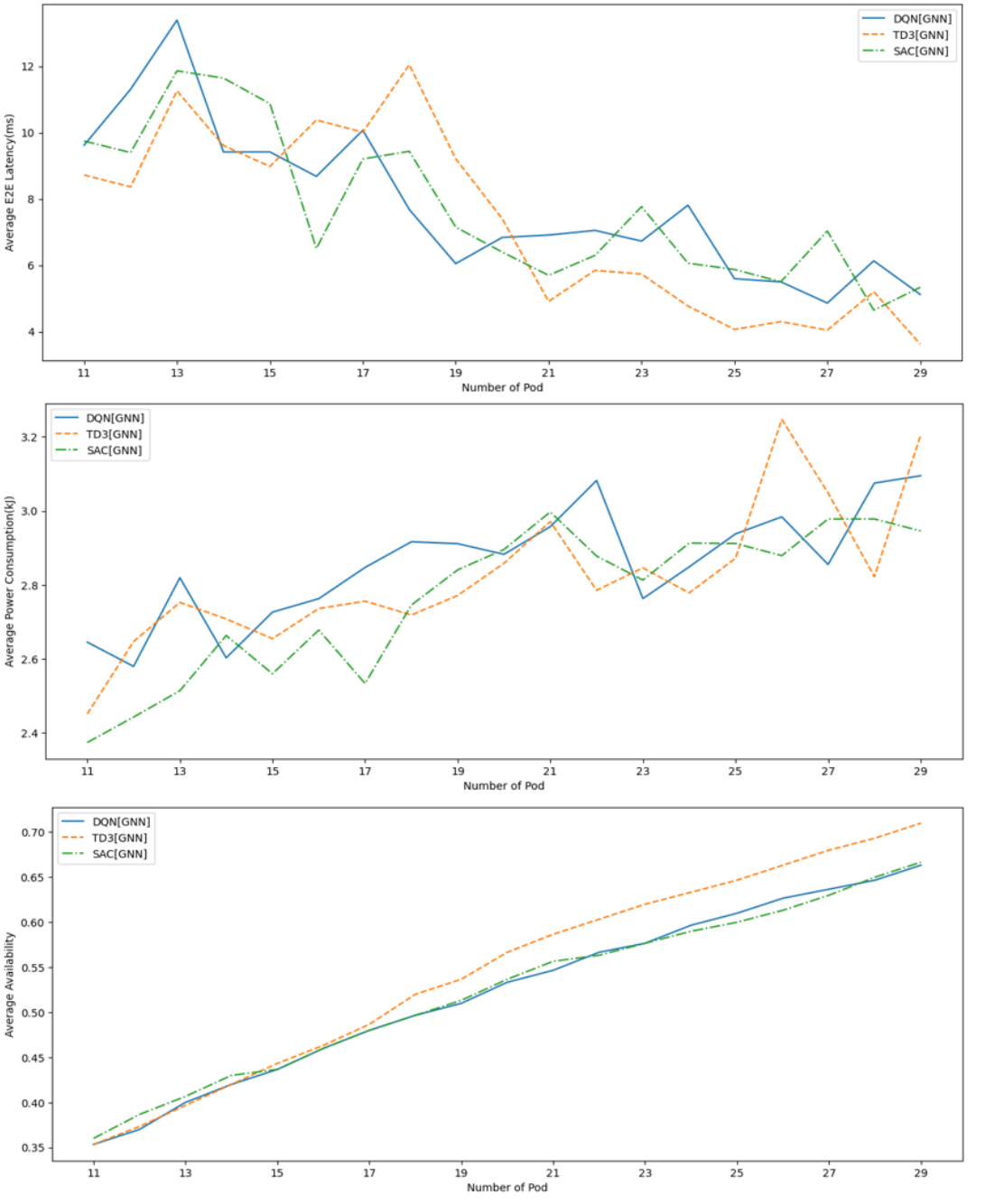


Figure 5.9: DRL algorithm comparison (2)

# VI. Conclusion

## 6.1 Summary

This study explored an algorithm for the optimal placement of SFCs deployed on Kubernetes clusters. The initial design of Kubernetes clusters aimed to maximize availability, which resulted in suboptimal performance in terms of QoS and power consumption. Previous research has focused on improving QoS, including power consumption and latency, but comprehensive research addressing all these aspects simultaneously has been lacking. Therefore, this study considered power consumption, QoS (especially latency), and service availability as optimization factors. To effectively solve this multi-objective problem, we applied a DRL algorithm.

For the training and experimentation of the DRL algorithm, we modeled the actual cluster environment as a Markov chain and designed the state, action, and reward accordingly. Previous studies tended to focus on the information of nodes, services, or containers alone during state modeling or attempted to use GNN to leverage relationships among single entities. However, this study proposed a heterogeneous graph structure that encompasses the relationships among all these entities. Consequently, the study trained the GNN using the relationships among nodes, services, and associated containers, effectively embedding the information of nodes and services.

Additionally, the training and experimentation were conducted in an actual cluster environment, and a new VNF simulation tool was defined to build a more advanced experimental setup. A monitoring environment integrating six modules was established to collect information on nodes, containers, services, node-to-node, node-to-container, and service-to-service interactions. This setup enabled performance comparisons with existing baseline systems, resulting in an average performance improvement in latency and power consumption of 5%, while taking service availability into account.

Furthermore, the entire system's construction, implementation, and experimentation processes were made publicly available on GitHub [34]. This openness aims to provide a foundation for future research to achieve further performance improvements.

## 6.2 Limitation

This study proposed a novel algorithm for SFC scheduling in a Kubernetes environment, utilizing GNN and DRL to demonstrate effective SFC scheduling. However, several limitations were identified in this research.

First, the study has the limitation that learning needs to be repeated continuously depending on the number of nodes. In other words, if new nodes are added to or removed from the cluster, the entire learning process must be redone. This presents a significant constraint in system design, necessitating a new approach to address this issue.

Second, there is a lack of consideration for multi-cluster environments. In real-world scenarios, deploying SFCs often involves multiple clusters working together. However, this research was limited to a single cluster, highlighting a need for further studies that include multi-cluster scenarios.

Lastly, the study used the default load balancer provided by Kubernetes for distributing traffic within the service. This load balancer employs a round-robin method, which is not the most effective for traffic distribution. Therefore, there is a need for more efficient load balancing or SFC pathfinding techniques. Designing SFC paths at the pod level could potentially lead to better performance.

## 6.3 Future Work

As outlined in the limitations, there are several areas for further research beyond this study. To address each limitation, we plan to consider the following research directions:

For the first issue related to the change in the number of nodes, we could perform imitation learning or pre-train the GNN layers to create a more general design. This approach could help in building an architecture that operates stably even with changes in the number of nodes during the system's learning process or one that can converge with minimal training.

Regarding the second issue of multi-cluster environments, the state size would need to cover a larger observation space. To address this, we could expand the node and cluster relationships into another graph and effectively embed this using GNN. This method could potentially achieve high performance.

To resolve the final issue, we could integrate research on SFC path design and implement it using the DRL algorithm, which could lead to a higher performance system. Thus, future research will explore these methods to overcome the identified limitations.

# 요 약 문

기술의 발전은 기반 시설에 해당하는 네트워크에 대한 요구사항 증가를 불러왔다. 이를 유연하고, 효과적으로 운용하기 위해서 Network Function Virtualization (NFV)와 Service Function Chain (SFC)이라는 개념이 등장하였지만, 이들을 효율적으로 운용하지 않으면 service의 가용성과 Quality of Service (QoS), 그리고 전력 소모 측면에서 많은 비용을 지불할 수 밖에 없다. 기존 연구에서는 이를 극복하기 위해서 AI를 활용하는 Deep Reinforcement Learning (DRL)과 Graph Neural Network (GNN)을 활용한 노력을 해왔다. 하지만, 기존 연구들에서는 cluster를 이루는 node, 그리고 service를 이루는 container 간의 관계를 명확하게 정의하지 않았고, 이를 효과적으로 embedding하는 방법을 제시할 수 없었고, 이로 인해서 효율적인 scheduling을 수행하는데 한계가 존재하였다. 따라서, 우리는 이 관계를 graph 형태로 정의하고, 이를 효과적으로 embedding할 수 있는 시스템을 제안하였다. 뿐만 아니라 기존 연구에서는 검증 단계에서 simulation에서 그치는 한계가 존재하였고, 실제로, network traffic을 수집하는 등의 과정은 수행되지 못했다. 이러한 연구에서는 직접적인 성능을 보장하는데 한계가 있었다. 따라서, 해당 연구에서는 실제 환경에서 이를 적용하기 위한 시스템을 구축하였고, 실제 성능 검증을 통해서 ㅣservice availability를 유의미하게 유지하면서도 latency와 power consumption을 5% 감축시킬 수 있었다.

# References


[1] kubernetes. Scheduling framework, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/. [Accessed May 27, 2024].

[2] Carlos J. Bernardos, Akbar Rahman, Juan-Carlos Zúñiga, Luis M. Contreras, Pedro Andres Aranda, and Pierre Lynch. Network Virtualization Research Challenges. RFC 8568, April 2019.

[3] Joel M. Halpern and Carlos Pignataro. Service Function Chaining (SFC) Architecture. RFC 7665, October 2015.

[4] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[5] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[7] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods, 2018.

[8] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th*

*International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870. PMLR, 10–15 Jul 2018.

[9] Abdikarim Mohamed Ibrahim, Kok-Lim Alvin Yau, Yung-Wey Chong, and Celimuge Wu. Applications of multi-agent deep reinforcement learning: Models and algorithms. *Applied Sciences*, 11(22), 2021.

[10] Łukasz Wojciechowski, Krzysztof Opasiak, Jakub Latusek, Maciej Wereski, Victor Morales, Taewan Kim, and Moonki Hong. Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–9, 2021.

[11] Al Morton. Considerations for Benchmarking Virtual Network Functions and Their Infrastructure. RFC 8172, July 2017.

[12] Paul Quinn and Thomas Nadeau. Problem Statement for Service Function Chaining. RFC 7498, April 2015.

[13] Rackspace Hosting and NASA. Openstack, 2010. [Online]. Available: https://opendev.org/openstack. [Accessed May 27, 2024].

[14] Google. Kubernetes, 2014. [Online]. Available: https://github.com/kubernetes/kubernetes. [Accessed May 27, 2024].

[15] CNCF. Case studies, 2023. [Online]. Available: https://www.cncf.io/case-studies. [Accessed May 27, 2024].

[16] OpenStack. Network functions virtualization (nfv), 2024. [Online]. Available: https://docs.openstack.org/charm-guide/latest/admin/compute/nfv.html. [Accessed May 27, 2024].

[17] ONAP. Onap, 2022. [Online]. Available: https://www.onap.org/. [Accessed May 27, 2024].

[18] Valentina Cacchiani, Manuel Iori, Alberto Locatelli, and Silvano Martello. Knapsack problems — an overview of recent advances. part ii: Multiple, multidimensional, and quadratic knapsack problems. *Computers  Operations Research*, 143:105693, 2022.

[19] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[20] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[21] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[22] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks?, 2022.

[23] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[24] Yu Liu, Xiaojun Shang, and Yuanyuan Yang. Joint sfc deployment and resource management in heterogeneous edge for latency minimization. *IEEE Transactions on Parallel and Distributed Systems*, 32(8):2131–2143, 2021.

[25] Angelo Marchese and Orazio Tomarchio. Orchestrating dag workflows on the cloud-to-edge continuum. 2023.

[26] Zeyuan Wang, Xinglin Zhang, and Lei Yang. Eis: Edge information-aware scheduler for containerized iot applications. In *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*, pages 280–289, 2023.

[27] Anna Reale Michael Chima Ogbuachi and Benedek Kovács. Context-aware kubernetes scheduler for edge-native applications on 5g. *Journal of communications software and systems*, 16(1):85–94, 2020.

[28] John Rothman and Javad Chamanara. An rl-based model for optimized kubernetes scheduling. In *2023 IEEE 31st International Conference on Network Protocols (ICNP)*, pages 1–6, 2023.

[29] Yanming Liu, Chuangchuang Zhang, Hongyong Yang, Shuning Zhang, Xingwei Wang, and Fuliang Li. Deep reinforcement learning based reliability aware sfc placement in multi-domain networks. In *2023 15th International Conference on Communication Software and Networks (ICCSN)*, pages 215–219, 2023.

[30] Lei Yang, Junzhong Jia, Hongcai Lin, and Jiannong Cao. Reliable dynamic service chain scheduling in 5g networks. *IEEE Transactions on Mobile Computing*, 22(8):4898–4911, 2023.

[31] Siyu QI, Shuopeng LI, Shaofu LIN, Mohand Yazid SAIDI, and Ken CHEN. Energy-efficient vnf deployment for graph-structured sfc based on graph neural network and constrained deep reinforcement learning. In *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 348–353, 2021.

[32] Meilin Xu, Min Jia, and Qing Guo. Delay-sensitive sfc scheduling optimization with drl in satellite-terrestrial networks. In *2023 IEEE 23rd International Conference on Communication Technology (ICCT)*, pages 1680–1684, 2023.

[33] Chengfeng Jian, Zhuoyang Pan, Lukun Bao, and Meiyu Zhang. Online-learning task scheduling with gnn-rl scheduler in collaborative edge computing. *Cluster Computing*, 27(1):589–605, 2024.

[34] Eui-Dong Jeong. k8s sfc deployment, 2024. [Online]. Available: https://github.com/k8s-SFC-deployment. [Accessed May 27, 2024].

[35] The Linux Foundation. Prometheus - monitoring system time series database, 2024. [Online]. Available: https://prometheus.io/. [Accessed May 27, 2024].

[36] Prometheus. Node exporter, 2013. [Online]. Available: https://github.com/prometheus/node_exporter. [Accessed May 27, 2024].

[37] Google. cadvisor, 2016. [Online]. Available: https://github.com/google/cadvisor. [Accessed May 27, 2024].

[38] Mike Muuss. Ping, 1983. [Online]. Available: https://ftp.arl.army.mil/ mike/ping.html. [Accessed May 27, 2024].

[39] Rusty Russell. iptabels, 1998. [Online]. Available: https://git.netfilter.org/iptables/. [Accessed May 27, 2024].

[40] Eui-Dong Jeong. Nmbn exporter, 2024. [Online]. Available: https://github.com/k8s-SFC-deployment/nmbn-exporter. [Accessed May 27, 2024].

[41] Istio Authors. Istio, 2017. [Online]. Available: https://istio.io/. [Accessed May 27, 2024].

[42] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225, 2019.

[43] Project Flotta. Powertop monitoring, 2022. [Online]. Available: https://github.com/project-flotta/powertop_monitoring. [Accessed May 27, 2024].

[44] Intel. Powertop, 2007. [Online]. Available: https://github.com/fenrus75/powertop. [Accessed May 27, 2024].

[45] Eui-Dong Jeong. sfc-e2e-collector, 2024. [Online]. Available: https://github.com/k8s-SFC-deployment/sfc-e2e-collector. [Accessed May 27, 2024].

[46] Eui-Dong Jeong. Deep reinforcement learning, 2024. [Online]. Available: https://github.com/k8s-SFC-deployment/rl. [Accessed May 27, 2024].

[47] OpenAI. gym, 2019. [Online]. Available: https://github.com/openai/gym. [Accessed May 27, 2024].

[48] Michel Gokan Khan. sfc-stress, Aug 6, 2018. [Online]. Available: https://github.com/michelgokan/sfc-stress. [Accessed May 27, 2024].

[49] ColinIanKing. stress-ng, May 28, 2024. [Online]. Available: https://github.com/ColinIanKing/stress-ng. [Accessed May 28, 2024].

[50] Eui-Dong Jeong. vnf-scc-sfc, 2024. [Online]. Available: https://github.com/k8s-SFC-deployment/VNF-SCC-SFC. [Accessed May 27, 2024].

# Acknowledgements

여러 가지 연구를 진행할 수 있도록 도와주신 홍원기 교수님 그리고 유재형 교수님께 감사 인사를 드립니다. 제가 석사 과정 동안 연구했던 AI 기반 Network Management(VM Consolidation, Switch Auto-Configuration, Container Scheduling)의 토대를 제시해주셨습니다. 연구를 진행하는 과정에서도 다양한 문제를 겪었는데, 그때마다 많은 도움을 주신 교수님들과 연구실 선배들에게 감사드립니다. 미팅을 진행하면서, 또는 그 이후에 희곤이형, 석현이형, tu, 상우와 이야기를 나누면서 새로운 해결책을 발견할 수 있었습니다. 동일한 연구 주제가 아닐지라도 제 이야기에 귀 기울이고, 다양한 의견을 주신 모든 분들께 감사드립니다.

연구 이외에도 다양한 경험을 할 수 있도록 도와주신 랩실 구성원 그리고 저와 추억을 나누었던 모두들에게 감사드립니다. 여러분들이 있었기에 포항에서 다양한 경험을 할 수 있었습니다. 같이 밥을 먹는 것부터, 여유롭게 마셨던 커피 한 잔, 철길숲을 자전거 타고 같이 달리기도 하고, 주마다 수영도 가고, 영화, 스노보드, 여행, 등등 많은 추억을 만들 수 있었습니다. 함께 해주신 모든 분들 덕분에 이 순간들이 모두 즐거운 추억이 되었습니다. 아마도 포항을 생각하면 다른 어떤 것보다 여러분들이 먼저 떠오를 거 같습니다. 앞으로도 하시는 일의 과정이 즐거울 뿐만 아니라, 많은 이들에게 영감을 줄 수 있기를 진심으로 응원하겠습니다.

처음 포항에 내려온 순간으로 부터 3년 정도의 시간이 흐르니 너무 많은 것을 배워서 돌아갈 수 있었습니다. 특히, 학교에서 들었던 수업, 학회 참석 및 발표를 하면서, 무엇을 하더라도 열정을 가지는 것이 얼마나 멋있는지 그리고, 얼마나 중요한지 배울 수 있었습니다. 이들을 통해서 동기부여를 받을 수 있었고, 하고자 하는 연구에 도전하여 최선을 다 해볼 수 있었던 거 같습니다. 이 과정이 즐거웠고, 앞으로도 삶에 있어서 좋은 동력이 될 거 같습니다.

마지막으로, 항상 믿어주시고, 응원해주시는 부모님과 형을 포함한 우리 가족 그리고, 올라갈 때마다 항상 반갑게 맞이해주시는 이모, 이모부, 한솔이 모두 감사합니다.

# Curriculum Vitae


## Eui-Dong Jeong

Master's degree in progress
Department of Computer Science and Engineering
POSTECH
77, Cheongam-ro, Nam-gu, Pohang-si, Gyeongsangbuk-do, Republic of Korea

: justicedong@ ✉ .tech.ac.kr
: +82) 010-957 📞 4127
: https://github.com/euidong
Blog: https://euidong.github.io


## EDUCATION

Sep. 2022 ~ Aug. 2024 — **Pohang University of Science and Technology**, Seoul, Korea
Department of Computer Science and Engineering

*Advisor: Prof. James Won-ki Hong*
*Co-Advisor: Prof. Jae-Hyoung Yoo*
Master Student
*GPA:* **4.15 / 4.3**

Mar. 2016 ~ Aug. 2022 — **Kyung Hee University**, Seoul, Korea
Department of Computer Science and Engineering

Computer Engineering
*GPA:* **3.874 / 4.3**

## RESEARCH INTEREST

- **AI-based Network Management**

✓ With the advent of 5G and 5G-Advanced, Network Function Virtualization (NFV) has been employed to enable flexible management of networks. Nevertheless, the multitude of network functions (NFs) present within the network, intended to satisfy a range of requirements (such as interference minimization, reduced latency, and energy-saving), has necessitated the need for advanced network management techniques. To solve this problem, we need an efficient scheduling method. However, as the network grows in complexity, with increasing number of servers and NFs, managing these elements efficient becomes more challenging (NP-Hard). Therefore, I studied an efficient scheduling method using AI to infer and approximate the solution to the optimization problem.

✓ Related Projects

■ 2022-2024, 자연어 처리 및 기계 학습 기반 네트워크 비정상 상태 탐지 기술 개발, Contributed to building environment (test bed) and presented ideas.

- 2022-2024, 인공지능 기반 가상 네트워크 관리 기술 개발 연구, Contributed to NFV Scheduling Algorithm development with Reinforcement Learning (RL).
- 2023, S-Witch: Switch Configuration Assistant with LLM, Developed Switch Configuration Assistant

✓ Related Course Work

- **Networking :** Advanced Network Management, Internet Traffic Monitoring, IoT Network, Cloud Computing, Datacenter Programming
- **AI :** Machine Learning, Deep Learning, Computer Vision, Statistical NLP

Also, interested in **Cloud Computing, Distributed Computing, Network Virtualization, and Deep Learning**

## PUBLICATIONS (INTERNATIONAL CONFERENCE)

1. Wonseok Choi, **Eui-Dong Jeong**, Jongsoo Woo, James Won-Ki Hong, "Optimizing Block Propagation in Bitcoin Network with Region-based Neighbor Selection Using Reinforcement Learning", ICBC2024, (2024) – **Oral by 1st Author**
2. **Eui-Dong Jeong**, Jae-Hyoung Yoo, James Won-Ki Hong, "S-Witch: Switch Configuration Assistant with LLM and Prompt Engineering", NOMS2024, (2024) – ***Oral by 1st Author***
3. **Eui-Dong Jeong**, Jae-Hyoung Yoo, James Won-Ki Hong, "SFC Consolidation: Energy-aware SFC Management using Deep Reinforcement Learning", NOMS2024, (2024) ***- Poster***
4. **Eui-Dong Jeong**, Jae-Hyoung Yoo, James Won-Ki Hong, "SDN Lullaby: VM Consolidation for SDN using Transformer-Based Deep Reinforcement Learning", CNSM2023, (2023) ***- Poster***
5. Sukhyun Nam, **Euidong Jeong**, Jibum Hong, Jae-Hyoung Yoo, James Won-Ki Hong, "Log Analysis and Prediction for Anomaly Detection in Network Switches", CNSM2023, (2023) ***- Oral by 2nd Author***

## PUBLICATIONS (DOMESTIC CONFERENCE)

1. **정의동**, 유재형, 홍원기, "화상 회의 시스템 분석을 위한 LSTM-CGAN 기반 네트워크 트래픽 생성 모델 설계", KNOM Conference 2023, (2023) ***- Poster***
2. 남석현, **정의동**, 홍지범, 유재형, 홍원기, "패턴 분석 기반 스위치 로그 예측 기법 연구", KNOM Conference 2023, (2023) ***- Poster***

## PROJECTS (RESEARCH LAB)

- **자연어 처리 기반 네트워크 비정상 상태 탐지 기술 개발, Samsung(SNIC), Korea**

Jun. 2022 ~ Present

✓ **Abstract:** In order to detect abnormal states of switches and routers in the network, the device's log data is analyzed through NLP, and the task of identifying abnormal states is performed. In this process, abnormal states are detected through the use of

natural language processing models such as BERT and classic natural language processing methods such as TF-IDF.

- ✓ **Contribution:** I led the construction of a **testbed** for data collection and the actual log collection process and presented ideas such as model structure (ensemble classical method and AI method) during the entire system construction process.

- **인공지능 기반 가상 네트워크 관리기술 개발, 과학기술정보통신부, Korea**

Jun. 2022 ~ Feb. 2024

- ✓ **Abstract:** In order to configure a flexible and efficient network, an orchestrator that can control Virtualized Network Functions (VNFs) distributed at the edge and data center is implemented. This implementation includes detailed modules such as VNF deployment, load balancing, autoscaling, intrusion detection, and power saving.
- ✓ **Contribution:** I was in charge of developing the **power-saving module**. To achieve this, a consolidation process based on reinforcement learning was executed, enabling the operation of VNFs (as Virtual Machines) with a minimal number of servers. Through this process, we successfully reduced power consumption by employing algorithms such as DQN, PPO, and Rainbow.
- ✓ **Github:** https://github.com/euidong/sdn-lullaby and https://github.com/dpnm-ni/ni-power-management

## PROJECTS (COURSE WORK)

- **Somewhere Over the Margin, POSTECH Computer Vision Course, Korea**

Aug. 2023 ~ Dec.2023

- ✓ **Abstract:** Due to the limitations of **Triplet Margin Loss** (can't use semi-negative samples) used in Metric Learning, research was conducted to replace the hinge function with other smoothed functions to activate semi-negative samples.
- ✓ **Contribution:** I proposed the overall idea, implemented the requirements by utilizing and modifying the pytorch metric learning library, and established an experiment environment.
- ✓ **Github:** https://github.com/JeongHeonDong/somewhere-over-the-margin

- **Side Teacher Loss, POSTECH Deep Learning Course, Korea**

Mar. 2023 ~ Jun. 2023

- ✓ **Abstract:** Conducted research to improve student performance using Teacher-Student Architecture in Transfer Learning. In this process, [1]**Tasanjiseok** proposed a method of reflecting one of the human learning methods in AI learning, and we researched to improve the model's performance in the following learning by using overfitted models.
- ✓ **Contribution** I presented the idea of **Tasanjiseok** and developed a plan to utilize it through discussion with team members. Additionally, I actively lead the implementation and experimentation of ideas using pytorch.
- ✓ **Github:** https://github.com/euidong/SideTeacherLoss

[1] **Tasanjiseok (타산지석)**: By other's faults wise men correct their own.

- **Video Traffic Generator, POSTECH Internet Traffic Monitoring Course, Korea**

Mar. 2023 ~ Jun. 2024

✓ **Abstract:** Collecting traffic data and applying it is a huge burden both in terms of memory and time. In particular, this problem is more pronounced in the case of video. Therefore, in this project, time series data generation using GAN is performed to generate video traffic data as needed.

✓ **Contribution:** I collected video traffic data, performed labeling, and built a Deep Learning Model Architecture using LSTM to generate time series data through GAN.

✓ **Github:** https://github.com/euidong/video-traffic-generation

## PROJECTS (PERSONAL)

- **S-Witch : Switch Configuration Assistant with LLM**

Dec. 2023 ~ Feb.2024

✓ **Abstract**: In order to perform switch/router configuration, the process of entering CLI commands is required. However, not only are the commands different based on the unique OS of each device such as Juniper and Cisco, but also different settings need to be made depending on the version even if it is the same vendor. When this process is performed by people, the possibility of human error increases significantly. Therefore, in order to solve this problem, the Switch Configuration Assistant was implemented using LLM.

✓ **Contribution:** I used LangChain Library and GPT-3, and in order to make the delivery of topology information clear, I built a topology using GNS3 and created a CLI command based on this using topology and other network information.

✓ **Github:** https://github.com/euidong/S-Witch

## AWARDS AND HONORS

| | |
|---|---|
| 2021 | KHUthon2021 (Excellence Award), Naver D2 and Kyunghee University, Korea |
| 2020 | KHUthon2020 (Grand Award), Naver D2 and Kyunghee University, Korea |
| | Startup Idea Hackathon (Bronze Award), Four major port corporations, Korea |
| 2019 | KHUthon2019 (Excellence Award), Naver D2 and Kyunghee University, Korea |
| | 2019 SW Festival (Junior Award), Kyunghee University, Korea |

## SKILLS AND TECHNIQUES

- **Programming Skills**

✓ Highly Skilled : Python, C++, Typescript and Javascript (FrontEnd / BackEnd)

✓ Experienced : Java, Golang, C, and C#

- **Virtualization and Cloud Techniques**

✓ Experienced : Docker, Vagrant, Kubernetes, ONOS, OpenFlow