저작자표시-비영리-변경금지 2.0 대한민국

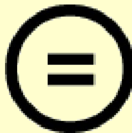
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Doctoral Dissertation

Auto-scaling of Network Services in Multi-access Edge Computing Environment

Doyoung Lee (이 도 영)

Department of Computer Science and Engineering

Pohang University of Science and Technology

2021

dCollection @ postech

MEC 환경에서의 네트워크 서비스 자원 할당 자동화 방법

Auto-scaling of Network Services in Multi-access Edge Computing Environment

Auto-scaling of Network Services in Multi-access Edge Computing Environment

by

Doyoung Lee
Department of Computer Science and Engineering
Pohang University of Science and Technology

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

Pohang, Korea
06. 22. 2021
Approved by
James Won-Ki Hong (Signature)
Academic advisor

Auto-scaling of Network Services in Multi-access Edge Computing Environment

Doyoung Lee

The undersigned have examined this dissertation and hereby certify that it is worthy of acceptance for a doctoral degree from POSTECH

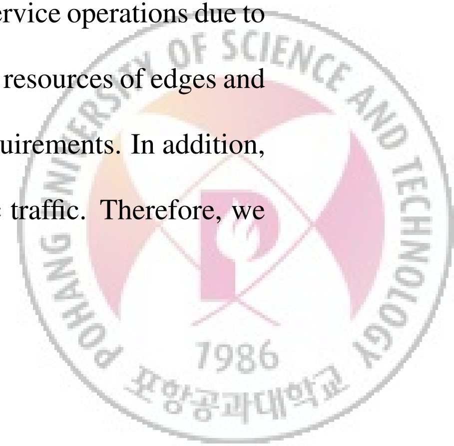
06. 22. 2021

Committee Chair	James Won-Ki Hong	(Seal)
Member	Young-Joo Suh	(Seal)
Member	Jae-Hyoung Yoo	(Seal)
Member	Inseok Hwang	(Seal)
Member	Hongtaek Ju	(Seal)

DCSE 20152333
이 도 영 Doyoung Lee
Auto-scaling of Network Services in Multi-access Edge Computing Environment.
MEC 환경에서의 네트워크 서비스 자원 할당 자동화 방법.
Department of Computer Science and Engineering , 2021, 75p
Advisor : James Won-Ki Hong.
Text in English.

ABSTRACT

In the 5G era, it is necessary to provide services to many devices connected to the networks while meeting diverse demands, such as high availability, low latency, and high data rates. Multi-access edge computing (MEC), one of the key technologies to achieve the goal, brings down the cloud-computing capabilities to network edges near users. The MEC environment provides resources allocated for the service operations and mitigates the core network's load by processing traffic at the edge of networks. Service running in the edges usually consists of multiple components interacting with each other because this architecture enables flexible and rapid service provisioning. However, those MEC and service architecture complicate the service operations due to many factors. Specifically, it is essential to consider the limited resources of edges and the location in which services run while satisfying the QoS requirements. In addition, it is hard to operate services manually in response to dynamic traffic. Therefore, we



propose an auto-scaling method using deep Q-network (DQN) in this thesis. The proposed method runs the adequate number of instances, that is, resources, composing service while considering the service performance and operating costs. The proposed approach consists of two models: the DQN model making scaling decisions, and the decision model choosing components (or locations) to be scaled. We implemented an auto-scaler as a module to allocate instances (virtual machines or containers) to service in the MEC environments. Our proposed method was validated by conducting several experiments with different scenarios.

Contents

I. Introduction **1**

1.1 Motivation and Problem Statement . . . 1

1.1.1 5G network era . . . 1

1.1.2 The needs for auto-scaling of services . . . 2

1.2 Research Goals and Approach . . . 3

1.3 Organization . . . 4

II. Background and Related Work **5**

2.1 Background . . . 5

2.1.1 Multi-access Edge Computing . . . 5

2.1.2 Microservice architecture . . . 6

2.1.3 Deep Reinforcement Learning . . . 7

2.2 Related Work . . . 9

2.2.1 Service provisioning in MEC . . . 9

2.2.2 Resource allocation to services . . . 10

III. Design of Auto-Scaler **14**

3.1 Overall Design . . . 14

3.2 Auto-scaling in the MEC environment . . . 16

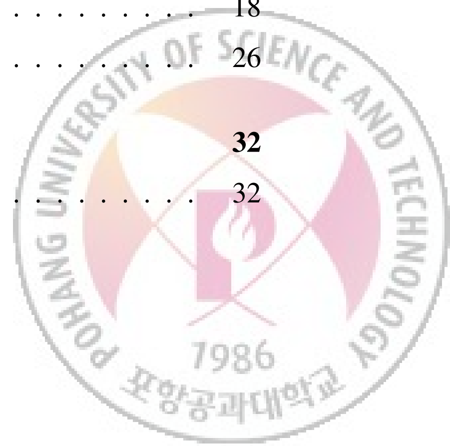
3.3 DQN-based Auto-Scaler . . . 18

3.3.1 Design of DQN Model . . . 18

3.3.2 Design of Decision Model . . . 26

IV. Implementation **32**

4.1 Monitoring Module . . . 32



4.2 NFV Orchestrator Module 33
4.3 Auto-scaling Module 34

V. Evaluation **37**
5.1 Experimental Environment 37
5.1.1 Distributed cloud environment 37
5.1.2 Edge cloud environment 38
5.1.3 Experimental Scenarios 39
5.2 Performance Evaluation 42
5.2.1 Scenario 1: Virtualized service #1 (Firewall service) 42
5.2.2 Scenario 2: Virtualized service #2 (Service Function Chain) . 45
5.2.3 Scenario 3: Containerzed service #1 (Web service) 49
5.2.4 Scenario 4: Containerzed service #2 (Video conferencing service) 52
5.2.5 Performance comparison among RL models 54

VI. Conclusion **57**
6.1 Summary 57
6.2 Contributions 58
6.3 Future work 59
6.3.1 Monitoring with low overhead 59
6.3.2 Hyper-parameter tuning 60
6.3.3 Life-cycle management of network services 60
6.3.4 Task scheduling with auto-scaling 61

Summary (in Korean) **62**

References **64**

List of Tables

2.1 ML/RL-based auto-scaling approaches 13

5.1 Environment Specification 38
5.2 Hyperparameters 42
5.3 Experiment results of auto-scaling of firewall service 45
5.4 Experiment results of auto-scaling of service function chain 49
5.5 Experiment results of auto-scaling of web service 51
5.6 Experiment results of auto-scaling of video conferencing service . . . 54

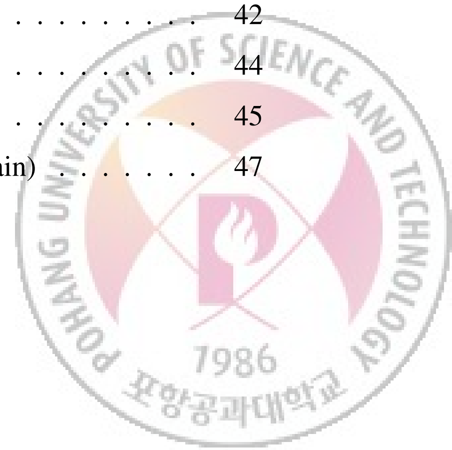
List of Figures

2.1 Service deployment in MEC 5
2.2 Microservice architecture 6
2.3 Deep reinforcement learning (DRL) architecture 8
2.4 Multi-access edge system reference architecture variant for MEC in NFV 9
2.5 Auto-scaling approaches 11

3.1 Overall design for auto-scaling 15
3.2 Service operation in the MEC environment 16
3.3 Deep Q-network (DQN) model for auto-scaling 18
3.4 Monitoring method 20
3.5 Auto-scaling process using Deep Q-network (DQN) 23
3.6 Decision model for auto-scaling 26
3.7 Decision-making sequence 31

4.1 Design of monitoring module 32
4.2 Design of NFVO module 33
4.3 Design of auto-scaler in the form of a module 34
4.4 Workflow of auto-scaling 36

5.1 OpenStack-based Testbed 37
5.2 Docker Swarm-based Testbed 39
5.3 Scenario 1: auto-scaling of firewall service 42
5.4 Experiment results of scenario 1 (firewall) 44
5.5 Scenario 2: auto-scaling of Service Function Chain 45
5.6 Experiment results of scenario 2 (service function chain) 47



5.7 Scenario 3: auto-scaling of web service 49
5.8 Experiment results of scenario 3 (web) 50
5.9 Scenario 4: auto-scaling of video conferencing service 52
5.10 Experiment results of scenario 4 (video conferencing service) 53
5.11 Reward of reinforcement learning (RL)-based auto-scaling approaches 55
5.12 Response time and the number of instances during episodes 56

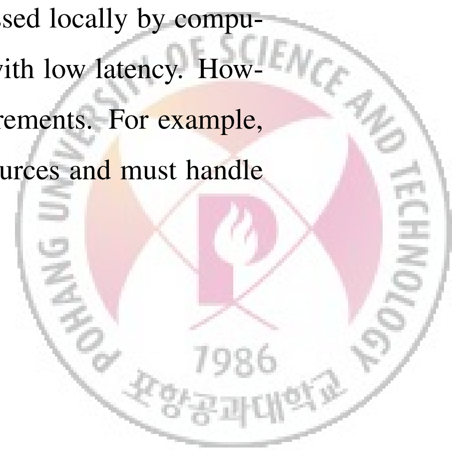
I. Introduction

1.1 Motivation and Problem Statement

1.1.1 5G network era

With the advent of 5G networks, it is necessary to provide services to numerous devices and users while meeting various quality-of-service (QoS) requirements such as high data rates and low latency. These requirements are classified by IMT-2020 of ITU-R [1] into three usage scenarios: enhanced mobile broadband (eMBB), massive machine type communications (mMTC), and ultra-reliable and low-latency communications (uRLLC). This can be realized by using network slicing [2], which creates a dedicated logical network, that is, a network slice for each service. Because networks have diverse demands, network functions and an adequate amount of resources should be allocated to network slices to satisfy the QoS requirements in response to these demands. To this end, software-defined networking (SDN) [3] and network function virtualization (NFV) [4] are used as fundamental technologies to build a network slice flexibly and rapidly.

In the era of 5G networks, global data traffic is foreseen to increase by more than 20,000 times from 2010 to 2030 [1]. It is essential to process these massive amounts of traffic effectively to reduce the generated network overhead. To this end, multi-access edge computing (MEC) [5] is a promising concept that brings down the cloud-computing center to edges near users. The MEC concept allows service providers to deploy their services at the edges to enable traffic to be processed locally by computational offloading; thus, it enables them to provide services with low latency. However, the operation of services based on MEC has strict requirements. For example, compared to a cloud-computing center, edges have fewer resources and must handle



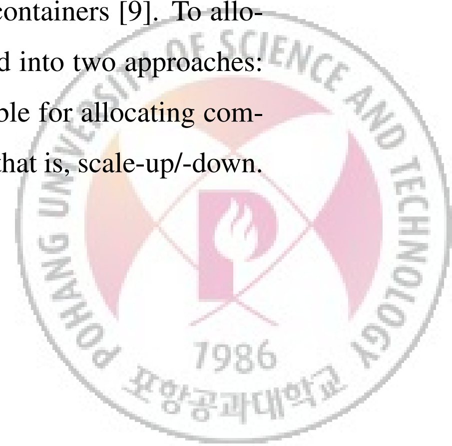
diverse devices that generate dynamic traffic to the edges. This makes it difficult for service providers to operate services manually while meeting the QoS requirements of services in a dynamic environment. Attempts to solve this problem have led to the use of machine learning (ML) to automate the operation and management of services.

ML is known as an enabling technology that offers the ability to automatically learn from data and make decisions. In the network management area, supervised and unsupervised learning have both been widely used for network management, such as traffic classification, anomaly/intrusion detection, and resource allocation [6]. However, it is necessary to collect a large amount of data that should be preprocessed (e.g., labeled) for supervised learning. Although unsupervised learning uses data that is not labeled, it can be computationally complex and less accurate. In addition, it is challenging to determine the data features for training and to obtain sufficient data that contain these features from networks. To overcome this problem, reinforcement learning (RL) [7] has attracted considerable attention.

1.1.2 The needs for auto-scaling of services

In 5G networks with an MEC scenario, it is essential to dynamically allocate the adequate amount of resources to services while satisfying the QoS requirements; therefore, many auto-scaling methods have been proposed to resize the amount of resources used by services in response to dynamic traffic. Moreover, there are various resources, such as communication resources and computation resources, used by services. In this thesis, we consider the computation resources (e.g., CPU, memory, disk, etc.) allocated to service because they significantly affect the service performance.

Service running in 5G networks consists of multiple components instantiated as independent instances that are virtual machines (VMs) [8] or containers [9]. To allocate resources to them, auto-scaling of service can be classified into two approaches: vertical scaling and horizontal scaling. The former is responsible for allocating computation resources to each instance (i.e., VMs and containers), that is, scale-up/-down.



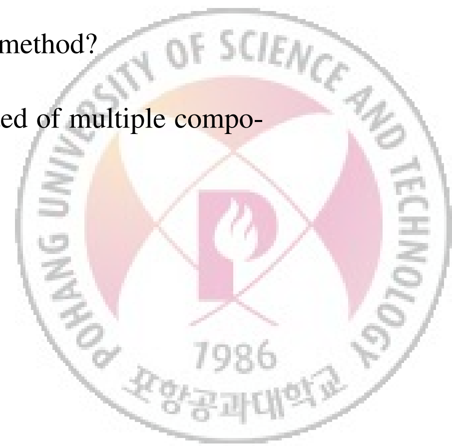
The latter resizes the number of instances composing service, that is, scale-in/-out. In general, vertical scaling causes service downtime because most operating systems do not allow resource allocation for instances without rebooting [10]. It is a major drawback for operating services that require high availability, so horizontal scaling is suitable to overcome the limitation.

With the advent of MEC, it is possible to satisfy strict QoS requirements by deploying services in edges near users. However, several factors should be considered. First, it needs to scale the proper service instance to improve the service performance while reducing operating costs. Subsequently, the scaling method should utilize efficiently the limited resources of the edges. Second, it is necessary to scale the service at the right time because inappropriate scaling may cause resource over-provisioning or performance degradation of the service. Third, it should consider the coexistence of edges and central cloud centers to deploy services in the proper sites according to their QoS requirements. In addition, it needs to choose proper nodes hosting service instances that require low latency in the edges. However, it is hard to make a scaling decision manually while considering those factors. In order to solve this problem, this thesis proposes an auto-scaling approach using deep Q-network (DQN) [11], which is a popular deep RL algorithm.

1.2 Research Goals and Approach

In this section, the research problems and goals are briefly listed. This thesis tries to answer the following key questions.

- How to automate scaling of network services in the MEC environment?
- Which factors should consider to design an auto-scaling method?
- How to apply the auto-scaling method to service consisted of multiple components?



- How to consider the trade-off between the service performance and operating costs?
- How to ease the application of auto-scaling method to the MEC environment?

Pertaining the questions, research goals give answers about the questions as follows. Specifically, it is an auto-scaling method using RL.

- A deep reinforcement learning (DRL) model for auto-scaling, that is, horizontal scaling
- A decision model to determine service components (or locations) to be scaled
- A speed-up method to quickly converge to an optimal scaling policy
- An implementation of auto-scaler running on the MEC environment

1.3 Organization

The remainder of this thesis is organized as follows. In Chapter 2, we provide the necessary background information and survey studies that are related to our approach. In Chapter 3, we present the detailed design of the DQN-based auto-scaling method. Details of the implementation are provided in Chapter 4. We evaluate and analyze the proposed approach in Chapter 5. Chapter 6 discusses the limitations of this study and future challenges that need to be addressed. Finally, we conclude this thesis with a summary and the scope for future work in Chapter 7.

II. Background and Related Work

2.1 Background

2.1.1 Multi-access Edge Computing

With the advent of the MEC concept, it is possible to provide cloud-computing capabilities at the edge of the network. It allows service providers to consider locations such as edges and central data centers for service deployment in terms of operational costs and service performance. To provide high-level guidance on how to deploy services in MEC, the European Telecommunications Standards Institute (ETSI) summarizes the key characteristics of the edges and the software architecture for the services. Figure 2.1 illustrates the service architecture of the MEC covered in the ETSI white paper [12].

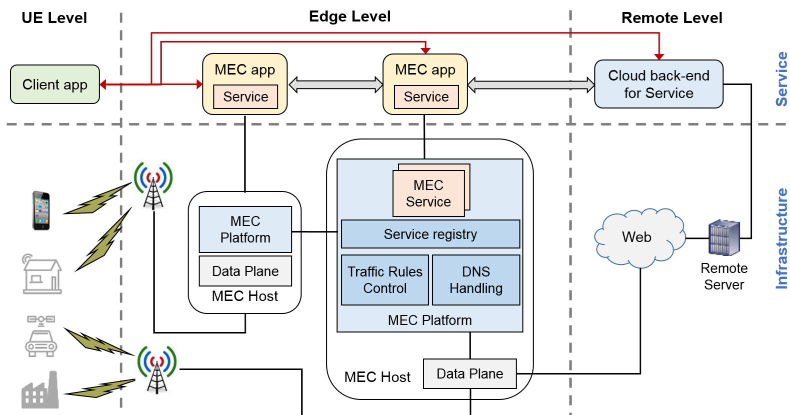
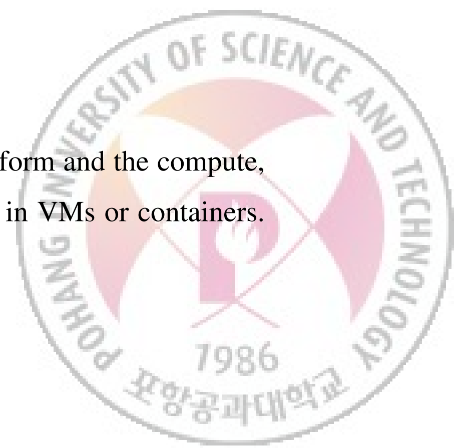


Figure 2.1: Service deployment in MEC

At the network edge, an MEC host contains an MEC platform and the compute, storage, and network resources for MEC applications running in VMs or containers.



The MEC platform is responsible for instantiating those instances (VMs or containers) and supporting MEC services, such as location awareness, bandwidth management, and user equipment (UE) identification, for the applications. An application running in this environment can be split into a few components: edge and remote components. In addition, the different components composing an application either run in a single edge or are distributed at multiple locations (e.g., neighbor edges); thus, MEC can be used to implement computation offloading techniques among all the application's components. Specifically, the edge components include a set of operations that the application performs at the edge level for low latency and predictable performance. In contrast, the remote components perform operations that require large computation resources (e.g., large storage and database access) at the remote level.

2.1.2 Microservice architecture

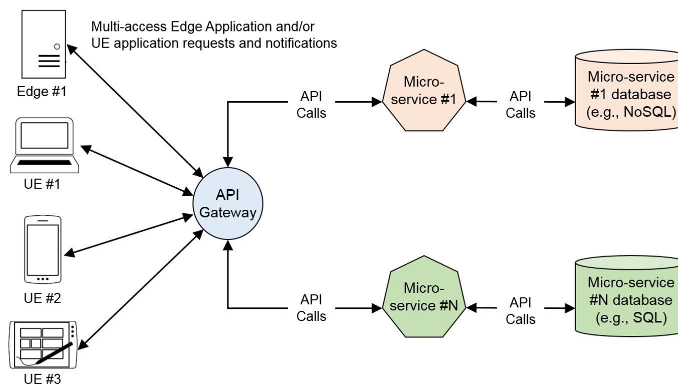
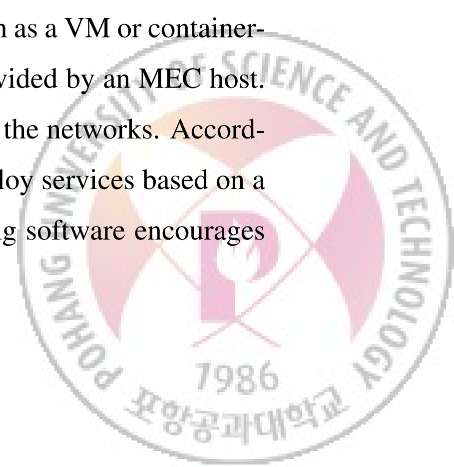


Figure 2.2: Microservice architecture

An MEC application runs as a virtualized application, such as a VM or containerized application, on top of the virtualization infrastructure provided by an MEC host. In addition, many edge clouds that employ MEC hosts exist in the networks. Accordingly, ETSI guides service providers on how to design and deploy services based on a microservice architecture. This service architecture modulizing software encourages



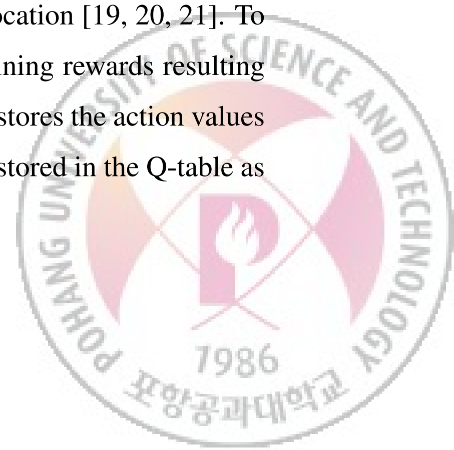
many current applications to be composed of multiple components (i.e., microservices) while placing the application's components in an appropriate location. The location can be either one of the edges or one of the MEC hosts. Figure 2.2 shows the microservice architecture presented in the ETSI white paper [12].

In this architecture, an MEC application is a loosely coupled set of microservices that interact with each other. Accordingly, it does not require all the application's components (microservices) to be refactored when the application is updated. Instead, the lifecycle of individual microservices can be handled independently, such as when requiring re-deployment, migration, or auto-scaling.

2.1.3 Deep Reinforcement Learning

RL is a promising ML method that learns through trial and error. Compared to other ML methods such as supervised and unsupervised learning, which require a large amount of training data, RL does not require training data in advance. Instead, RL learns by obtaining rewards for actions applied to the environment. RL is widely used to solve optimization problems presented as a Markov Decision Process (MDP) [13]. The MDP, which is a model for decision-making, consists of state, action, reward, and transition probability. An RL agent observes the current state in the environment and takes an action. The action carried out in each state results in a reward value and determines the next state through the transition probability. An RL agent repeats this process to learn which action results in a high reward from a given state. Thus, the optimization problem consists of steps finding a policy that can be used to make a decision in each state.

RL has been widely used to conduct network optimization tasks, such as traffic routing [14, 15, 16], task scheduling [17, 18], and resource allocation [19, 20, 21]. To solve these problems, an RL agent updates the policy by obtaining rewards resulting from actions. In general, this policy is presented as a table that stores the action values for each state. For example, Q-learning [22] uses the Q-values stored in the Q-table as



a policy. When an RL agent that uses Q-learning observes the current state, the agent selects an action that provides the highest Q-value in that state. However, although tabular representation is a simple way to present a policy, it is insufficient to solve complex problems in which numerous states and actions exist because the size of the table increases significantly.

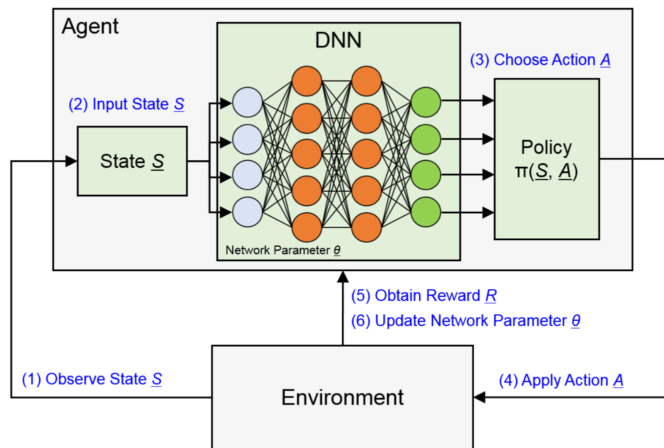
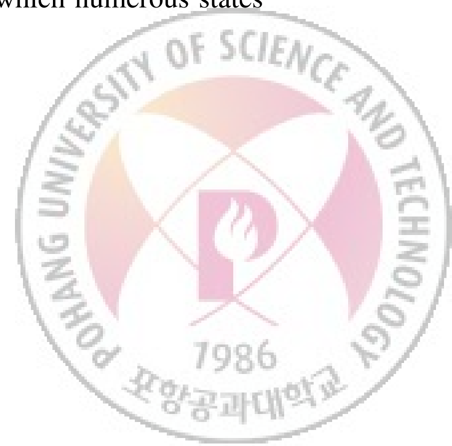


Figure 2.3: Deep reinforcement learning (DRL) architecture

Deep reinforcement learning (DRL) [23], which was proposed to overcome this limitation, uses neural networks to present a policy. Figure 2.3 illustrates the DRL architecture based on a deep neural network (DNN). With the DRL architecture, an agent observes the current state and inserts it into the DNN as input data, and the DNN outputs the action values. Based on those outputs, the agent chooses an action and applies it to the environment. When the agent obtains a reward for the action, the agent updates the network parameters of the DNN. DRL uses the DNN as a policy to enable it to accommodate complex optimization problems in which numerous states and actions exist.



2.2 Related Work

2.2.1 Service provisioning in MEC

MEC provides compute and network resources, and functions that enable MEC applications (i.e., services) to run efficiently and seamlessly in a multi-access network. It is necessary to build a fully virtualized MEC infrastructure with SDN/NFV because multiple virtual networks and services would have to be created on physical infrastructure to meet the QoS requirements of the applications. Accordingly, ETSI designed an MEC framework [24] that is responsible for managing the lifecycle of the applications on top of the virtualization infrastructure. Figure 2.4 illustrates the overall architecture of MEC framework.

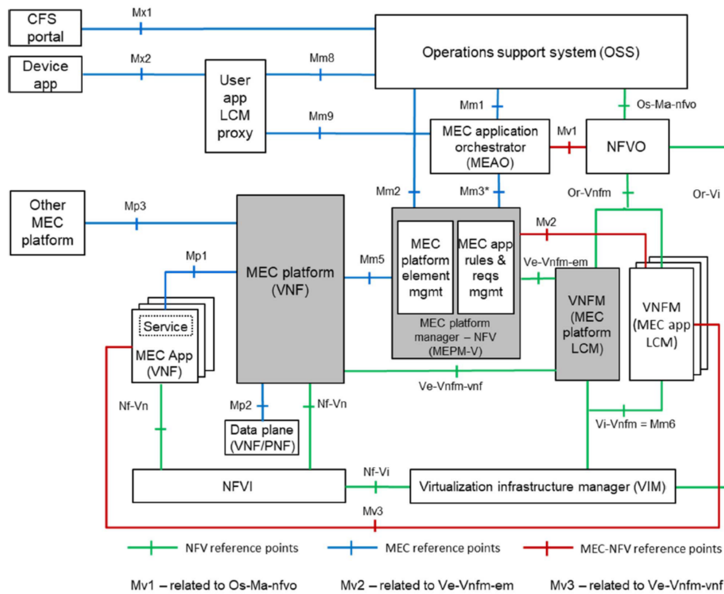
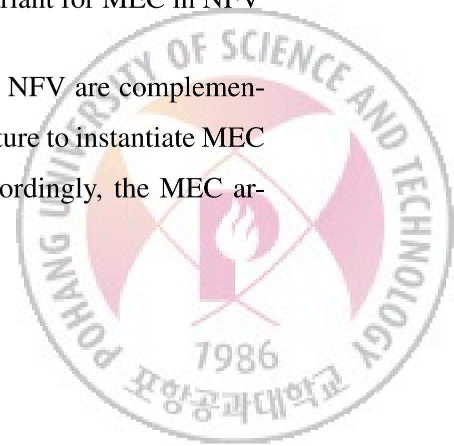


Figure 2.4: Multi-access edge system reference architecture variant for MEC in NFV

The MEC reference architecture considers that MEC and NFV are complementary concepts; thus, it enables the same virtualization infrastructure to instantiate MEC applications and virtual network functions (VNFs) [25]. Accordingly, the MEC ar-



chitecture can reuse ETSI NFV MANO components [26] to fulfill a part of the MEC management and orchestration tasks. Hence, the MEC applications running in the network edges appear as VNFs towards the ETSI NFV MANO components. In other words, a service consists of multiple VNFs, so service provisioning in MEC is an extension of VNF lifecycle management.

2.2.2 Resource allocation to services

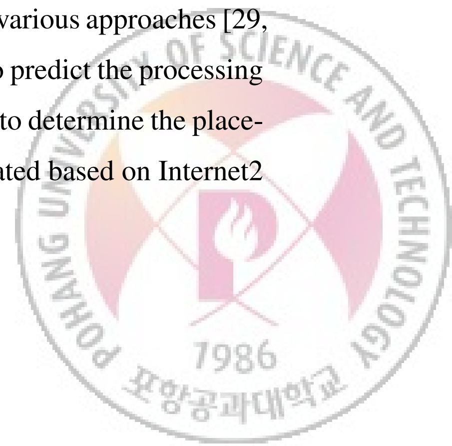
In MEC environments, edge sites are small and have limited resources compared to central cloud-computing centers. Thus, it is necessary to consider the available resource in MEC and the performance of the services that are being provided. To address this issue, researchers have proposed approaches, which can be categorized into service deployment and auto-scaling.

◦ **Service deployment**

The deployment of service, that is, the VNF deployment problem, has been studied in response to dynamic traffic change. VNF deployment determines the placement of VNFs and the initial number of VNF instances while meeting user demands. Many researchers attempted to automate the deployment process using ML [27].

ML-based VNF deployment requires training data in advance. To solve this issue, Lange *et al.* [28] used integer linear programming (ILP) to generate ground-truth labels for dynamic service request traces. This study used supervised learning that learns through traces and predicts the number of VNF instances for the service. However, this study only determined the number of VNF instances without considering VNF placement.

The placement of VNF instances was determined by using various approaches [29, 30, 31]. These studies used graph neural network (GNN) [32] to predict the processing time of VNFs running on each server. This prediction was used to determine the placement of the VNF instances. However, those studies were validated based on Internet2



network topology [33] only without considering an MEC scenario.

The advent of the MEC concept has resulted in the development of new approaches [34, 35] to deploy VNF instances in the MEC environment. These studies deployed VNFs on edges while minimizing latency and resource usage. In addition, an issue, which fewer resources are available in the MEC environment compared to cloud-computing centers, was addressed [36, 37]. Those studies considered budget-constrained and resource-constrained edges to coordinate many services on the environments.

○ Service auto-scaling

Although the above-mentioned studies provided directions for instantiating services with an adequate amount of resources, it is necessary to satisfy the QoS requirements of running services. Attempts to solve this problem gave rise to studies on the auto-scaling method that dynamically allocates computing resources to services. Specifically, this thesis focuses on a horizontal scaling approach, which allocates resources to services without downtime.

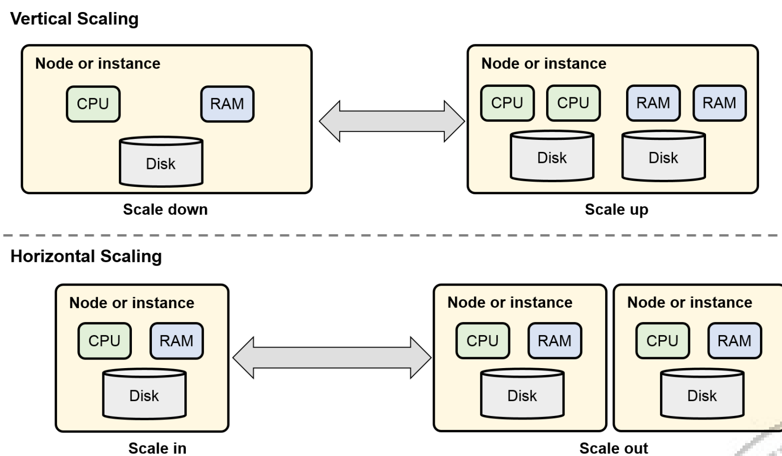


Figure 2.5: Auto-scaling approaches

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

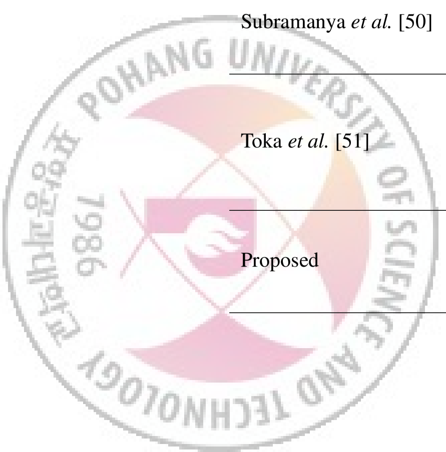
Solving the auto-scaling problem necessitates the assignment of an appropriate number of VNF instances to the service while minimizing resource usage and meeting QoS requirements. Moreover, the amount of resources required by the VNF instance depends on the type of VNF. Therefore, allocating the appropriate amount of resources to the VNF instance requires detailed knowledge of the VNF. Because this complicates manual scaling, Ghanbari *et al.* [38] and Evangelidis *et al.* [39] proposed performance models to predict the resource usage of VNFs. This prediction was used to dynamically resize the number of instances.

In addition, there have been attempts to use ML algorithms to make the performance models used for auto-scaling. Muhammad *et al.* [40] used neural networks and a linear regression algorithm to predict a post-scaling state. In another approach, Waheed *et al.* [41] used supervised learning to predict the appropriate resource of a multi-tier application and allocate the resources to each tier. However, it is hard to collect a huge amount of training data for those approaches, and the prediction accuracy highly depends on the data. Services consisted of multiple virtualized components are deployed in the 5G networks; therefore, Yazdanov *et al.* [42] and Wang *et al.* [43] proposed auto-scaling methods applied to a multi-tier application. Those studies used RL that is suitable to make a scaling decision in dynamic environments.

Many auto-scaling methods have used ML/RL for auto-scaling. Table 2.1 summarizes characteristics of some of them published within the last five years and the proposed approach. It shows that scaling service appeared as multiple microservices in the MEC environments has attracted attention. In addition, it is necessary to handle VMs and containers concurrently because microservices are instantiated through them according to the service characteristics.

Table 2.1: ML/RL-based auto-scaling approaches

Reference	Approach	Target Service	Target Instance	Target Environment	Objectives	Experimental Platform
Wang *et al.* [43].	Q-learning, DQN	Web server	VM	Centralized (Cloud)	CPU utilization, Costs	Simulation
Dezhabad *et al.* [44]	Q-learning	Firewall	VM	Centralized (Cloud)	CPU utilization, Response time	Simulation
Benifa *et al.* [45]	SARSA	Web server	VM	Centralized (Cloud)	CPU utilization, Throughput, Response time	Simulation
Rossi *et al.* [46]	Q-learning	Web server	Container	Centralized (Cloud)	Response time, CPU utilization, Costs	Docker Swarm
Rahman *et al.* [47]	Random Forest, Support Vector Regression, DNN	Microservice	Container	Centralized (Cloud)	CPU utilization, Latency	Kubernetes
Lee *et al.* [48]	DQN	Service Function Chain	VM	Centralized (Cloud)	Response time, Costs	OpenStack
Yuan *et al.* [49]	Genetic algorithm	Microservice	VM	Distributed (Edges)	CPU utilization, Latency, Costs	OpenStack
Subramanya *et al.* [50]	DNN	Service Function Chain	VM, Container	Distributed (Edges)	Latency, CPU utilization, Link utilization	Simulation
Toka *et al.* [51]	LSTM, HTM, SARSA, Q-learning	Web server	Container	Distributed (Edges)	Packet loss rate, Costs	Kubernetes
Proposed	DQN	Miicroservice	VM, Container	Distributed (Edges)	Packet loss rate, Response time, Costs	OpenStack, Docker Swarm



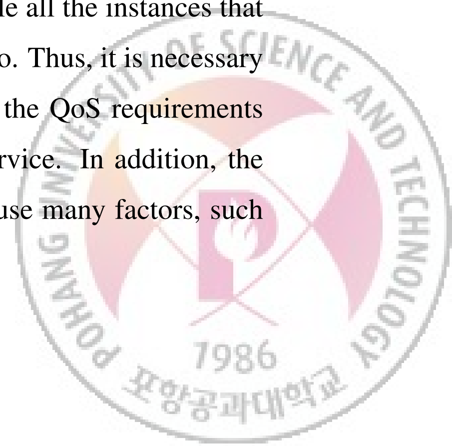
III. Design of Auto-Scaler

3.1 Overall Design

The proposed approach scales the number of instances assigned to network services running in an MEC environment. According to the service design [12] for the MEC scenario, the service consists of multiple components. These can be deployed independent of one another in instances (VMs or containers) and interact with each other through an application programming interface (API). To make clear terms used in the following sections, we describe them as below.

- **Service:** It is an application composed of multiple components (e.g., microservices). The service is responsible for processing requests of users who connect to one of the edges and use the service.
- **Instance:** It is the smallest component, which can be a VM or container, composing the service. In this thesis, scaling is applied to a specific type of instance instead of resizing all instances. An instance performs a single task and does not depend on other types of instances [52].
- **Instance type:** Multiple types of instances compose service. Each type supports a different functionality and can have multiple instances belonging to the same type. Instances of the same type handle the task with load balancing.

When a service needs to be auto-scaled, attempting to scale all the instances that compose the service is not an efficient way in the MEC scenario. Thus, it is necessary to scale a few instances only that need to be resized to meet the QoS requirements while maintaining the appropriate number of instances for service. In addition, the MEC environment complicates the auto-scaling problem because many factors, such



as the placement of instances and limited resources of edges, need to be considered for the operation of services. Although those factors can be considered within a DQN model [48], it complicates the DQN model. Specifically, various actions (output layer neurons) that require more trials through exploration should be in the output layer of the DQN model; therefore, it is time-consuming to find the optimal policy.

To solve this problem, we design the DQN and decision models separately. These models complement each other and are able to make scaling decisions in the MEC environment. Figure 3.1 illustrates the overall design of the auto-scaling method for the service.

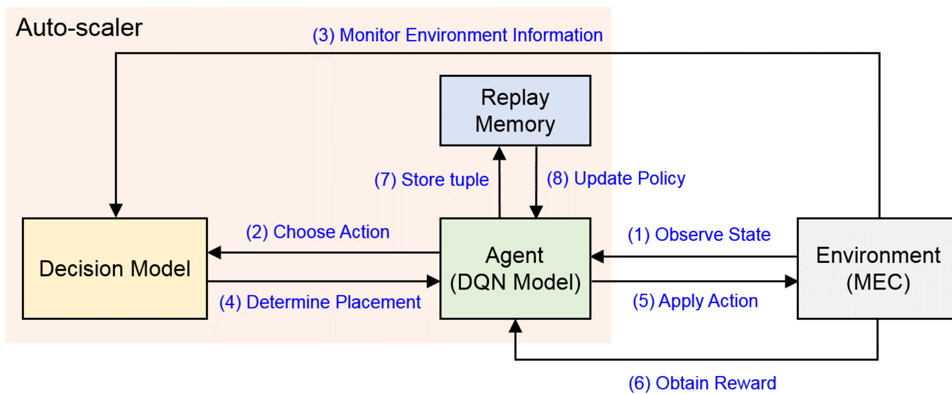
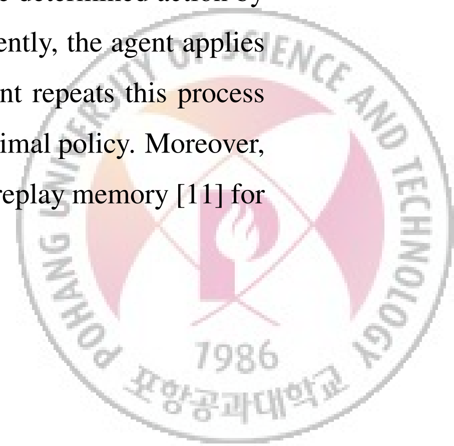


Figure 3.1: Overall design for auto-scaling

An agent periodically observes the state of the service and chooses an action to resize the number of instances. When the agent decides to add/remove an instance, the decision model selects an instance type to be resized and determines where to attempt the action. In other words, the decision model complements the determined action by using the information monitored in the environment. Subsequently, the agent applies the action to the environment and obtains a reward. The agent repeats this process and updates the network parameters of the DQN to gain the optimal policy. Moreover, our approach uses a mini-batch sampling by interacting with a replay memory [11] for



stable training of the DQN model.

3.2 Auto-scaling in the MEC environment

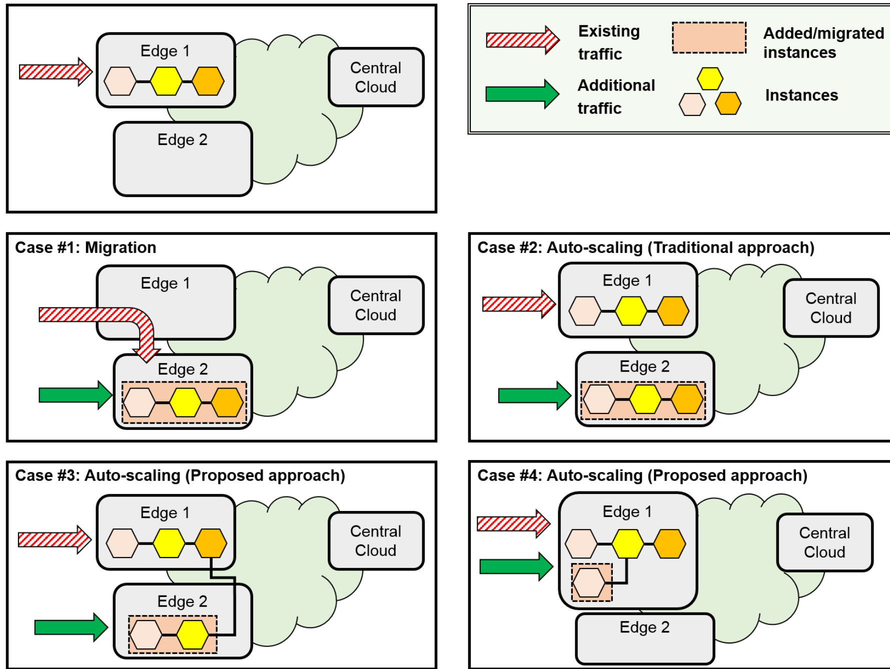
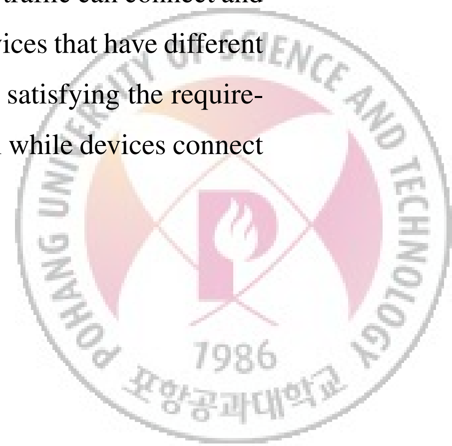


Figure 3.2: Service operation in the MEC environment

In the MEC environment, an edge can provide not only access services for broadband and mobile networks but also cloud services [53]. In addition, the service's components are placed in proper edges according to the service requirements [12]. For using those services, various devices that generate dynamic traffic can connect and move among edges [53, 54]. Because those devices request services that have different QoS requirements, it is necessary to operate the services while satisfying the requirements. Figure 3.2 illustrates four cases for the service operation while devices connect to an edge.



The first case is to migrate service instances to another edge near to users. Because of the mobility that is an essential scenario of 5G, some devices that use the services can move to a neighbor edge. To address this issue, ETSI defines use cases to support the mobility [55]. Moreover, it is necessary to satisfy the QoS requirements of service while migrating instances among edges [56]. For example, if UEs connecting to edge 1 move edge 2 and require the low-latency service, all instances running in edge 1 can migrate to edge 2. In addition, traffic steering is essential for UEs accessing edge 1 and using the service because the service instances are migrated to edge 2. However, this service migration cannot guarantee to provide service with low latency to the UEs of edge 1. In addition, there is a scalability issue because it is hard to frequently migrate the service while identifying the numerous UEs that use the service.

In another case, there is a traditional auto-scaling approach that adds all instances of the service. For example, this approach creates new instances in edge 2. This method is common and simple if the service consists of tightly coupled instances. However, it inefficiently consumes resources to perform the scaling.

The proposed approach adds minimum instances in the most appropriate location to overcome the limitation of the traditional method. For example, this method creates instances, which process tasks that require low latency, in the edge in response to additional traffic. In addition, it reuses other instances running in a neighbor edge. Therefore, our approach allows service provides to exploit the geographical distribution to serve different user populations (i.e., traffic) while reducing scaling costs. In addition, all instances of service can be placed in the same location (e.g., single edge only). In this case, our approach resizes instances while determining the node (server) to host the scaled instances instead of choosing one of the edges.

3.3 DQN-based Auto-Scaler

3.3.1 Design of DQN Model

○ DQN model architecture

An agent that uses DQN is an auto-scaler that resizes the number of instances for a service. Accordingly, the auto-scaling problem must be defined as an MDP composed of a state, action, and reward. The proposed auto-scaling approach using DQN inputs a current state and chooses a scaling action, such as *Add*, *Maintain*, and *Remove*. Figure 3.3 illustrates the auto-scaling DQN model.

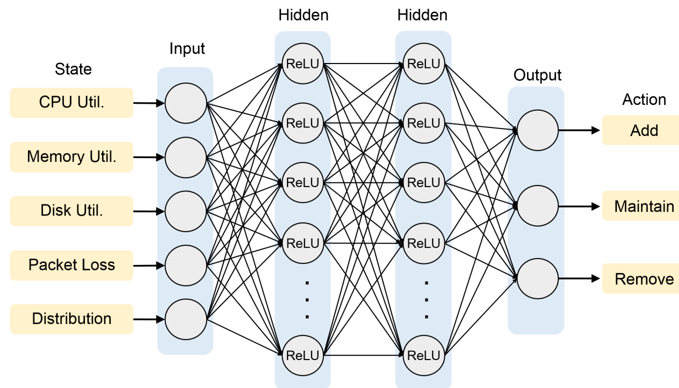
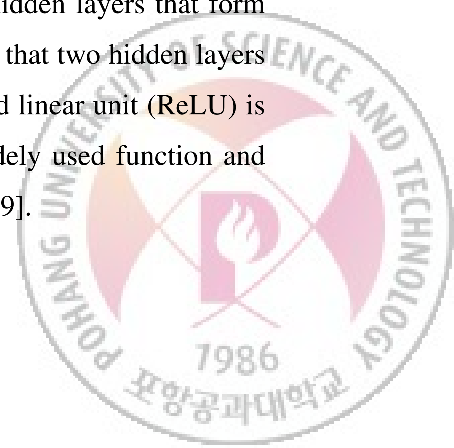


Figure 3.3: Deep Q-network (DQN) model for auto-scaling

Our DQN model inserts a state at the input layer and outputs the action values (Q-values) at the output layer. Additionally, the DQN model should include hidden layers, but there are no definitive answers on how to determine the number of hidden layers. In general, one or two hidden layers are sufficient for the large majority of problems [57, 58]; therefore, our DQN model includes two hidden layers that form a feed-forward neural network (FNN). We empirically derived that two hidden layers work well for the auto-scaling problem. In addition, a rectified linear unit (ReLU) is used as an activation function because ReLU is the most widely used function and performs better than other activation functions in most cases [59].



○ **Data features for state of DQN model**

The proposed DQN model is responsible for selecting a scaling action to meet the QoS requirements of a service from a given state. In other words, the DQN model resizes the number of instances if the current state leads to the performance degradation of the service. Accordingly, a state consists of metrics that affect service performance. Because the time consumed for packet processing and propagation is the dominant QoS metric for the service running in a network, a state consists of five elements that affect the performance: resource utilization (CPU, memory, disk), packet loss rate, and distribution of instances of the service.

Among the resources, the instances consume CPU and memory capacity to process incoming packets. Therefore, these resources are indicators of the availability of instances for packet processing. Moreover, high memory utilization can lead to a swapping activity in which the operating system attempts to free up memory by migrating pages from memory to disk [42]. Because disk access is slower than memory access, these I/O operations cause bottlenecks that delay packet processing. Thus, the number of disk operations was used to define the state in terms of disk utilization.

The DQN model also considers the packet loss rate as an indicator of service availability. For example, if the packet loss rate is high owing to the overload of instances, the DQN model would have to apply an action to add a new instance. Finally, the DQN model regards the distribution of service, which is a value that indicates the placement density of instances. The value thereof is small if the instances run on multiple locations. In general, instances of the same type process packets by using load balancing. Therefore, packets have different paths to those instances, which affects the packet propagation time. To this end, the DQN model uses the distribution of the service to define a state.

○ **Data collection from environment**

While service is being auto-scaled, the agent periodically observes the state and delivers it to the DQN model. Thus, the agent is responsible for creating the input data, i.e., the state, from the monitoring metrics observed in the environment. Figure 3.4 shows the monitoring process the agent uses to create the input data.

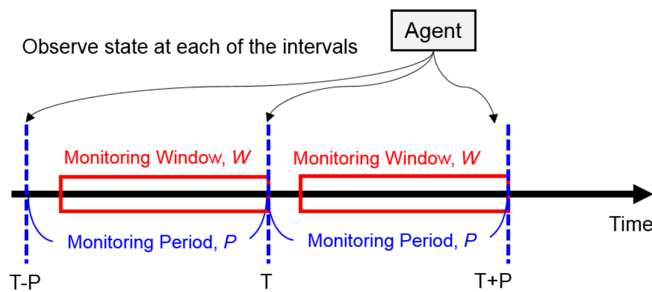
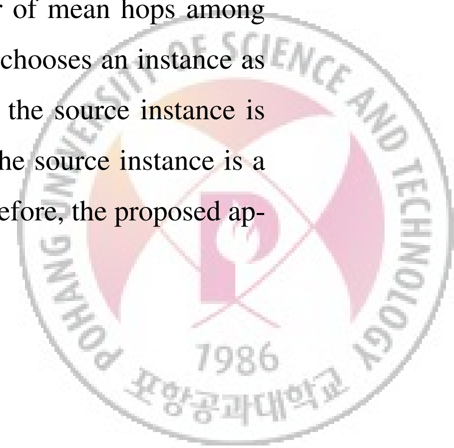


Figure 3.4: Monitoring method

The agent obtains monitoring metrics at every interval, P, and performs scaling. Thus, it is necessary to assign a longer time than the cool-down time, which waits for the effect of the previous scaling, to P. The interval P, which is longer than the cool-down time, prevents oscillation in terms of the number of instances during auto-scaling. The agent calculates the average value for the resource utilization and packet loss rate from the data collected during the monitoring window, W. The number of instances does not change during the monitoring window because W is shorter than P. Therefore, instead of calculating the average value, the agent obtains the distribution value at every P. The distribution value is the number of mean hops among the instances of the service. To compute this value, the agent chooses an instance as a source and counts the hops from the source node in which the source instance is running to other nodes that host other instances. In general, the source instance is a service component that receives and parses user requests. Therefore, the proposed ap-


POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

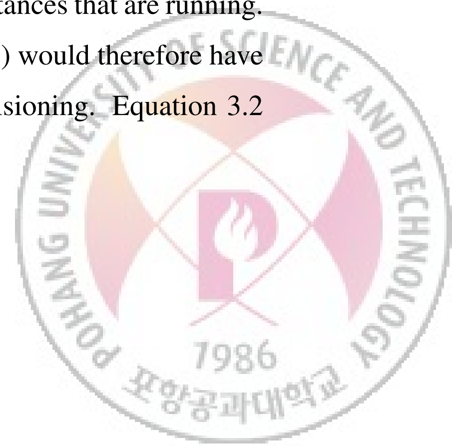
proach can consider not only the distribution of instances within a single edge but also among distributed edges. According to the available resources of edges and the QoS requirements of the service, our method chooses a proper location to scale the service instances.

$$Average = \frac{\sum_{n=1}^{N}\sum_{m=1}^{M} data_m}{N \times M}, \quad Dist = \frac{\sum_{i=1}^{N} hop(Node_{src}, Node_{inst_i})}{N} \tag{3.1}$$

The average values and the distribution value can be calculated using Equation 3.1, where *N* is the number of total instances comprising the service, and *M* is the number of monitoring data values collected during *W*, i.e., $data_m$. The *Average* calculates the mean values per CPU, memory, disk, and packet loss rate. Moreover, *Dist* outputs a distribution value in which $Node_{src}$ is the node running the source instance, and $Node_{inst_i}$ is the node running *i*th instance.

○ **Design of Reward Model**

The agent obtains rewards from the environment as a result of its actions; thus, a reward model must be defined to calculate the values of rewards. Our proposed approach is designed with the aim of operating the service while meeting QoS requirements and minimizing operating costs. The proposed reward model considers the packet loss rate and response time to evaluate compliance with the QoS requirements. These two metrics are the most important to determine the availability and performance of the service in the networks. In addition, the reward model uses the number of instances to obtain the operating costs for the service because most infrastructure-as-a-service providers determine the cost based on the number of instances that are running. A service provider who pays for the use of resources (instances) would therefore have to be careful not to increase the operating costs by over-provisioning. Equation 3.2



expresses our reward model.

$$Reward = -w_1 \ln(1 + resTime) - w_2 \ln(1 + Loss) - w_3 \ln(1 + Instance) \quad (3.2)$$

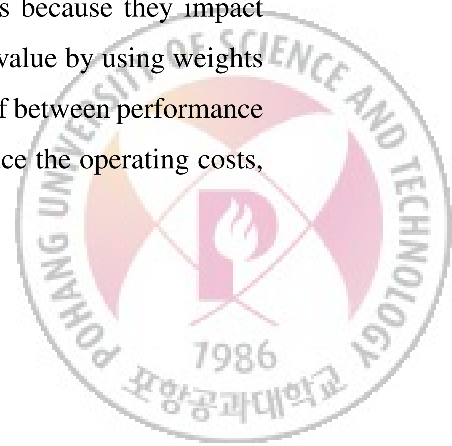
This model considers the response time (*resTime*) to estimate the service performance and the packet loss rate (*Loss*) to identify the service availability. Moreover, the model uses the number of instances (*Instance*) to present the operating costs. We use a natural logarithm that adjusts these considerations as expressed in the above equations to output the reward.

$$resTime = Response\ time \times 10^{-3},\ Loss = \frac{Dropped\ packets}{Total\ packets} \quad (3.3)$$

$$Instance = \frac{Current\ instances}{Maximum\ instances} \quad (3.4)$$

In Equation 3.3 considering the service performance and availability, *Response time* is the mean response time measured in milliseconds, and the model converts the time to seconds by multiplying 10^{-3}. Specifically, the response time is the consumed time it took for a user to send a request message to the service and receive a response message. *Loss* is a percentage of the number of dropped packets among the total packets processed in the service. Moreover, *Instance* expressed in Equation 3.4 is a percentage of the number of running instances based on the maximum number of instances. We assume that there is an upper bound about the number of instances assigned to the service because instances can be added infinitely until satisfying the QoS requirements if there is no limitation.

Owing to the natural logarithm, our DQN model intends to keep the short response time, small packet loss rate, and small operating costs because they impact more on increasing the reward values. Finally, we adjust each value by using weights such as w_1, w_2, and w_3 ($w_1, w_2, w_3 \geq 0$) to consider the trade-off between performance and operating costs. For example, if the model intends to reduce the operating costs, w_3 has a larger value than w_1 and w_2.



◦ **Application of DQN-based Auto-Scaler**

The policy that is used to determine an action is updated using a trial-and-error approach while obtaining rewards. In other words, the agent trains the DQN model by sequentially using the data collected as a result of the actions. However, this sequential data can negatively affect the identification of the optimal policy. Because the training data collected from the environment sequentially over time are highly correlated, this correlation destabilizes the learning process of the DQN. To prevent this problem, the agent runs with replay memory [11]. Figure 3.5 illustrates the DQN learning process.

$$L_i(\theta_i) = E_{(s,a,r,s')}[(r + \gamma maxQ(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2] \tag{3.5}$$

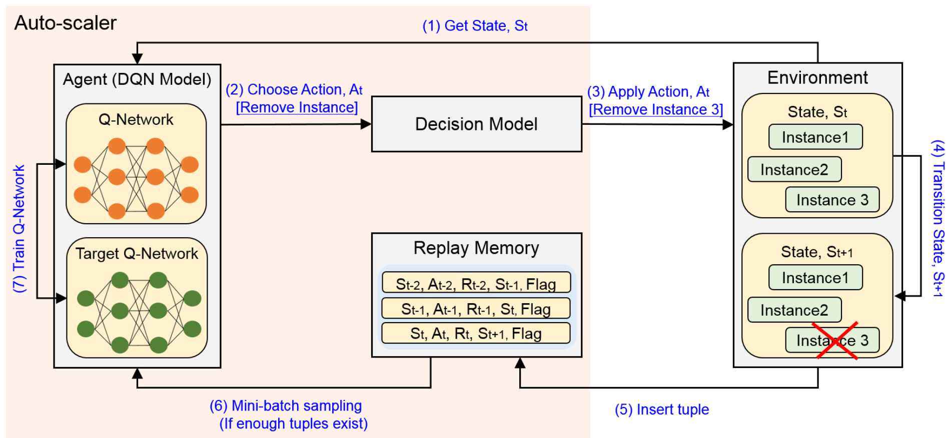
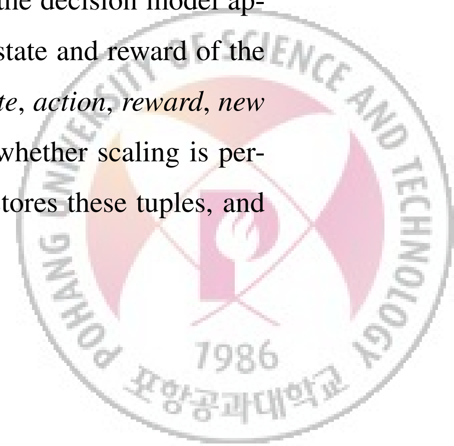


Figure 3.5: Auto-scaling process using Deep Q-network (DQN)

Whenever a DQN agent chooses an action in each state, the decision model applies the action to the environment. After observing the next state and reward of the action, the agent creates a tuple that consists of the *current state*, *action*, *reward*, *new state*, and *flag*. Among them, the *flag* is a value that checks whether scaling is performed successfully in an environment. The replay memory stores these tuples, and



the agent learns by using mini-batch sampling if sufficient tuples are available. During this process, we employed the target Q-Network [11] to stabilize the training. The target Q-Network is used to back-propagate the DQN to minimize a loss function shown in Equation 3.5 and train the main Q-Network. Q-values at iteration i is updated by the equation in which r is a reward value and γ is a discount factor. The target Q-Network's parameters (θ_i^-) are not trained, but they are periodically synchronized with the parameters of the main Q-Network (θ_i). Hence, the replay memory and target Q-Network enable the DQN model to learn stably and prevent the correlation of datasets. Moreover, the DQN model uses the ϵ-greedy algorithm, which determines the course of action (i.e., exploitation or exploration) based on the ϵ-value. If the ϵ-value is large, the scaling action is randomly selected.

Algorithm 1 shows the proposed DQN-based auto-scaling algorithm. Because the purpose of our auto-scaling method is dynamically resizing the number of instances in service, a scaling thread is created and applied to the service. While the service to be scaled is running in the environment, the DQN-based auto-scaling method periodically triggers a scaling action to the service until the scaling thread is expired.

Algorithm 1: DQN-based auto-scaling algorithm

```
Initialize replay memory M, Q-network Q, Target Q-network Q̂
while Service_status = Running and episodes < limit do
    State_current ← state_pre_processor(Service_info)  // Create a state
    Action ← get_scaling_action(ϵ)        // Decide a scaling action
    if Action = Add or Remove then
        Target ← decision_model(Action)          // See Section 3.3.2
        Flag ← scaling(Target)     // Flag is 0 when scaling fails
    else
        Flag ← 1              // Flag is 1 when scaling successes
    Sleep(Cooldown_time)
    State_next ← state_pre_processor(Service_info)
    Reward ← reward_calculator(Service_info)     // See Equation 3.2
    Store Tuple(State_current, Action, Reward, State_next, Flag) in M
    if Enough tuples exist in M then
        Train Q              // Update Q according to Equation 3.5
        if n_epi % update_interval = 0 then
            Copy network parameters of Q to Q̂            // Synchronize
    episodes ← episodes+1
    Sleep(interval)                // Repeat scaling after sleeping
```

3.3.2 Design of Decision Model

○ The needs for decision model

Scaling all instances of service is expensive and time-consuming; thus, instances only that significantly affect the service performance should be adjusted. In addition, the placement of instances affects the service performance. Therefore, it is necessary to consider where scaling is performed. To this end, the proposed decision model is responsible for choosing the type of instance to be scaled and where to scale them. Figure 3.6 shows that a decision model applies the scaling action to the environment.

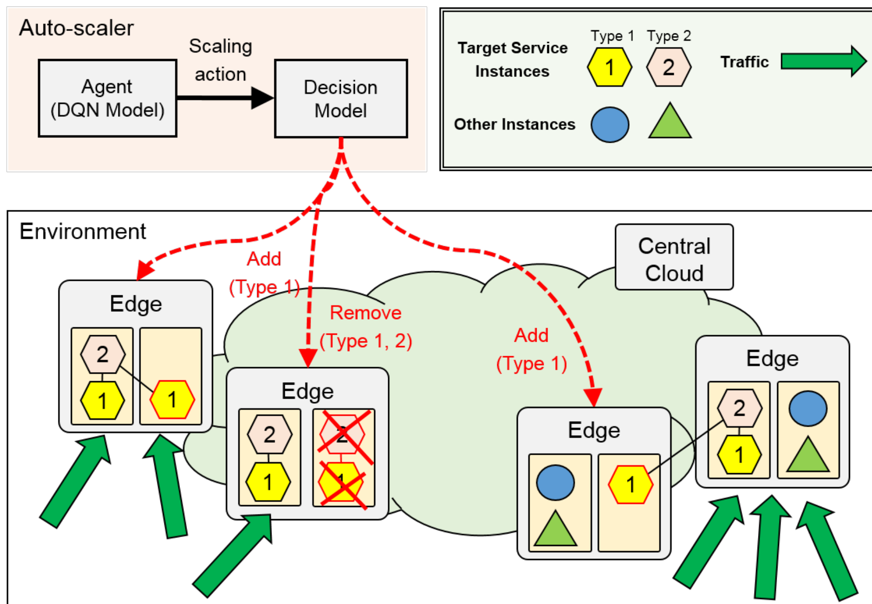
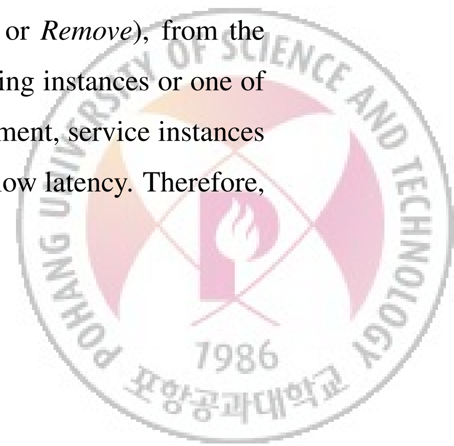


Figure 3.6: Decision model for auto-scaling

The decision model takes an input, scaling action (*Add* or *Remove*), from the agent; subsequently, this model chooses either one of the running instances or one of the nodes according to the scaling action. In the MEC environment, service instances are usually placed in a single edge (i.e., local edge) to provide low latency. Therefore,



the decision model chooses an instance to be removed or a node hosting a new instance within the edge to maintain an adequate number of service components in response to traffic processed by the service. However, the decision model should consider deploying a new instance in a neighbor site when the local edge has insufficient resources. In this case, the decision model determines an edge that is the nearest neighbor edge having sufficient resources to host the instance first. After that, the model chooses one of the nodes within the edge.

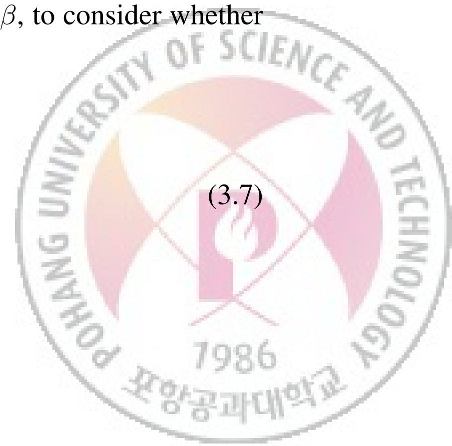
◦ **Decision making to choose type of instance**

The decision model estimates the score for each type of instance by using Equation 3.6 and selects the one with the highest score. This formula consists of a binary variable, *mask*, and a score function, $f(Type_i)$. The *mask* variable indicates whether it is possible to adjust the instances of the type. If insufficient instances are available for scale-in (e.g., one instance in the type) or no computation resources that are allocated to instances are available for the scale-out, the *mask* is zero. Therefore, the variable sets the score to zero for the type. When scaling is possible for the type, the value of *mask* allows the equation to estimate the score.

$$score = mask \times f(Type_i) \tag{3.6}$$

The score function calculates the score by obtaining the resource usage (*resUtil*) shown in Equation 3.7. Among the resources, the CPU and memory directly affect the processing of packets by instances (e.g., VMs or containers) belonging to the same type; thus, the decision model measures the CPU score and memory score as in Equation 3.8. In addition, each score is adjusted by weights, α and β, to consider whether the service is memory-intensive or CPU intensive ($\alpha, \beta \geq 0$).

$$resUtil = \alpha CPU_{Score} + \beta MEM_{Score} \tag{3.7}$$



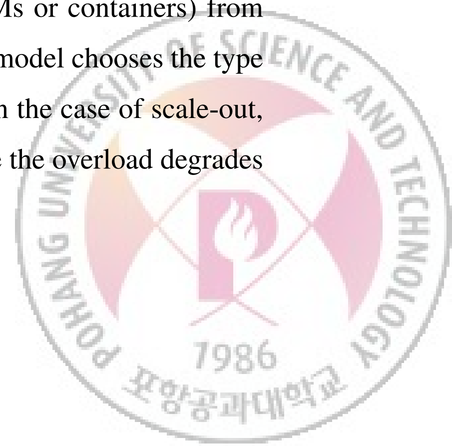
The decision model measures the scores using the average values. Note that instances belonging to the same type are allocated the same amount of resources (CPU cores and memory capacity). In other words, the amount of resources allocated to instances are specified in advance to provide enough computing power to meet the QoS requirements. For example, virtualization platforms hosting VMs and containers support template files and APIs to allocate a specific amount of resources to instances.

$$CPU_{Score} = \frac{CPU_{Util}}{CPU_{Cores}}, \quad MEM_{Score} = \frac{MEM_{Util}}{MEM_{Total}} \tag{3.8}$$

First, the decision model estimates the CPU score by measuring the average CPU utilization of the instances belonging to the same type. The CPU utilization is presented by a percentage of CPU usage. The CPU score is high if the average CPU utilization is high or the number of CPU cores allocated per instance of the type is small. For example, when instances with a small number of CPU cores are highly utilized, a high CPU score is obtained. Additionally, the memory score is calculated by estimating the average memory utilization of instances belonging to the same type and the memory capacity (i.e., bytes) allocated per instance of the type. The memory utilization is presented by a percentage of memory usage that is the used memory (bytes) based on the total memory capacity (bytes) of the instance.

$$f(Type_i) = \begin{cases} Dist_i \times e^{-resUtil} & \text{(scale-in)} \\ \frac{1}{Dist_i} \times e^{resUtil} & \text{(scale-out)} \end{cases} \tag{3.9}$$

Based on each score, the decision model measures the total score for each type by using Equation 3.9 while considering both scale-in/-out cases. In the case of scale-in, the decision model removes one of the running instances (VMs or containers) from the type to reduce the operating costs of the service. Thus, the model chooses the type of instance which has more resources with lower utilization. In the case of scale-out, it adds a new instance to mitigate overloaded instances because the overload degrades the service performance.



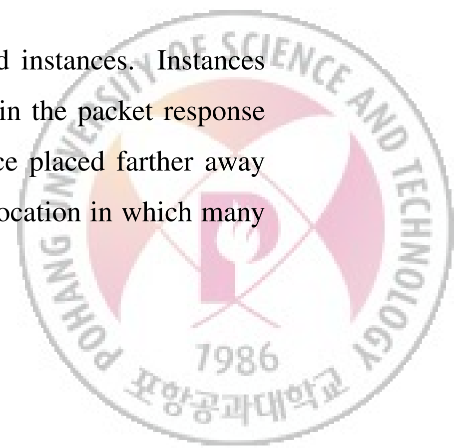
In addition, the distribution of instances that affects packet propagation time is considered when choosing the type of instance. To this end, the decision model selects the type of instance that would need to be scaled. In the scale-in case, the score function estimates a high score if instances belonging to the type have a large amount of resources but low utilization, or if the instances are widely distributed. In contrast, the score for scale-out is high when instances belonging to the type have a small amount of resources but high utilization, or when the instances are close to each other. To compute the distribution value for each type, we use Equation 3.1. Finally, the decision model selects the type with the highest score.

○ **Decision making to choose the placement**

After choosing the instance type to be scaled, the decision model selects the location to perform the scaling. Specifically, it selects one of the running instances for scale-in and adds a new instance to the proper location for scale-out. The decision model lists the candidates in terms of their packet loss rate, distribution, and available resources to determine the instance or location.

The packet loss rate of each instance significantly harms the service availability; thus, it is necessary to mitigate the problem by scaling. To this end, the decision model identifies whether an instance with high resource utilization and high packet loss rate in the scale-out case. It indicates the instance to be generating high overload. Thus, this model selects the location (i.e., node) to create a new instance that load-balances the overload. In addition, in the case of an instance with low resource utilization and low packet loss rate, the decision model removes the instance to prevent over-provisioning. Hence, an instance and a location (node) would be candidates according to the packet loss rate for scale-in/out.

The distribution considers the number of densely placed instances. Instances that are running close to each other can reduce the variance in the packet response time caused by packet propagation. For example, an instance placed farther away would be a candidate for removal by scale-in. In addition, a location in which many



instances are working in high density could be a candidate for the addition of a new instance in the case of scale-out. However, if the decision model were to only consider the distribution, scaling decisions would be greedy without considering the available resources in the MEC environment. This would depend on the initial placement of the instances, thus instances could be concentrated in specific locations.

To solve the problem, the decision model would have to consider the available resources at each location to enable multiple services to cooperate. For scale-in, the decision model selects an instance located at the edge, where few available resources exist, to release the resources allocated to the instance. For scale-out, a location with sufficient available resources could be a candidate for the addition of a new instance.

Based on each factor mentioned above, the decision model lists candidates according to the priorities that need scaling and estimates a score per candidate. For example, a candidate with high priority has a high score. To choose an instance (scale-in) or location (scale-out), the decision model sums the scores estimated from each factor and selects one with the highest score. Finally, the scaling decision is carried out in the environment. Figure 3.7 illustrates the overall auto-scaling process using the decision model.

State
Agent
(DQN Model)
Scaling action
(Scale-in/out)
Environment
Central
Cloud
Edge
Node
Node
Node
Edge
Node
Node
Node
Monitoring metrics
Decision model
Decision making #1 (Instance type)
Estimate Score
Estimate all instance's types
Scaling
possible?
Yes
Save
Score list
Select Instance type
Instance Type to be scaled
Decision making #2 (Location/instance)
Packet Loss Rate
Distribution
Available Resources
List candidates
Location/
instance list
Location/
instance list
Location/
instance list
Consider factors to make a decision
Decision
Scaling
possible?
Yes
Instance to be removed
or location to host a new instance
Apply scaling decision

Figure 3.7: Decision-making sequence

IV. Implementation

Many edge computing systems rely on OpenStack to build VIM and MANO because it provides various functionalities for managing the MEC architecture [60, 61]. Thus, we first targeted to implement the auto-scaler in the form of a module running on OpenStack. Additionally, service running in edges can be instantiated through containers; therefore, we also designed the module to make it interact with a container orchestration platform, that is, Docker swarm [62]. The implemented auto-scaler consists of three modules: monitoring, NFV orchestrator (NFVO), and auto-scaling. All the code developed and used in this implementation can be found on GitHub [63].

4.1 Monitoring Module

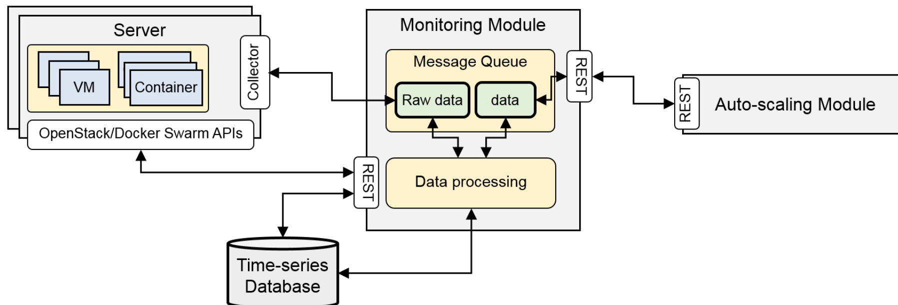
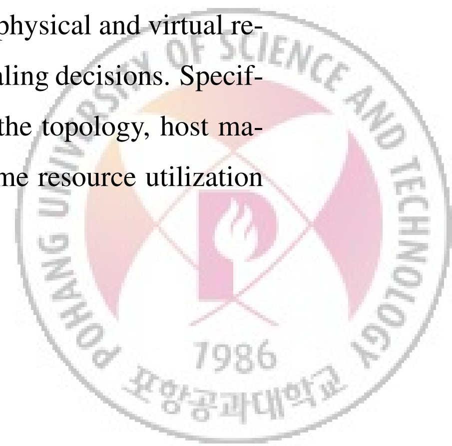


Figure 4.1: Design of monitoring module

A monitoring module observes the environment in which physical and virtual resources coexist and provide data that can be used for making scaling decisions. Specifically, this module provides not only static information (e.g., the topology, host machine specification, and instance specification) but also real-time resource utilization



(e.g., CPU, memory, disk, bandwidth usage). The monitoring module continually interacts with servers that are host machines in the environment. Figure 4.1 illustrates the design of the monitoring module.

In our implementation, static information is retrieved from OpenStack and Docker swarm via REST APIs. Real-time resource utilization is collected via the collector, which is deployed in every server. We used Collectd [64], a daemon that collects, transmits, and stores performance data on computers and network equipment because this collector is fast and does not consume much resources [65]. Specifically, each server's Collectd in our environment uses a small amount of resources, CPU 0.7% and memory 0.1% on average.

The collector collects raw data and sends the data to the message queue of the monitoring module. We collected the raw data every 1 s because it is necessary to provide real-time and fine-grained data for the auto-scaling module. Data processing in the monitoring module retrieves raw data, converts them into the desired metric format, and stores the processed data in the time-series database (e.g., InfluxDB [66]). In addition, this module obtains data from the database, processes the data, and provides them to the auto-scaling module. Our monitoring module also supports REST APIs implemented by Swagger to be used by other modules. Therefore, our auto-scaling module can make scaling decisions by obtaining environment data through those REST APIs.

4.2 NFV Orchestrator Module

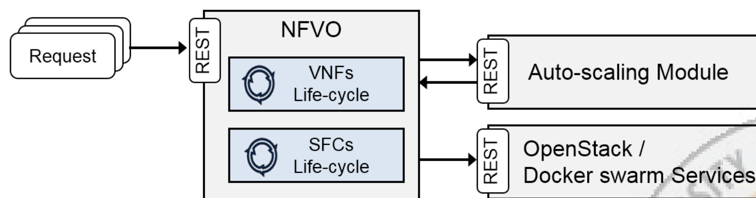
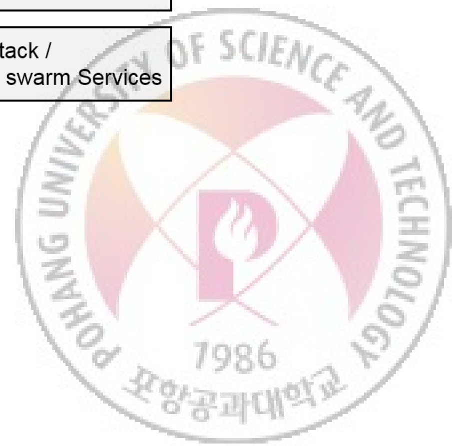


Figure 4.2: Design of NFVO module


POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

An NFV orchestrator (NFVO) module creates, removes, and updates instances and SFCs. This module retrieves requests via REST APIs implemented by Swagger and handles the lifecycle of instances and SFC in the environment. In our implementation, this module works tightly with the auto-scaling module because auto-scaler's scaling decisions are applied to the environment by the NFVO module. Our NFVO module used the OpenStack Nova API [67] to deploy instances and OpenStack SFC API [68] to manage SFCs. Moreover, the NFVO module interacts with Docker swarm to handle services consisted of containers in the environment. Figure 4.2 illustrates the design of the NFVO module.

4.3 Auto-scaling Module

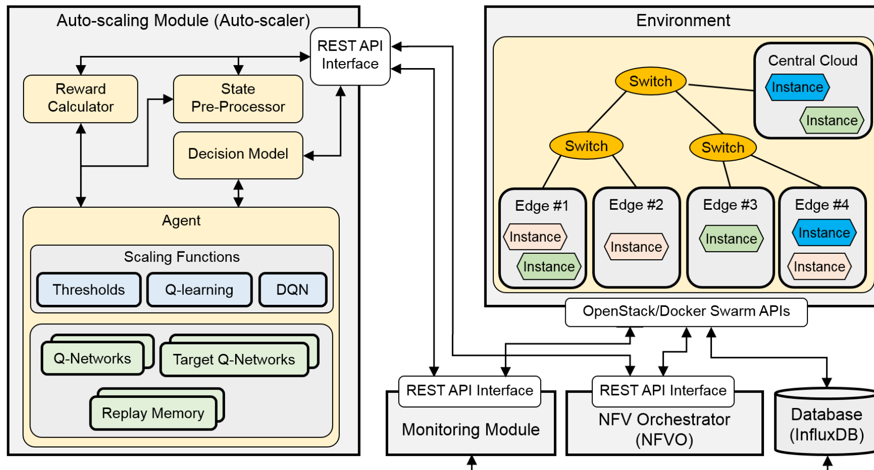
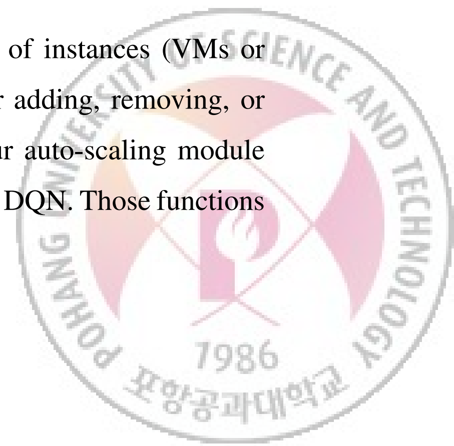


Figure 4.3: Design of auto-scaler in the form of a module

The proposed auto-scaling approach adjusts the number of instances (VMs or containers). In other words, the auto-scaler is responsible for adding, removing, or maintaining instances in response to the dynamic traffic. Our auto-scaling module provides scaling functions using thresholds, Q-learning, and the DQN. Those functions



are triggered via the REST API interface which is responsible for parsing incoming messages and sending them to sub-modules: reward calculator, state preprocessor, and decision model. Figure 4.3 illustrates the design.

The auto-scaling module receives a request through the REST API interface implemented by Swagger and executes one of the scaling functions. In this implementation, the auto-scaling module creates a thread to continue the scaling process. Therefore, our auto-scaler can run multiple scaling threads simultaneously.

When the auto-scaler runs DQN-based auto-scaling, the agent in the module creates a Q-Network, Target Q-Network, and replay memory. The auto-scaling module periodically requires the state to determine the scaling decision using the DQN. Therefore, the auto-scaler sends the request messages to the monitoring module to obtain the data used to define the state. Because a state consists of several average values and the distribution value, the state preprocessor in the auto-scaler calculates these values and generates a state in the form of a tensor consumed by the DQN. The output of the DQN is a scaling action, such as *Add*, *Maintain*, and *Remove*.

The agent provides this output to the decision model, which selects the type of instance and the location in which scaling is carried out. Finally, the auto-scaling module applies this decision to the environment and requests the monitoring module for the data used to calculate the reward value. The reward calculator in the module estimates the reward value. Subsequently, the auto-scaling module generates a tuple and inserts it into the replay memory. During the auto-scaling process, the auto-scaler identifies whether enough tuples are in replay memory. If sufficient tuples exist, more than 20 tuples in this implementation, the agent periodically updates the network parameters of the DQN by using mini-batch sampling. The auto-scaling process continues until the preset duration expires or the auto-scaler receives a request that terminates the thread. We used PyTorch [69] and the adaptive moment estimation (ADAM) optimizer [70] to implement the DQN-based auto-scaling function. Figure 4.4 shows the workflow of the auto-scaling thread and involved entities about each process.

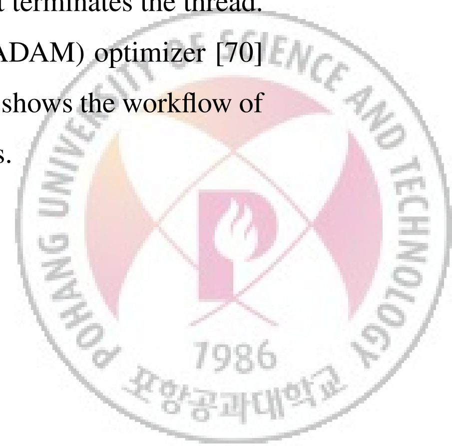


Figure 4.4: Workflow of auto-scaling

V. Evaluation

In this chapter, we describe the experimental environment used to validate our approach. We evaluated our approach compared to baseline methods. The results of the evaluation were analyzed in detail to demonstrate the effectiveness of the proposed approach.

5.1 Experimental Environment

5.1.1 Distributed cloud environment

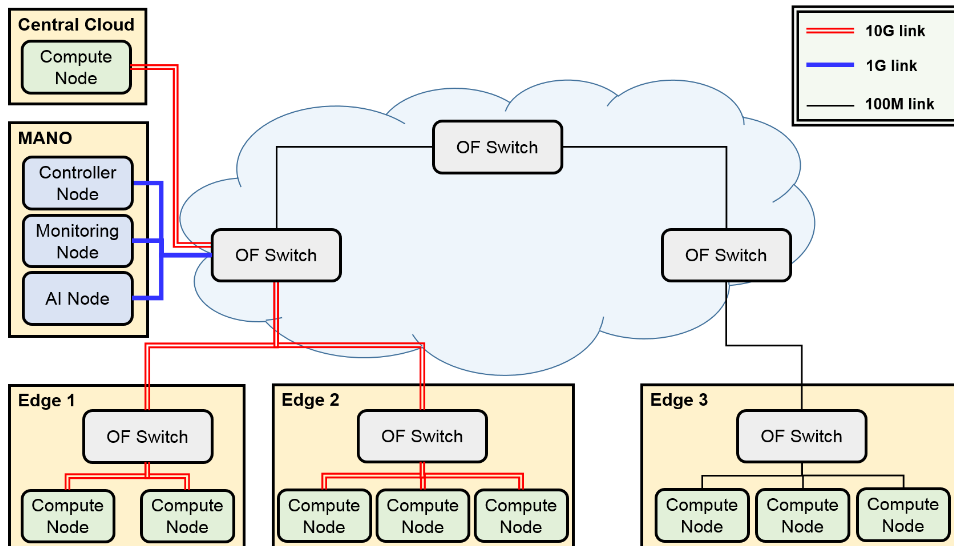
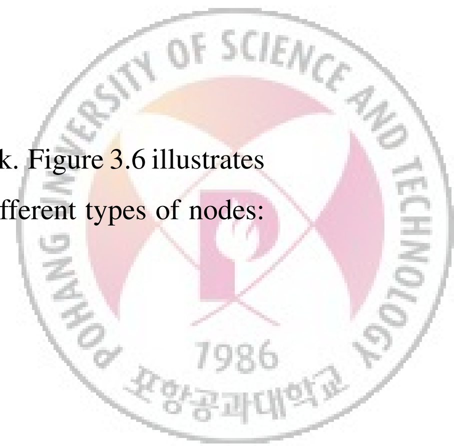


Figure 5.1: OpenStack-based Testbed

We built distributed cloud environment based on OpenStack. Figure 3.6 illustrates the OpenStack-based testbed. This environment consists of different types of nodes:


POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

controller, compute, monitoring, and AI. The controller node provides OpenStack services to operate the OpenStack infrastructure and manage the lifecycle of instances. This node creates, removes, and updates VMs and SFCs that are running in the compute nodes. Note that collector, Collectd [64], is deployed in every compute node. The monitoring node runs a monitoring module and time-series database, InfluxDB [66]. In addition, the AI node is a GPU server using NVIDIA Quadro RTX5000 to run our auto-scaling module.

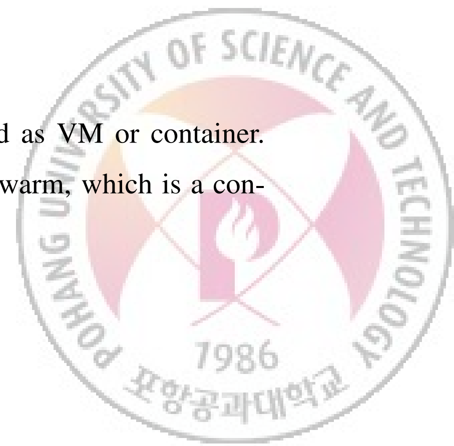
In this testbed, compute nodes are classified into an edge and central cloud to emulate distributed cloud environment. Specifically, we deployed one central cloud and three edges. Table 5.1 presents the specification of the cluster. Based on this environment, service instances are placed in one of the edges by default. However, if the edge has insufficient resources to deploy a new instance, they can be placed in neighbor edges according to the decision model. In addition, we use DEMU [71], a DPDK-based delay emulator that adds constant delays to the different types of links, to set the delay between two edges to 10 ms and between an edge and the central cloud to 17ms. It assumes that the central cloud is far from users than the edges in which the users connect; thus, the DEMU assigns a longer delay to the central cloud.

Table 5.1: Environment Specification

	Physical server(s)	**vCPU [Cores]**	**RAM [GB]**	**Disk [TB]**
Central cloud	1	32	94	5.5
Edge1	2	64	64	1.0
Edge2	3	76	68	1.7
Edge3	3	32	48	2.4
Total	**9**	**204**	**274**	**10.6**

5.1.2 Edge cloud environment

In the MEC scenario, a microservice can be instantiated as VM or container. Thus, we built an edge cloud environment based on Docker swarm, which is a con-



tainer orchestration tool that allows the operator to manage multiple containers deployed across multiple host machines. Figure 5.2 presents the environment.

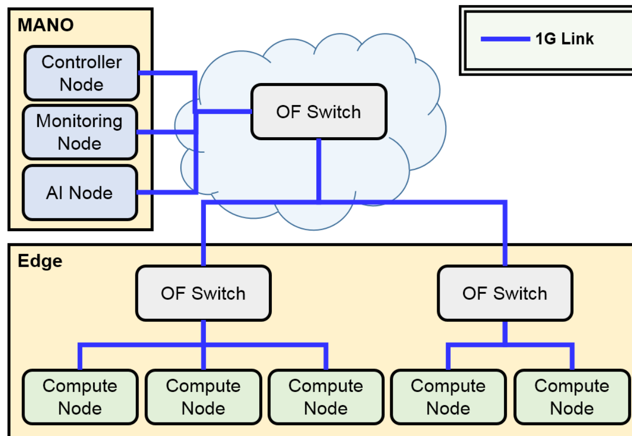


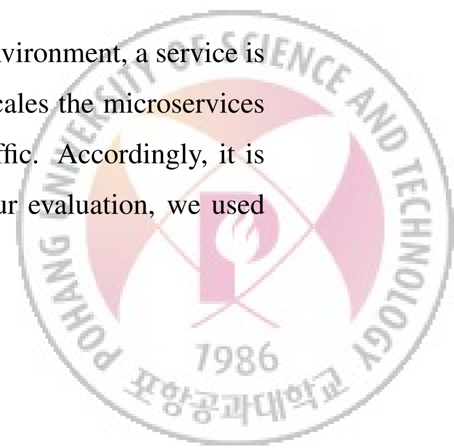
Figure 5.2: Docker Swarm-based Testbed

To operate this environment, we deployed three types of nodes: controller, monitoring, and AI. The controller node is a Docker swarm's manager node that manages the lifecycle of Docker containers. In addition, we deployed monitoring and AI nodes that were used in the OpenStack-based testbed. We assume that this environment is a single edge cloud that operates services composed of containers. This testbed consists of 5 compute nodes, each compute node has vCPU 16 cores, RAM 64 GB, and disk 500 GB.

5.1.3 Experimental Scenarios

○ Scenario setup

Based on the guideline of software design for the MEC environment, a service is a loosely coupled set of microservices. Thus, our approach scales the microservices that appear as VMs or containers in response to dynamic traffic. Accordingly, it is necessary to generate traffic that the service processes. In our evaluation, we used



open-source tools, such as RUBiS [72] and Apache-Bench [73], to generate traffic and probe packets to measure the performance of the service.

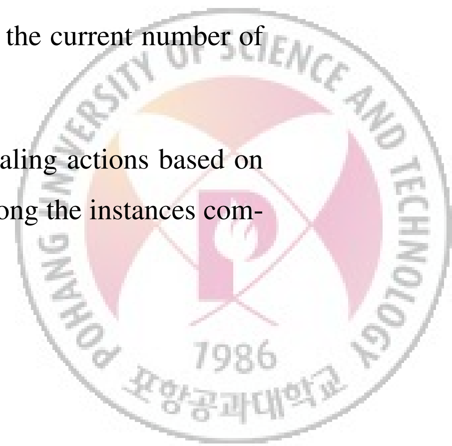
In this evaluation, we measured performance metrics, such as response time and packet loss rate, that were also used to define thresholds triggering a scaling action. However, those values can be highly variable depending on the experimental environment; therefore, the evaluation needs to set appropriate thresholds. To address this issue, we used a percentile concept to define the service-level agreement (SLA) [74]. In general, the quality of network service can be measured by the 90 and 95 percentiles of the measured values (e.g., response time, packet loss rate) [75, 76]. Thus, we used the 90 percentile as a threshold to trigger scale-out. In addition, we used the 75 percentile as a threshold to trigger scale-in.

◦ **Auto-scaling approaches**

We attempted scaling methods based on different algorithms to evaluate their effectiveness. Our auto-scaling module used one of them to scale the service. During the auto-scaling process, the auto-scaler determined the scaling decisions every 30 seconds. Moreover, the monitoring window was 10 seconds to create states if the auto-scaler used RL algorithms. To train the RL models, we repeated 950 episodes in each scenario while generating traffic. The details of these methods are described below.

- **Thresholds:** This determines scaling actions from the mean response time or packet loss rate measured by the auto-scaler. If those values are greater than the preset threshold for scale-out, the auto-scaler decides to add a new instance. Moreover, it removes one of the running instances when the values are under the scale-in threshold. Otherwise, the auto-scaler maintains the current number of instances.

- **Q-learning:** This Q-learning-based auto-scaler takes scaling actions based on states defined by the utilization of CPU and memory among the instances com-



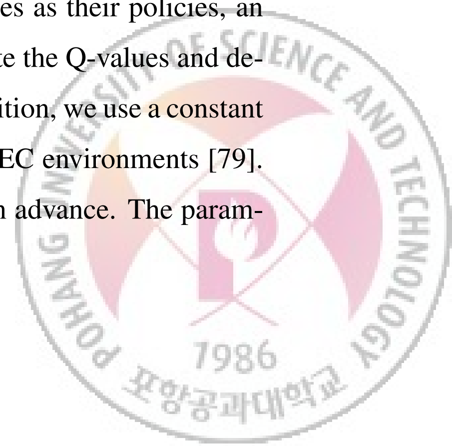
prising the service [77, 78]. These states are classified into three ranges: *high*, *normal*, and *low*. The utilization is *high* if the resource utilization is over 80%, and *low* if the utilization is less than 20%. Otherwise, it is *normally* utilized. We consider the CPU and memory, so the total number of states for Q-learning is nine. The auto-scaling using Q-learning outputs a scaling action (*Add*, *Maintain*, or *Remove*). Q-learning uses the reward model proposed in this thesis for training.

- **DQN:** It uses the DQN model proposed in our previous work [48]. It uses monitoring metrics at the level of each instance type, to define the states. Therefore, the number of data features to create a state increases according to the number of types of instances (e.g., VNF types). In addition, actions consider not only adding/removing instances but also choosing the placement. In other words, the DQN model is responsible for determining the placement without the decision model. In this evaluation, the DQN uses the same hyperparameters and reward model proposed in this thesis.

- **Proposed (DQN with Decision Model):** This method uses the DQN and decision models proposed in this thesis. It receives the service-level metrics as a state and outputs the scaling actions (*Add*, *Maintain*, or *Remove*). After determining the scaling action, the decision model selects an instance or placement at which to perform the action.

◦ Hyper Parameter Tuning

Our proposed auto-scaler provides auto-scaling functions based on thresholds, Q-learning, and DQN. Because Q-learning and DQN use Q-values as their policies, an agent is placed in the auto-scaling module to periodically update the Q-values and determine the scaling actions using the ϵ-greedy algorithm. In addition, we use a constant ϵ-value to continually update a policy for a long time in the MEC environments [79]. This process requires hyperparameters of the RL to be tuned in advance. The param-



eters are listed in Table 5.2. We empirically derived the values for these parameters after numeroustest runs. The results with varying values of these parameters are out of scope of this thesis.

Table 5.2: Hyperparameters

Approach	Parameter	Value(s)
Q-learning, DQN, Proposed	ϵ (probability of exploration)	0.10
	η (learning rate)	0.01
	γ (discount factor)	0.98
	α, β (Weights for estimating CPU and memory scores)	0.85, 0.15
	w_1, w_2, w_3 (Weights for estimating rewards)	1.0, 1.0, 1.5
DQN, Proposed	Number of neurons in hidden layer	128
	Mini-batch sampling size	16

5.2 Performance Evaluation

5.2.1 Scenario 1: Virtualized service #1 (Firewall service)

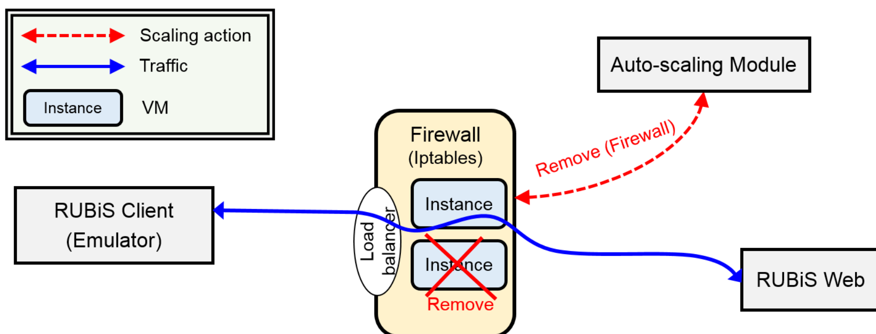
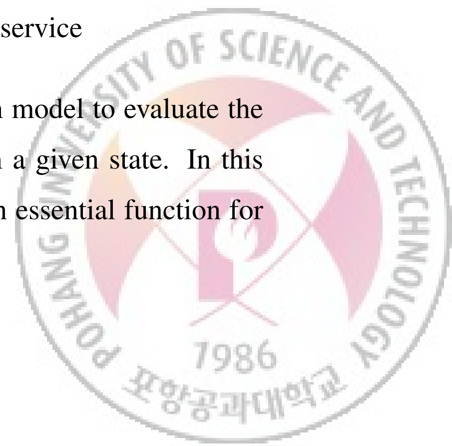


Figure 5.3: Scenario 1: auto-scaling of firewall service

First, we used auto-scaling functions without the decision model to evaluate the DQN model whether it made the right scaling decisions from a given state. In this scenario, we deployed a firewall service because it has been an essential function for

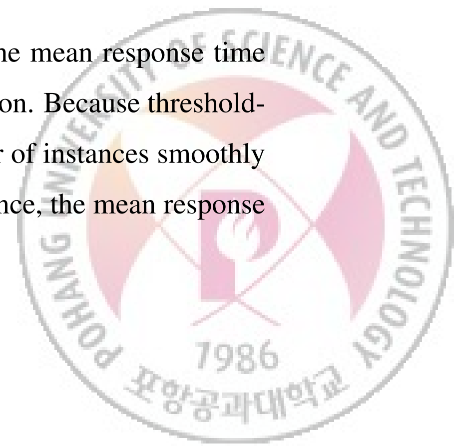


network operations, and the scaling of virtual firewalls has been the major topic of previous studies [80, 81]. Thus, we created firewall instances appeared as VMs, which were scaled by the auto-scaler, in the OpenStack-based testbed. In addition, the initial number of firewall VMs was three, and the maximum number of instances was limited to five. Figure 5.3 illustrates an auto-scaling scenario of firewall service.

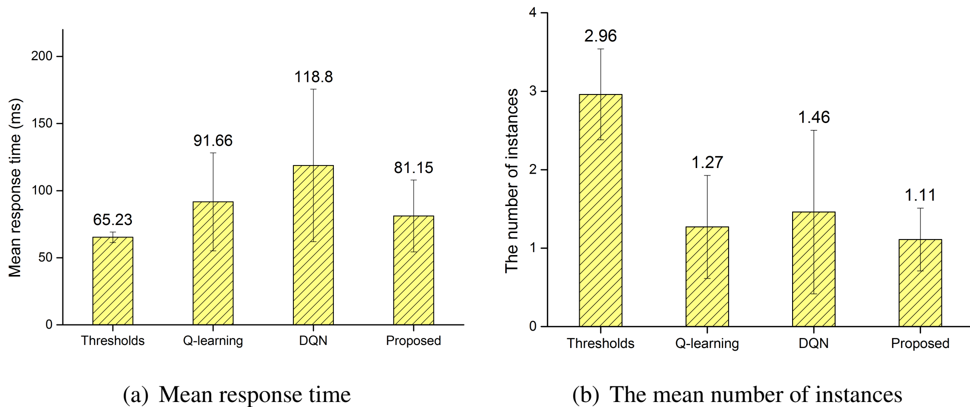
In this evaluation, each edge had a RUBiS web VM, auction web server; subsequently, a RUBiS client VM was randomly placed in one of the edges. The RUBiS client VM is an emulator that generates dynamic traffic, that is, HTTP messages, which are requests by clients for access to the auction web server, RUBiS web VM; therefore, the client VM sends the traffic to the RUBiS web VM in the same edge. After applying scaling actions, the auto-scaler measured the response time of messages passing through the firewall service. To measure the time, we utilized an Apache-Bench tool to send 150 probe packets (HTTP requests). In addition, the measured response time was used as thresholds because the response time has been one of the essential QoS metrics of network services [82]. To determine these values, we repeatedly measured the response time while generating dynamic traffic and obtained each percentile. Specifically, thresholds were 64.86 ms for scale-in and 76.72 ms for scale-out.

We validated whether the proposed approach made appropriate scaling decisions maintaining a reasonable number of instances while meeting the QoS requirements of the service. Therefore, the effectiveness of the method to determine appropriate scaling actions were measured in terms of its ability to improve the performance and reduce the operating costs. Without the decision model, the auto-scaling methods did not consider which type of instance was scaled; therefore, this scenario was similar to scaling a service designed as a monolithic architecture. Figure 5.4 shows the results of the auto-scaling method.

The auto-scaler using thresholds periodically measures the mean response time and compares it with preset thresholds to make a scaling decision. Because threshold-based auto-scaling is a reactive approach, it adjusts the number of instances smoothly to maintain the mean response time between stable ranges. Hence, the mean response



time obtained by the thresholds was the smallest among the auto-scaling methods. Nonetheless, threshold-based auto-scaling had a disadvantage in terms of operating costs, for example, over-provisioning, because it only considered the service performance (i.e., response time). Thus, the auto-scaler using thresholds maintained the mean number of instances larger than RL methods in response to dynamic traffic.

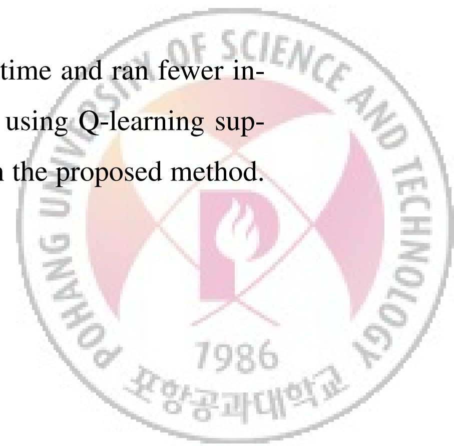


(a) Mean response time (b) The mean number of instances

Figure 5.4: Experiment results of scenario 1 (firewall)

RL-based auto-scaling approaches consider both the performance and operating costs. Because the reduction in the number of instances increased the reward, the methods maintained the number of instances less than the threshold-based method. The mean response time of the RL-based methods was longer than that of the auto-scaler using thresholds. Specifically, our previous work, **DQN**, provided the longest mean response time. The method learns the interrelation between each type of instance and uses it to determine the scaling action. However, the target service consists of a single type of instance, that is, firewall VM. Therefore, the **DQN** method was not fit the service so provided a long response time.

The proposed approach supported a reasonable response time and ran fewer instances than the other methods. Additionally, the auto-scaler using Q-learning supported a mean response time shorter than **DQN** but longer than the proposed method.


POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

The mean response time of **Q-learning** was also unstable, a larger standard deviation, than **Proposed**.

We summarized the experiment results in Table 5.3 with the case of no scaling. There were no scaling actions in response to dynamic traffic in the no scaling case, so three fixed instances only handled the traffic. Finally, this table shows how much improvement each scaling method made in terms of response time and the number of instances (i.e., operating costs to handle traffic) compared to the no scaling case.

Table 5.3: Experiment results of auto-scaling of firewall service

	Mean response time		**Mean instances**	
Approach	**Values (ms)**	**Improvement**	**Value**	**Improvement**
No scaling	121.26	-	3.00	-
Thresholds	65.23	46.21%	2.96	1.33%
Q-learning	91.66	24.41%	1.27	57.67%
DQN	118.8	2.03%	1.46	2.03%
Proposed	81.15	33.08%	1.11	63.01%

5.2.2 Scenario 2: Virtualized service #2 (Service Function Chain)

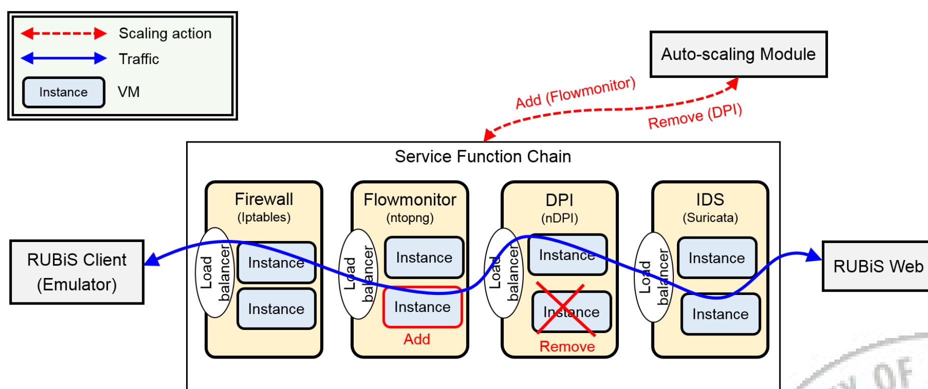
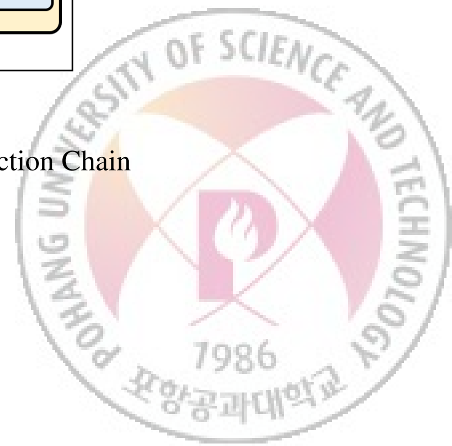


Figure 5.5: Scenario 2: auto-scaling of Service Function Chain

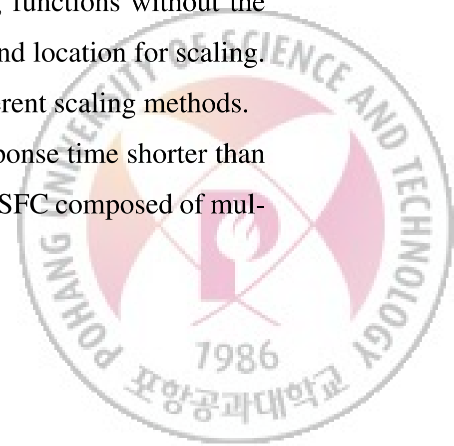


A network service in the MEC environment is provided by SFC composed of a series of VNFs [52, 83]. Although it is necessary to re-architect services with microservice, to the best of our knowledge, few open-source network services are fully designed based on the microservice architecture [52, 84]. Because of this limitation, based on the microservice concept, in which the microservice is the smallest component to perform an independent task, we design this scenario that applies auto-scaling functions to SFC. To this end, a VNF instance and SFC are considered as microservice and service respectively in this evaluation. Hence, the component to be scaled by auto-scaling is a VM running the VNF process. Figure 5.5 illustrates this scenario about auto-scaling of SFC.

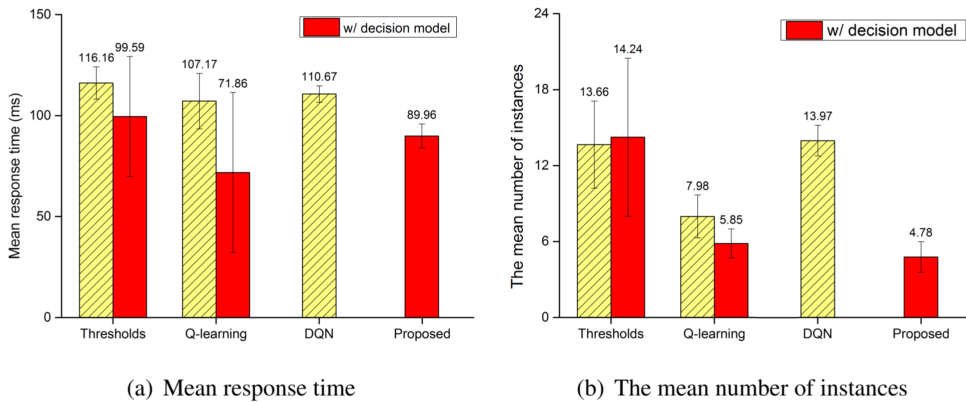
In the OpenStack-based testbed, we created the SFC composed of four types of network functions: a firewall, flow monitor, deep packet inspection, and intrusion detection system. To deploy network functions making the SFCs, we used open-source software such as iptables [85], ntopng [86], nDPI [87], and Suricata [88]. We deployed the same traffic generator, RUBiS, used in scenario 1; thus, the RUBiS web VM was placed in each edge, and the RUBiS client VM was randomly placed in one of the edges. In addition, the response time was used as thresholds, and those values were 112.28 ms for scale-in and 121.47 ms for scale-out. The initial number of each type's instances was three, and the maximum number of each type's instances was limited to five. Therefore, the SFC had twelve instances in advance, and it allowed to have 20 instances in maximum.

In this evaluation, it was necessary to select the type to be scaled to minimize the cost of scaling because the SFC had multiple VNF types. We validated whether the decision model complemented the scaling action determined by the DQN model for improved performance. In addition, we used auto-scaling functions without the decision model, so they randomly selected a type of instance and location for scaling. Figure 5.6(a) shows the mean response time measured for different scaling methods.

Although threshold-based auto-scaling kept the mean response time shorter than other methods in scenario 1, this method was ineffective for an SFC composed of mul-



tiple types of instances in this scenario. Because the auto-scaler using thresholds only used the response time as a threshold, this method could not consider the interrelation between different types of instances affecting the performance. If the auto-scaler uses additional metrics to define thresholds, it can overcome this limitation. However, it is hard to set the appropriate thresholds because this requires detailed knowledge of the service and network conditions. By contrast, RL-based auto-scaling methods provided a small mean response time without any preset conditions.

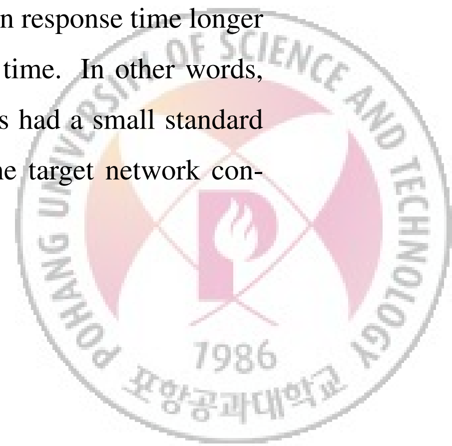


(a) Mean response time (b) The mean number of instances

Figure 5.6: Experiment results of scenario 2 (service function chain)

The Q-learning-based method with the decision model provided the shortest mean response time because the agent updated the policy sequentially from every action. Thus, this method could obtain quickly the policy to perform a scaling action that supported a short response time. Moreover, because the decision model complemented the scaling action, it could help to scale instances in the appropriate location, resulting in improved service performance (i.e., short response time).

By contrast, **DQN** and **Proposed** methods provided a mean response time longer than that of **Q-learning**, but they provided a stable response time. In other words, the response time measured in both **DQN** and **Proposed** cases had a small standard deviation because those methods used replay memory and the target network con-



cept for training. Thus, they could minimize the oscillation of the policy. Among the DQN-based methods, our previous work, **DQN**, had a smaller and more stable response time than the threshold-based method that did not use the decision model. However, because the previous **DQN** method also did not use the decision model, the mean response time was longer than that of the threshold-based method that used the decision model. In other words, there was no significant improvement compared to the threshold-based methods because the previous **DQN** method was designed to purposefully maintain a reasonable number of instances in the cloud-computing center without considering an MEC scenario. However, the proposed approach had the shortest response time among the methods except for **Q-learning** that used the decision model but had a more stable response time than **Q-learning**.

The number of instances relates to the operating costs of the service. Thus, we compared the mean number of instances maintained by different auto-scaling methods, as shown in Figure 5.6(b). Because each type had three instances in advance, there were 12 instances in the SFC before the application of auto-scaling. The threshold-based method maintained the largest number of instances because it added or removed smoothly without considering the operating costs. However, RL-based auto-scaling methods maintained a smaller number of instances than threshold-based methods because the RL models used the reward model to reduce the number of running instances. Finally, the **Proposed** maintained the smallest number of instances while showing a stable and reasonable mean response time.

We summarized the results in Table 5.4 including the no scaling case that 12 fixed instances (three instances per each type) handled dynamic traffic. This table showed how much improvement each scaling method made compared to the no scaling case.

Table 5.4: Experiment results of auto-scaling of service function chain

		Mean response time		Mean instances	
Approach	**w/Decision model**	**Values (ms)**	**Improvement**	**Value**	**Improvement**
No scaling		129.53	-	12.00	-
Thresholds		116.16	10.32%	13.66	-13.83%
	○	99.59	23.11%	14.24	-18.67%
Q-learning		107.17	17.26%	7.98	33.51%
	○	71.86	44.58%	5.85	51.25%
DQN		110.67	14.15%	13.97	-16.41%
Proposed	○	89.96	30.55%	4.78	60.17%

5.2.3 Scenario 3: Containerzed service #1 (Web service)

In the above scenarios, the services were instantiated as VMs, which were deployed in the distributed cloud environment. However, a service can also appear as containers, and all containers composing the service can be placed in the same location (e.g., single edge cloud). Therefore, we validated our approach by scaling containerized web service in the single edge, the Docker swarm-based testbed, without interacting with other edges. Figure 5.7 illustrates an auto-scaling scenario of containerized web service.

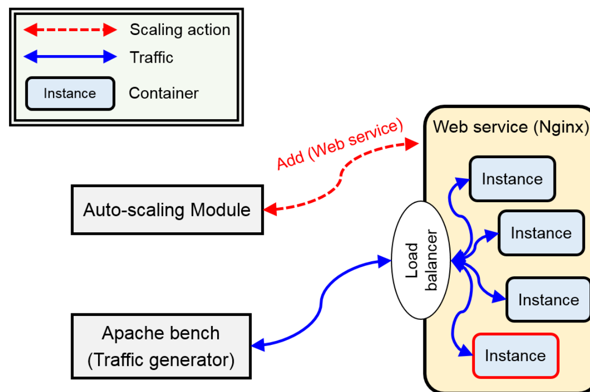
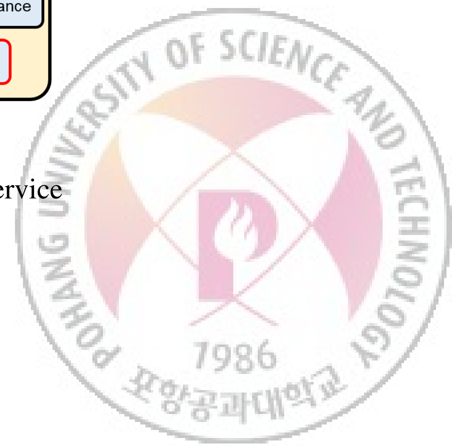
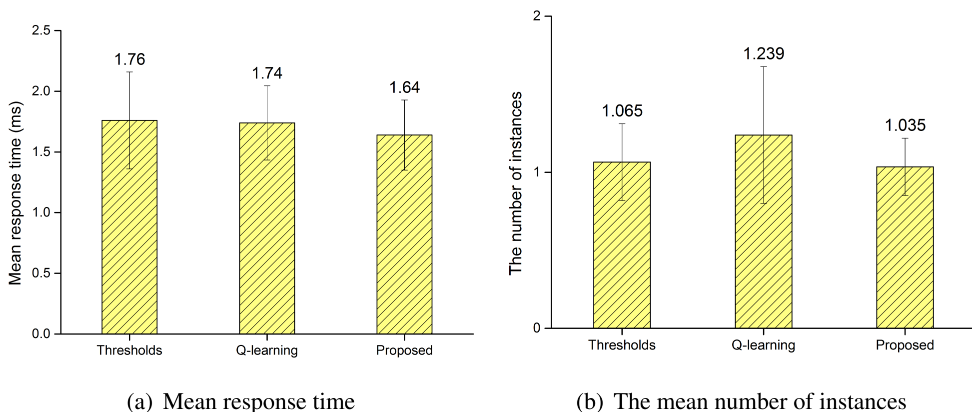


Figure 5.7: Scenario 3: auto-scaling of web service



POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

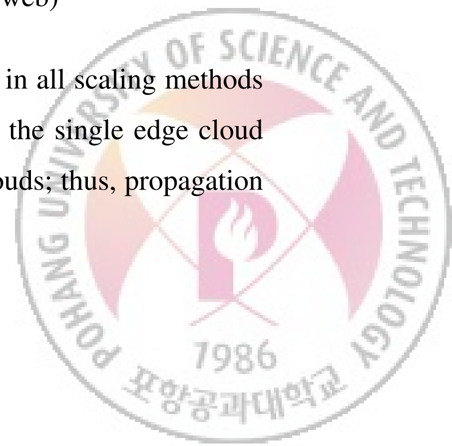
In this evaluation, we used a traffic generator, Apache-Bench, to send random HTTP requests to the web containers. In addition, we generated a dynamic traffic pattern based on the Poisson distribution, 2,000 HTTP requests on average at every interval. The web service was instantiated as containers running Nginx [89]. Nginx web server itself is a microservice, so it is widely used as a component, load-balancer or proxy, in many services. We applied three auto-scaling functions using thresholds, Q-learning, and the proposed method to the web service. Moreover, these methods worked with the decision model choosing the node to add/remove containers. Note that the DQN method addressed in our previous work did not work well in scenarios 1 and 2; thus, we did not apply it to this scenario 3. Moreover, we used the response time as thresholds, 1.837 ms for scale-in and 2.501 ms for scale-out. The initial number of web containers was three, and the maximum number of instances was limited to five. Figure 5.8 shows the mean response time and mean instances observed by different scaling methods.



(a) Mean response time (b) The mean number of instances

Figure 5.8: Experiment results of scenario 3 (web)

The result showed that the mean response time measured in all scaling methods was much smaller than the above scenarios. In this scenario, the single edge cloud processed all HTTP requests without sending them to other clouds; thus, propagation



time increasing the mean response time was small as well. In other words, the response time was mainly determined by the processing time of Nginx containers.

RL-based auto-scaling methods provided the mean response time shorter than the threshold-based method. However, Q-learning did not show significant improvement because this method would not work well for containerized service. In general, containers deployed in the same node utilized the node's entire resources while competing with other containers. In other words, the node allocated all amount of resources to each container, so few Q-learning states (e.g., *low*, *normal*) only matched in most time. Therefore, it was difficult for the Q-learning method to obtain the optimal policy to maintain the small number of instances with a small mean response time.

On the other hand, our proposed method provided the mean response time and mean instances smaller than other methods. Although the amount of resources allocated to instances (containers in this scenario) were quite different than the scenarios using VMs, this approach identified well the effect between the resource usage and the service performance.

We summarized the results in Table 5.5 including the no scaling case in which three fixed instances handled dynamic traffic. This table showed how much improvement each scaling method made compared to the no scaling case.

Table 5.5: Experiment results of auto-scaling of web service

	Mean response time		**Mean instances**	
Approach	**Value (ms)**	**Improvement**	**Value**	**Improvement**
No scaling	1.875	-	3	-
Thresholds	1.757	6.3%	1.07	64.3%
Q-learning	1.740	7.2%	1.24	58.7%
Proposed	1.636	12.8%	1.04	65.3%

5.2.4 Scenario 4: Containerzed service #2 (Video conferencing service)

In this evaluation, we deployed Jitsi Meet [90], an open-source WebRTC-based video conferencing service, in the Docker swarm-based testbed. This service is designed based on a microservice architecture, so all service functions are instantiated as independent containers. We scaled selective forwarding unit (SFU) media server requiring high bandwidth but low computational power to forward the video/audio to users [91]. Figure 5.9 illustrates an auto-scaling scenario of containerized video conferencing service.

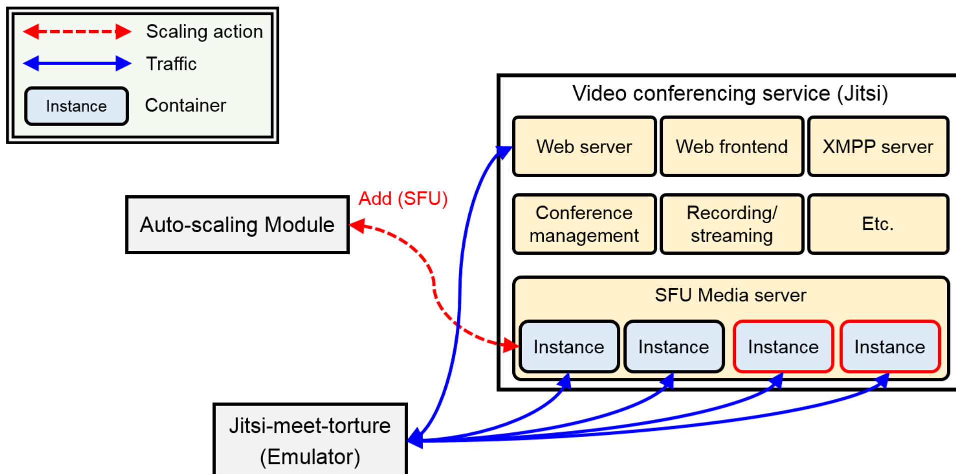
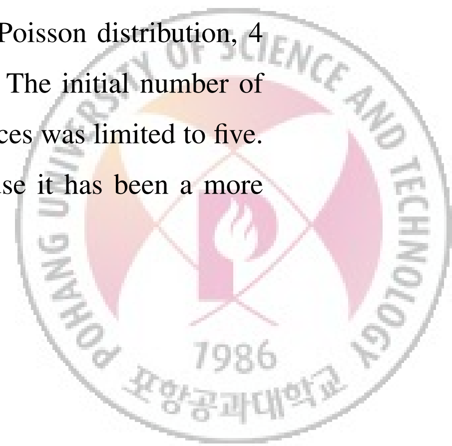


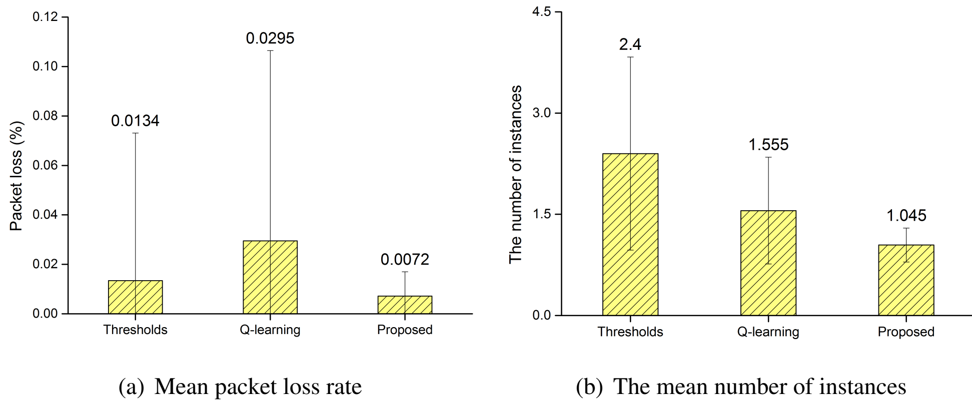
Figure 5.9: Scenario 4: auto-scaling of video conferencing service

We used Jitsi-meet-torture, an emulator to generate conferences and video/audio traffic that should be processed in SFU of Jitsi Meet. We set the emulator to generate random conferences, participants, and duration based on the Poisson distribution, 4 conferences, 20 participants, and 120 s duration on average. The initial number of SFU containers was three, and the maximum number of instances was limited to five. In addition, we used the packet loss rate as thresholds because it has been a more



POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

critical performance metric than the response time for the video conferencing service. Specifically, they were 0.0103% for scale-in and 0.0239% for scale-out. Figure 5.10 shows the mean packet loss rate and mean instances observed for different scaling methods.



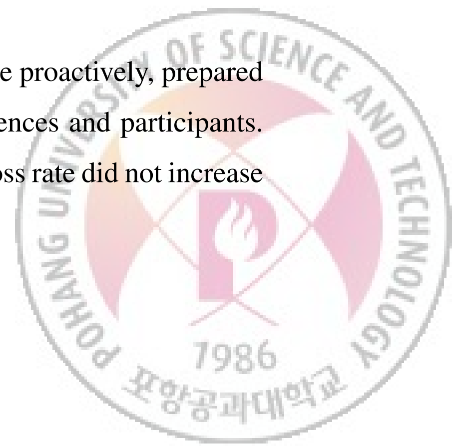
(a) Mean packet loss rate

(b) The mean number of instances

Figure 5.10: Experiment results of scenario 4 (video conferencing service)

This result showed that the threshold-based auto-scaling method maintained the mean number of instances larger than other methods while providing the mean packet loss rate smaller than Q-learning. However, threshold-based auto-scaling, a reactive method, had a major drawback. For example, although this method added SFU instances when the packet loss rate increased, it did not reduce the mean packet loss rate. The main reason was that conferences and participants were already assigned to a small number of SFU instances, so the additional SFUs would be idle, resulting in a higher cost for operating more SFUs without reducing the packet loss rate. Q-learning had a similar issue because this method considered resource utilization only to scale the service.

On the other hand, the proposed method, which could scale proactively, prepared SFU instances that were enough to handle the coming conferences and participants. When more participants for each conference came, the packet loss rate did not increase



as much compared to the one suffered by the other methods. Hence, the auto-scaler using Q-learning and thresholds suffered more losses compared to our approach even when the number of SFU instances was a lot higher.

We summarized the results in Table 5.6 including the no scaling case that three fixed instances handled dynamic traffic. This table showed how much improvement each scaling method made compared to the no scaling case.

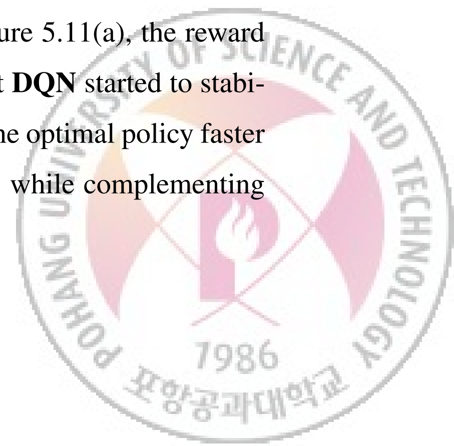
Table 5.6: Experiment results of auto-scaling of video conferencing service

	Mean packet loss		**Mean instances**	
Approach	**Value (%)**	**Improvement**	**Value**	**Improvement**
No scaling	0.027	-	3	-
Thresholds	0.013	51.85%	2.4	20%
Q-learning	0.029	-7.4%	1.56	48%
Proposed	0.007	74.07%	1.05	65%

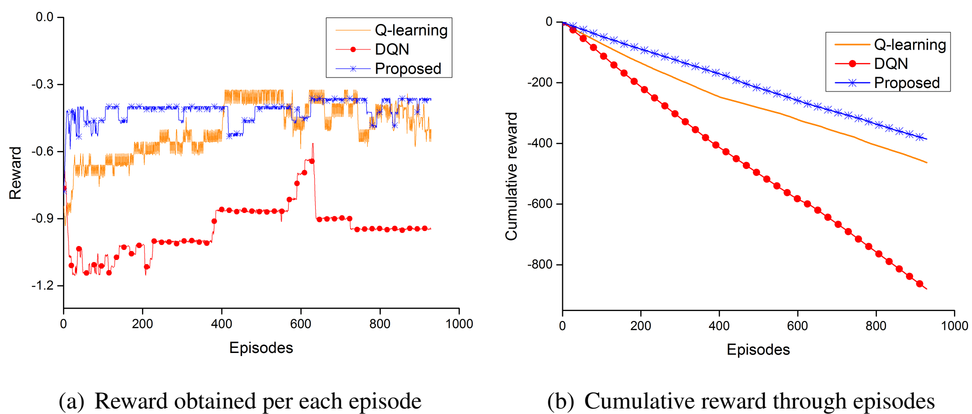
5.2.5 Performance comparison among RL models

It is necessary to obtain an optimal policy by training to use effectively RL-based auto-scaling methods. We compared the three RL methods using Q-learning with the decision model, DQN (addressed in our previous work), and the proposed approach in the distributed cloud environment while scaling SFC. Figure 5.11 shows the reward and cumulative rewards obtained during each training phase (950 episodes). Moreover, the RL methods used the reward model proposed in this thesis.

All RL-based methods experienced oscillation of rewards in the initial training phase because they were yet to have optimal policies. However, they obtained stable rewards by repeating the episodes. Specifically, compared to **DQN**, **Proposed** stabilized more quickly. According to our evaluation shown in Figure 5.11(a), the reward of **Proposed** stabilized at approximately the 109th episode, but **DQN** started to stabilize at the 226th episode. Hence, **Proposed** could converge to the optimal policy faster than **DQN** because **Proposed** simplified the states and actions while complementing



the actions by interacting with the decision model. However, Q-learning with the decision model was adversely affected by the oscillation of rewards for episodes because Q-learning learned by sequential scaling actions that were highly correlated. Despite this problem, the average rewards obtained by using Q-learning were higher than those obtained by using DQN because the decision model complemented the actions of Q-learning. **DQN** had smaller rewards than other RL methods because of the lack of consideration of the MEC scenario, but it obtained stable rewards after training.

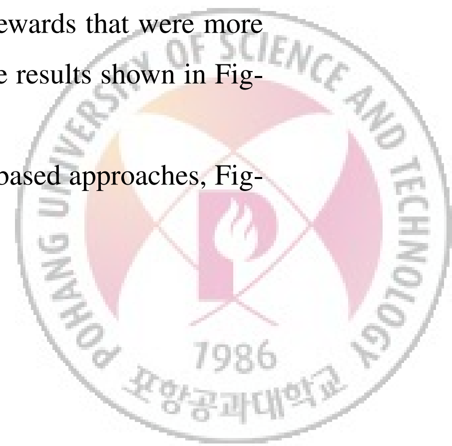


(a) Reward obtained per each episode (b) Cumulative reward through episodes

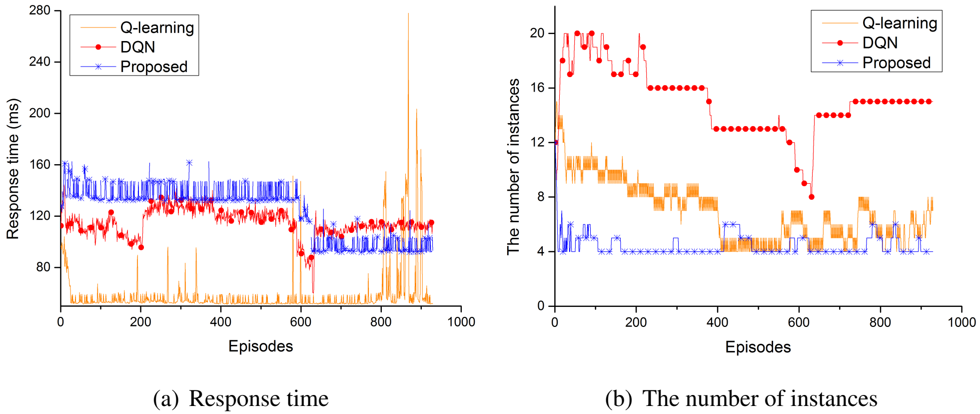
Figure 5.11: Reward of reinforcement learning (RL)-based auto-scaling approaches

According to the results, the proposed approach could achieve the highest rewards compared to other RL models. Because the reward model considers the operating costs (the number of instances) and the service performance (response time and packet loss rate), the result indicates that **Proposed** supports reasonable performance and operating costs. Although the highest reward during episodes could be diverse because the traffic was dynamically generated by the RUBiS emulator, **Proposed** obtained higher rewards than the other methods. Hence, **Proposed** provided rewards that were more stable and higher than those of other methods, according to the results shown in Figure 5.11(b).

To analyze the details of the rewards obtained by the RL-based approaches, Fig-



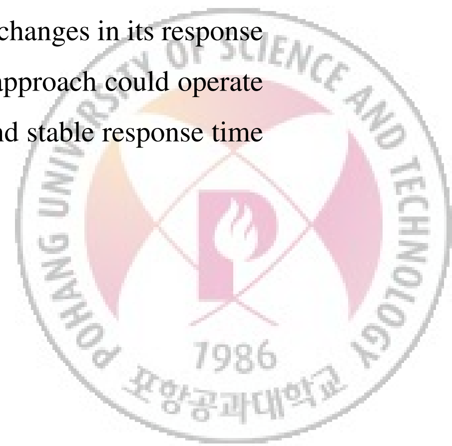
ure 5.12 presents the response time and the number of instances that affect the rewards for episodes.



(a) Response time (b) The number of instances

Figure 5.12: Response time and the number of instances during episodes

Proposed and Q-learning maintained a small number of instances during training, but **DQN** ran more instances than others. However, these DQN-based methods stabilized the number of instances to a greater extent than Q-learning. Because sequential training data used by Q-learning negatively affected the training, there was an oscillation in terms of the number of instances. Although Q-learning obtained higher rewards than **DQN** because the proposed reward model only considered the number of instances to estimate the operating costs, the operations that added or removed instances also incurred costs. Hence, the oscillation monitored in Q-learning was not appropriate for minimizing the operating costs. In addition, DQN-based methods could provide a stable response time. In other words, they operated the service efficiently in response to dynamic traffic. However, although Q-learning provided the shortest response time for episodes, it sometimes experienced significant changes in its response time. Based on the results, we summarized that the proposed approach could operate a service while supporting a reasonable number of instances and stable response time in the MEC scenario.

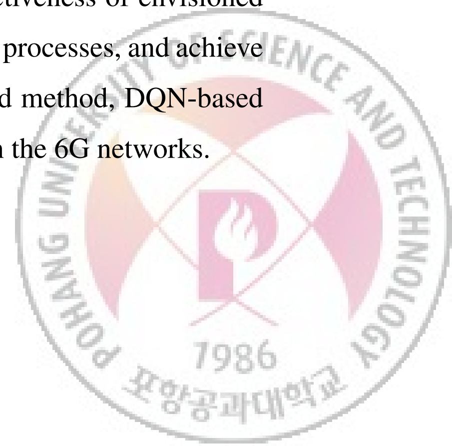


VI. Conclusion

6.1 Summary

In 5G networks with an MEC environment, it is necessary to provide services while satisfying various QoS requirements (e.g., high data rate, low latency) and minimizing operating costs of the services. In this thesis, we proposed a DQN-based auto-scaling approach that resizes service components in response to dynamic traffic. To maintain a reasonable number of service components while considering the performance and operating costs of the service, our approach consists of two models, a DQN model making a scaling decision and a decision model choosing the proper components to be scaled. We implemented an auto-scaler in the form of a module that resizes the number of instances (i.e., VMs or containers) composing the service. To validate our approach, we built two testbeds hosting VMs based on OpenStack and containers managed by Docker swarm. In the testbed, we deployed a few services and applied several auto-scaling functions to them. Compared to baseline methods, the proposed DQN-based auto-scaling method maintains a reasonable number of instances in response to dynamic traffic while providing stable response time and a low packet loss rate.

In this thesis, the proposed approach works on a MEC environment that is an essential scenario of 5G networks. However, it does not mean that our method is limited to be working on the 5G networks only. Towards the next generation, 6G network, AI will be necessary to maintain operation cost-effectiveness of envisioned complex 6G services, automate some levels of decision-making processes, and achieve a zero-touch approach. Therefore, we expect that our proposed method, DQN-based auto-scaling, will be useful to automate the service operation in the 6G networks.



6.2 Contributions

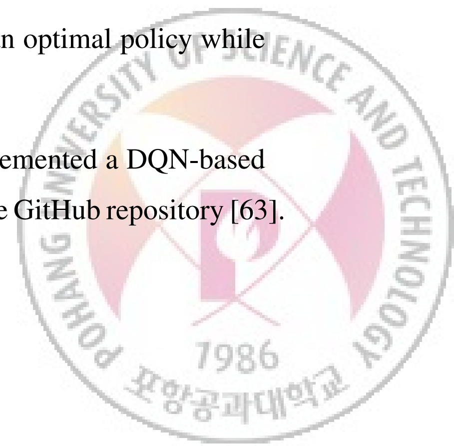
Operating a service manually and attempting to satisfy the QoS requirements is difficult because many factors need to be considered in an MEC scenario; therefore, we implemented a DQN-based auto-scaling method to automatically resize the number of instances composing the service in response to dynamic traffic. The contributions of this thesis are listed as follows.

- **DQN-based auto-scaling model for MEC environment:** We defined an auto-scaling model using DQN to apply to services in the MEC environment. In the DQN model, a state consists of performance metrics that affect the way in which services process packets and the action is to resize the number of instances assigned to the services. The reward is computed by observing the performance and operating costs of services.

- **Decision model considering service composed of multiple components:** We implemented a decision model to apply a scaling action to an appropriate component comprising a service. In the MEC environment, a service consists of multiple components (e.g. microservices). Our decision model not only chooses a component to be scaled but also adds or removes it in the appropriate location. Furthermore, our model enables a DQN agent to scale a service while considering the service performance and limited resources in the MEC environment.

- **Speed-up convergence model for auto-scaling:** We simplified the scaling actions and identified data features to define the states of the DQN model. The DQN model determines whether each state requires scaling; subsequently, the decision model complements the determined scaling actions. This enables the proposed approach to accelerate the process of finding an optimal policy while considering the MEC scenario.

- **Open-source implementation of auto-scaler:** We implemented a DQN-based auto-scaler as open-source software and published it in the GitHub repository [63].



The auto-scaler was implemented in the form of a module handling service composed of multiple containers or VMs. The module can interact with an OpenStack environment hosting VMs and container orchestration platform (e.g., docker swarm) hosting containers. The environments can be used as a virtual infrastructure manager (VIM) and for NFV management and orchestration (MANO) in a referential framework architecture defined by ETSI [26, 92].

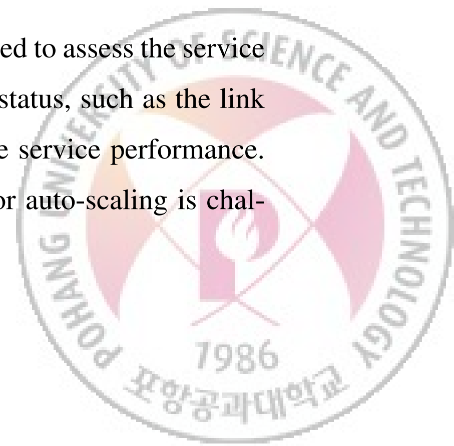
6.3 Future work

The proposed auto-scaling method using DQN effectively maintains a reasonable number of instances composing service. However, this method still has a few limitations, so they need to be solved in the future. This section describes those further issues for consideration.

6.3.1 Monitoring with low overhead

In 5G networks, a service consists of multiple components (microservices) that can be managed independently. Therefore, it is necessary to monitor fine-grained metrics at the microservice level for service operations. In general, the performance metrics can be observed by lightweight monitoring agents (e.g., Collectd) installed in either every instance running microservice or node hosting the instances. However, although each agent requires low computing power, numerous instances or nodes with the agents waste network resources inefficiently. Thus, a method for reducing the network overhead of the monitoring is needed. For example, a data aggregation method applied to a monitoring agent can reduce the data volume transmitted through the networks.

In addition, various monitoring metrics should be considered to assess the service performance and scale the service. For example, the network status, such as the link utilization, network congestion, and network delay, impact the service performance. However, obtaining appropriate network metrics to be used for auto-scaling is chal-



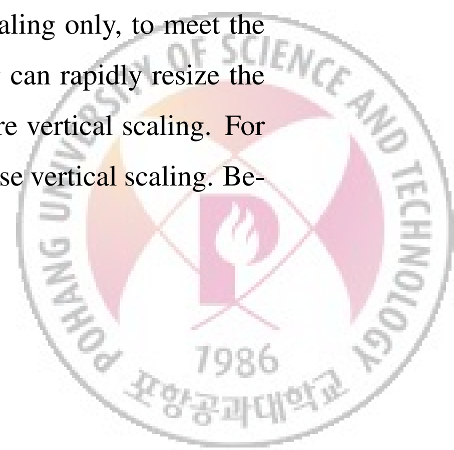
lenging because an in-depth analysis of the metrics that significantly affect the service performance is necessary. Specifically, it is required to measure the end-to-end performance of service, but it is difficult to measure exactly it in 5G networks. For example, the packets forwarded by using the microservices can have different paths, and the microservices can migrate dynamically among the edges. Moreover, it is challenging to obtain monitoring metrics with a low processing overhead. To this end, in-band network telemetry (INT) [93], which collects and reports the network status directly from the data plane, can be used. Because INT provides network visibility by obtaining fine-grained network data for analysis and measurement, it helps to obtain features to define the auto-scaling model using ML.

6.3.2 Hyper-parameter tuning

Hyperparameters, such as learning rate, batch size, and discount factor, are the properties that govern the entire training process of DQN; therefore, those values have a significant impact on the performance of the DQN-based auto-scaling model is being trained. However, the process of setting the values requires expertise and extensive trial and error. Because it is necessary to set optimal hyperparameters that enable ML/RL models to learn an optimal policy fast, we should consider finding the optimal values for the auto-scaling model. There have been methods, such as grid search, random search, and Bayesian optimization, to find optimal hyperparameters. Thus, our future work includes hyperparameter tuning for the DQN-based auto-scaling model by using the search approaches.

6.3.3 Life-cycle management of network services

In this thesis, we used auto-scaling, that is, horizontal scaling only, to meet the QoS requirements of the service. Although horizontal scaling can rapidly resize the number of instances assigned to the service, some cases require vertical scaling. For example, back-end services using relational databases tend to use vertical scaling. Be-

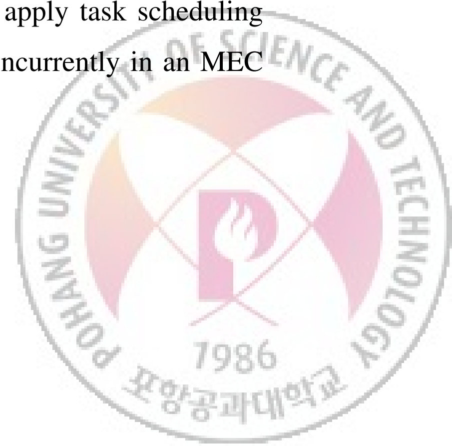


cause vertical and horizontal scaling each has advantages and disadvantages, it is necessary to address both cases by considering the service to be scaled.

It is essential to consider other functions that are used for the life-cycle management of network services. For example, in response to requests to deploy new services, we would need to assign an adequate amount of resources to the services or networks. Researchers addressed this issue in virtual network embedding and VNF deployment problems. In addition, service migration is essential for an MEC environment in which users access and migrate among edges. If the current site supporting the service has a problem or cannot meet the QoS requirements, the service can migrate to other sites. Moreover, in dynamic networks, it is necessary to orchestrate diverse functions for the life-cycle management of network services while considering the service performance, operating costs, and efficient usage of resources in the MEC environment.

6.3.4 Task scheduling with auto-scaling

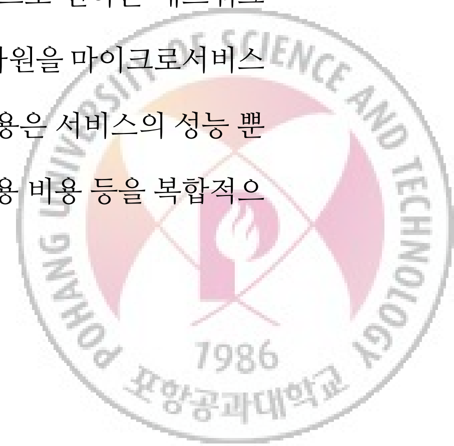
Auto-scaling can be classified into two sub-problems: scaling and task scheduling [94]. Scaling involves the adjustment of the number of instances (VMs or containers) or resizing the computing resources assigned to them. In contrast, task scheduling assigns tasks to the acquired instances. Although our approach adjusts the number of instances without considering task scheduling, scheduling should also be considered in the MEC environment. Because edges have fewer resources than the central cloud, it is necessary to consider the ways in which limited resources can be distributed efficiently to services. Task scheduling enables services to handle more user requests by using limited resources; therefore, it is essential for the MEC environment. However, task scheduling requires detailed knowledge of the service and system-level metrics to implement it. Thus, it is difficult for service operators to apply task scheduling to the service. Moreover, handling scaling and scheduling concurrently in an MEC environment is challenging owing to the many considerations.



요 약 문

수 많은 기기들이 연결되는 5G 네트워크에서는 초광대역 서비스, 고신뢰 및 초저지연 통신을 제공하는 것이 요구된다. MEC (Multi-access Edge Computing)는 이를 실현하기 위한 핵심 기술 중 하나로, 클라우드 컴퓨팅 환경을 사용자와 가까운 네트워크 엣지 (Edge), 즉, 기지국에 배치하는 방법이다. MEC는 서비스 운용을 위한 컴퓨팅 환경을 제공할 뿐만 아니라, 네트워크 종단에서 트래픽을 처리함으로써 코어 네트워크로 전달되는 트래픽을 감소시켜 네트워크 부하를 줄일 수 있다는 장점이 있다. 일반적으로 MEC 환경에서 운용되는 서비스들은 유연한 관리와 신속한 배포를 위해 마이크로서비스 (Microservice) 구조로 설계되며, 서비스를 구성하는 마이크로서비스들은 독립적인 기능을 수행하며 다른 종류의 마이크로서비스와 상호 통신한다.

MEC 환경과 마이크로서비스 구조는 서비스 요구사항에 따라 적합한 위치에 손쉽게 서비스를 배포할 수 있다는 장점이 있지만, 한편으로는 서비스 운용을 복잡하게 만드는 원인이 된다. 특히, MEC 환경은 일반적인 데이터센터 환경과는 달리 제한된 컴퓨팅 자원을 갖고 있기 때문에 주어진 자원을 효율적으로 활용하는 것이 필수이다. 또한, 마이크로서비스 구조로 설계된 서비스는 각 기능별로 모듈화된 마이크로서비스를 통해 세밀한 운용이 가능하기 때문에, 동적으로 변하는 네트워크 트래픽에 대응하여 서비스 성능을 보장하기 위한 적정량의 자원을 마이크로서비스에 할당하는 것이 필요하다. 즉, MEC 환경에서의 서비스 운용은 서비스의 성능 뿐 아니라, MEC 환경의 가용 자원, 서비스에 할당될 자원과 운용 비용 등을 복합적으



로 고려해야 한다. 하지만, 복잡한 MEC 환경에서 관리자의 판단에 따라 수작업으로 서비스를 운용하는 것은 어려운 일이다.

본 논문에서는 위와 같은 문제를 해결하기 위해 강화학습 기반의 오토 스케일링 (Auto-scaling) 방법을 제안한다. 제안하는 방법은 심층 강화학습 알고리즘인 DQN (Deep Q-network)을 활용하여 현재 운용 중인 서비스의 컴퓨팅 자원 사용량, 운용 비용, 성능 등을 고려하여 서비스에 할당되는 자원을 조절한다. 이 때, 서비스 운용을 위한 자원은 마이크로서비스로 동작하는 인스턴스를 의미하며, 제안하는 방법은 수평적 스케일링 (Horizontal scaling) 측면에서 서비스에 할당된 인스턴스의 개수를 조절한다. 또한, 본 논문의 오토 스케일링 방법은 스케일링 여부를 결정하는 DQN 모델 뿐 아니라, MEC 환경의 가용 자원, 서비스 성능 등을 고려하여 스케일링 대상이 되는 인스턴스 및 위치를 결정하는 결정 모델 (Decision model)을 포함한다. 제안된 방법은 가상 머신 (VM, Virtual Machine)과 컨테이너 (Container)들로 구성되는 서비스를 대상으로 오토 스케일링을 적용할 수 있는 모듈 형태로 구현되었다. 제안한 방법은 MEC 환경을 가정하여 구축된 실험환경에서 다양한 시나리오를 통해 성능 평가를 수행하였으며, 각 시나리오 별로 제안한 방법이 적정 개수의 인스턴스를 유지하며 안정적인 서비스 성능을 보장하는 것으로 검증되었다.

References

[1] Dajie Jiang and Guangyi Liu. An overview of 5G requirements. *5G Mobile Communications*, pages 3–26, 2017.

[2] Shunliang Zhang. An overview of network slicing for 5G. *IEEE Wireless Communications*, 26(3):111–117, 2019.

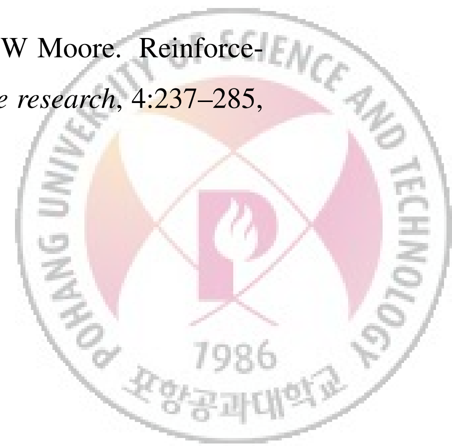
[3] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[4] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[5] Quoc-Viet Pham, Fang Fang, Vu Nguyen Ha, Md Jalil Piran, Mai Le, Long Bao Le, Won-Joo Hwang, and Zhiguo Ding. A survey of multi-access edge computing in 5g and beyond: Fundamentals, technology integration, and state-of-the-art. *IEEE Access*, 8:116974–117017, 2020.

[6] Raouf Boutaba, Mohammad A Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M Caicedo. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):16, 2018.

[7] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.



[8] Yunfa Li, Wanqing Li, and Congfeng Jiang. A survey of virtual machine system: Current technology and future trends. In *2010 Third International Symposium on Electronic Commerce and Security*, pages 332–336. IEEE, 2010.

[9] Richard Cziva, Simon Jouet, Kyle JS White, and Dimitrios P Pezaros. Container-based network function virtualization for software-defined networks. In *2015 IEEE symposium on computers and communication (ISCC)*, pages 415–420. IEEE, 2015.

[10] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[12] Dario Sabella, Vadim Sukhomlinov, Linh Trang, Sami Kekki, Pietro Paglierani, Ralf Rossbach, Xinhui Li, Yonggang Fang, Dan Druta, Fabio Giust, et al. Developing software for multi-access edge computing. *ETSI white paper*, 20:1–38, 2019.

[13] Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

[14] Justin Boyan and Michael Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in neural information processing systems*, 6:671–678, 1993.

[15] Zoubir Mammeri. Reinforcement learning based routing in networks: Review and classification of approaches. *IEEE Access*, 7:55916–55950, 2019.

[16] Hasan AA Al-Rawi, Ming Ann Ng, and Kok-Lim Alvin Yau. Application of reinforcement learning to routing in distributed wireless networks: a review. *Artificial Intelligence Review*, 43(3):381–416, 2015.

[17] Alexandru Iulian Orhean, Florin Pop, and Ioan Raicu. New scheduling approach using reinforcement learning for heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 117:292–302, 2018.

[18] Chathurangi Shyalika, Thushari Silva, and Asoka Karunananda. Reinforcement Learning in Dynamic Task Scheduling: A Review. *SN Computer Science*, 1(6):1–17, 2020.

[19] Nguyen Cong Luong, Dinh Thai Hoang, Shimin Gong, Dusit Niyato, Ping Wang, Ying-Chang Liang, and Dong In Kim. Applications of deep reinforcement learning in communications and networking: A survey. *IEEE Communications Surveys & Tutorials*, 21(4):3133–3174, 2019.

[20] Mehmet Demirci. A survey of machine learning applications for energy-efficient resource management in cloud computing environments. In *2015 IEEE 14th international conference on machine learning and applications (ICMLA)*, pages 1185–1190. IEEE, 2015.

[21] Abdul Hameed, Alireza Khoshkbarforoushha, Rajiv Ranjan, Prem Prakash Jayaraman, Joanna Kolodziej, Pavan Balaji, Sherali Zeadally, Qutaibah Marwan Malluhi, Nikos Tziritas, Abhinav Vishnu, et al. A survey and taxonomy on energy efficient resource allocation techniques for cloud computing systems. *Computing*, 98(7):751–774, 2016.

[22] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[23] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *arXiv preprint arXiv:1811.12560*, 2018.

[24] ETSI GS MEC 003 V2.2.1. Multi-access Edge Computing (MEC); Framework and Reference Architecture. *ETSI white paper*, pages 1–21, 2020.

[25] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications surveys & tutorials*, 18(1):236–262, 2015.

[26] Mehmet Ersue. ETSI NFV management and orchestration-An overview. In *Proc. of 88th IETF meeting*. ETSI, 2013.

[27] Thang Le Duc, Rafael García Leiva, Paolo Casari, and Per-Olov Östberg. Machine learning methods for reliable resource provisioning in edge-cloud computing: A survey. *ACM Computing Surveys (CSUR)*, 52(5):1–39, 2019.

[28] Stanislav Lange, Hee-Gon Kim, Se-Yeon Jeong, Heeyoul Choi, Jae-Hyung Yoo, and James Won-Ki Hong. Predicting VNF Deployment Decisions under Dynamically Changing Network Conditions. In *2019 15th International Conference on Network and Service Management (CNSM)*, pages 1–9. IEEE, 2019.

[29] DongNyeong Heo, Stanislav Lange, Hee-Gon Kim, and Heeyoul Choi. Graph Neural Network based Service Function Chaining for Automatic Network Control. In *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 7–12. IEEE, 2020.

[30] Hee-Gon Kim, Suhyun Park, Stanislav Lange, Doyoung Lee, Dongnyeong Heo, Heeyoul Choi, Jae-Hyoung Yoo, and James Won-Ki Hong. Graph neural network-based virtual network function management. In *2020 21st Asia-*

Pacific Network Operations and Management Symposium (APNOMS), pages 13–18. IEEE, 2020.

[31] Suhyun Park, Hee-Gon Kim, Jibum Hong, Stanislav Lange, Jae-Hyoung Yoo, and James Won-Ki Hong. Machine learning-based optimal vnf deployment. In *2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 67–72. IEEE, 2020.

[32] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

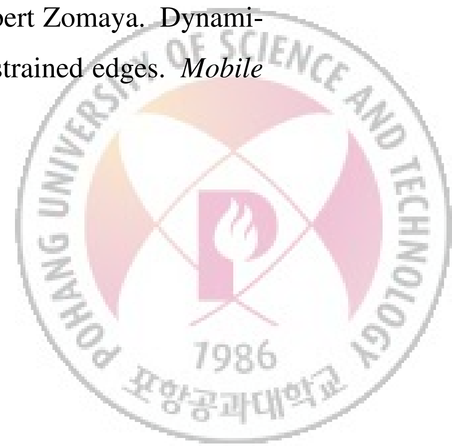
[33] Felix Yeung. Internet 2: Scaling up the backbone for R&D. *IEEE Internet Computing*, 1(2):36–37, 1997.

[34] Louiza Yala, Pantelis A Frangoudis, and Adlen Ksentini. Latency and availability driven VNF placement in a MEC-NFV environment. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2018.

[35] Lidia Ruiz, Ramón J Durán, Ignacio De Miguel, Pouria S Khodashenas, Jose-Juan Pedreno-Manresa, Noemí Merayo, Juan C Aguado, Pablo Pavon-Marino, Shuaib Siddiqui, Javier Mata, et al. A genetic algorithm for vnf provisioning in nfv-enabled cloud/mec ran architectures. *Applied Sciences*, 8(12):2614, 2018.

[36] Jingya Zhou, Jianxi Fan, Jin Wang, and Juncheng Jia. Dynamic service deployment for budget-constrained mobile edge computing. *Concurrency and Computation: Practice and Experience*, 31(24):e5436, 2019.

[37] Zhengzhe Xiang, Shuiguang Deng, Javid Taheri, and Albert Zomaya. Dynamical service deployment and replacement in resource-constrained edges. *Mobile Networks and Applications*, 25(2):674–689, 2020.



[38] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, Cornel Barna, and Gabriel Iszlai. Optimal autoscaling in a IaaS cloud. In *Proceedings of the 9th international conference on Autonomic computing*, pages 173–178, 2012.

[39] Alexandros Evangelidis, David Parker, and Rami Bahsoon. Performance modelling and verification of cloud-based auto-scaling policies. *Future Generation Computer Systems*, 87:629–638, 2018.

[40] Muhammad Wajahat, Anshul Gandhi, Alexei Karve, and Andrzej Kochut. Using machine learning for black-box autoscaling. In *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, pages 1–8. IEEE, 2016.

[41] Waheed Iqbal, Abdelkarim Erradi, Muhammad Abdullah, and Arif Mahmood. Predictive auto-scaling of multi-tier applications using performance varying cloud resources. *IEEE Transactions on Cloud Computing*, 2019.

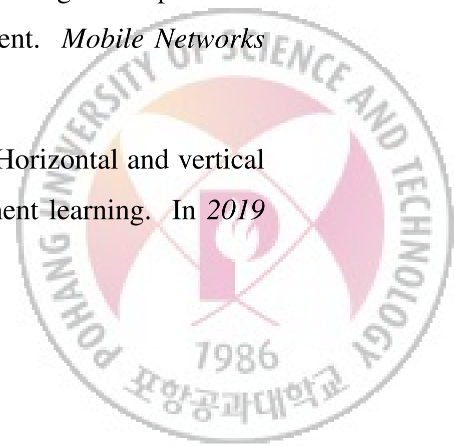
[42] Lenar Yazdanov and Christof Fetzer. Lightweight automatic resource scaling for multi-tier web applications. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 466–473. IEEE, 2014.

[43] Zhiguang Wang, Chul Gwon, Tim Oates, and Adam Iezzi. Automated cloud provisioning on aws using deep reinforcement learning. *arXiv preprint arXiv:1709.04305*, 2017.

[44] Naghmeh Dezhabad and Saeed Sharifian. Learning-based dynamic scalable load-balanced firewall as a service in network function-virtualized cloud computing environments. *The Journal of Supercomputing*, 74(7):3329–3358, 2018.

[45] JV Bibal Benifa and D Dejey. Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment. *Mobile Networks and Applications*, 24(4):1348–1363, 2019.

[46] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *2019*



IEEE 12th International Conference on Cloud Computing (CLOUD), pages 329–338. IEEE, 2019.

[47] Joy Rahman and Palden Lama. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pages 200–210. IEEE, 2019.

[48] Doyoung Lee, Jae-Hyoung Yoo, and James Won-Ki Hong. Deep Q-Networks based Auto-scaling for Service Function Chaining. In *2020 16th International Conference on Network and Service Management (CNSM)*, pages 1–9. IEEE, 2020.

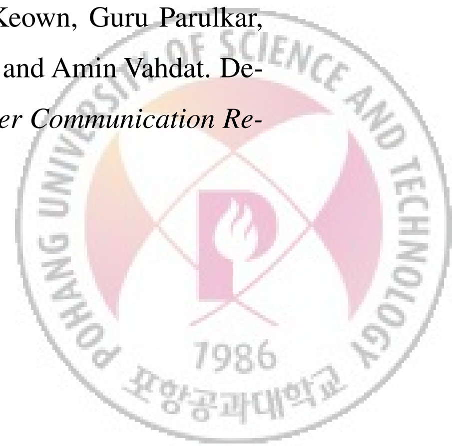
[49] Quan Yuan, Xinsheng Ji, Hongbo Tang, and Wei You. Toward Latency-Optimal Placement and Autoscaling of Monitoring Functions in MEC. *IEEE Access*, 8:41649–41658, 2020.

[50] Tejas Subramanya, Davit Harutyunyan, and Roberto Riggio. Machine learning-driven service function chain placement and scaling in MEC-enabled 5G networks. *Computer Networks*, 166:106980, 2020.

[51] László Toka, Gergely Dobreff, Balázs Fodor, and Balázs Sonkoly. Machine Learning-Based Scaling Management for Kubernetes Edge Clusters. *IEEE Transactions on Network and Service Management*, 18(1):958–972, 2021.

[52] Shihabur Rahman Chowdhury, Mohammad A Salahuddin, Noura Limam, and Raouf Boutaba. Re-architecting NFV ecosystem with microservices: State of the art and research challenges. *IEEE Network*, 33(3):168–176, 2019.

[53] Larry Peterson, Tom Anderson, Sachin Katti, Nick McKeown, Guru Parulkar, Jennifer Rexford, Mahadev Satyanarayanan, Oguz Sunay, and Amin Vahdat. Democratizing the network edge. *ACM SIGCOMM Computer Communication Review*, 49(2):31–36, 2019.



[54] Shangguang Wang, Jinliang Xu, Ning Zhang, and Yujiong Liu. A survey on service migration in mobile edge computing. *IEEE Access*, 6:23511–23528, 2018.

[55] ETSI. Multi-access Edge Computing (MEC): MEC 5G Integration. *ETSI GR MEC 031 V2.1.1*, 2020.

[56] Tao Ouyang, Zhi Zhou, and Xu Chen. Follow me at the edge: Mobility-aware dynamic service placement for mobile edge computing. *IEEE Journal on Selected Areas in Communications*, 36(10):2333–2345, 2018.

[57] Alan J Thomas, Miltos Petridis, Simon D Walters, Saeed Malekshahi Gheytassi, and Robert E Morgan. Two hidden layers are usually better than one. In *International Conference on Engineering Applications of Neural Networks*, pages 279–290. Springer, 2017.

[58] Muhammad Uzair and Noreen Jamil. Effects of Hidden Layers on the Efficiency of Neural networks. In *2020 IEEE 23rd International Multitopic Conference (INMIC)*, pages 1–6. IEEE, 2020.

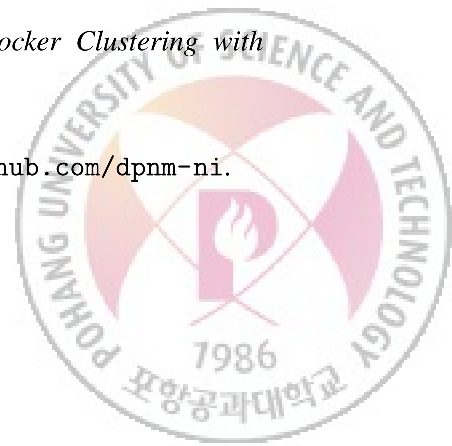
[59] Sagar Sharma. Activation functions in neural networks. *towards data science*, 6, 2017.

[60] Fabio Giust, Xavier Costa-Perez, and Alex Reznik. Multi-access edge computing: An overview of ETSI MEC ISG. *IEEE 5G Tech Focus*, 1(4):4, 2017.

[61] Zeyi Tao, Qi Xia, Zijiang Hao, Cheng Li, Lele Ma, Shanhe Yi, and Qun Li. A survey of virtual machine management in edge computing. *Proceedings of the IEEE*, 107(8):1482–1499, 2019.

[62] Fabrizio Soppelsa and Chanwit Kaewkasi. *Native Docker Clustering with Swarm*. Packt Publishing Ltd, 2016.

[63] DPNM lab. Network Intelligence Project. https://github.com/dpnm-ni.



[64] Collectd. Collectd v5.8.1. `https://github.com/collectd/collectd`.

[65] Fengtao Huang, Hao Li, Zhihao Yuan, and Xian Li. An Application Deployment Approach based on Hybrid Cloud. In *2017 ieee 3rd international conference on big data security on cloud (bigdatasecurity), ieee international conference on high performance and smart computing (hpsc), and ieee international conference on intelligent data and security (ids)*, pages 74–79. IEEE, 2017.

[66] InFluxDB. InfluxDB v1.7.9. `https://github.com/influxdata/influxdb`.

[67] OpenStack SFC project. OpenStack SFC. `https://opendev.org/openstack/networking-sfc`.

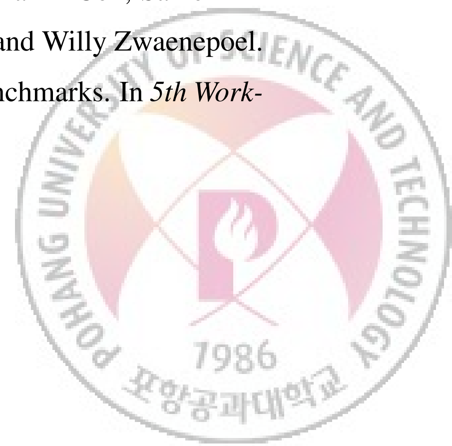
[68] OpenStack Nova project. OpenStack Nova. `https://opendev.org/openstack/nova`.

[69] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703*, 2019.

[70] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[71] Shuhei Aketa, Takahiro Hirofuchi, and Ryousei Takano. DEMU: A DPDK-based network latency emulator. In *2017 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6. IEEE, 2017.

[72] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Alan L Cox, Sameh Elnikety, Romer Gil, Julie Marguerite, Karthick Rajamani, and Willy Zwaenepoel. Specification and implementation of dynamic web site benchmarks. In *5th Workshop on Workload Characterization*, 2002.



[73] Apache HTTP server project. AB - Apache HTTP server benchmarking tool. `https://httpd.apache.org/docs/2.4/programs/ab.html`.

[74] Yasaman Amannejad, Diwakar Krishnamurthy, and Behrouz Far. Predicting web service response time percentiles. In *2016 12th International Conference on Network and Service Management (CNSM)*, pages 73–81. IEEE, 2016.

[75] Diwakar Krishnamurthy and Jerome Rolia. Predicting the QoS of an electronic commerce server: Those mean percentiles. *ACM SIGMETRICS Performance Evaluation Review*, 26(3):16–22, 1998.

[76] Mauro Andreolini, Emiliano Casalicchio, Michele Colajanni, and Marco Mambelli. A cluster-based web system providing differentiated and guaranteed services. *Cluster Computing*, 7(1):7–19, 2004.

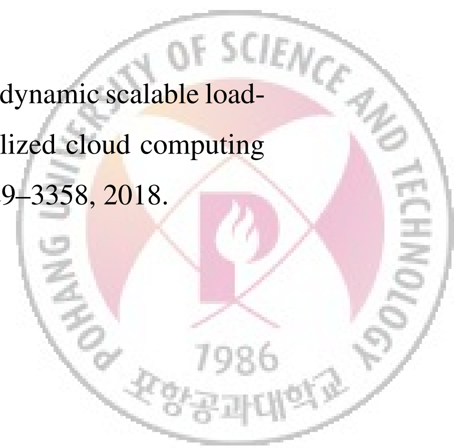
[77] Shay Horovitz and Yair Arian. Efficient cloud auto-scaling with SLA objective using Q-learning. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 85–92. IEEE, 2018.

[78] Bahar Asgari, Mostafa Ghobaei Arani, and Sam Jabbehdari. An effiecient approach for resource auto-scaling in cloud environments. *International Journal of Electrical and Computer Engineering*, 6(5):2415, 2016.

[79] Thijs Nieuwdorp. Dare to Discover: The Effect of the Exploration Strategy on an Agent's Performance. *B.S. thesis, Radboud Universiteit*, 2017.

[80] Khaled Salah, Prasad Calyam, and Raouf Boutaba. Analytical model for elastic scaling of cloud-based firewalls. *IEEE Transactions on Network and Service Management*, 14(1):136–146, 2016.

[81] Naghmeh Dezhabad and Saeed Sharifian. Learning-based dynamic scalable load-balanced firewall as a service in network function-virtualized cloud computing environments. *The Journal of Supercomputing*, 74(7):3329–3358, 2018.



[82] Kaiqi Xiong and Harry Perros. Service performance and analysis in cloud computing. In *2009 Congress on Services-I*, pages 693–700. IEEE, 2009.

[83] Boutheina Dab, Ilhem Fajjari, Mathieu Rohon, Cyril Auboin, and Arnaud Diquélou. Cloud-native service function chaining for 5G based on network service mesh. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2020.

[84] Amina Boubendir, Emmanuel Bertin, and Noémie Simoni. A VNF-as-a-service design through micro-services disassembling the IMS. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, pages 203–210. IEEE, 2017.

[85] D Coulson. Network Security Iptables, 2003.

[86] Luca Deri, Maurizio Martinelli, and Alfredo Cardigliano. Realtime high-speed network traffic monitoring using ntopng. In *28th Large Installation System Administration Conference (LISA14)*, pages 78–88, 2014.

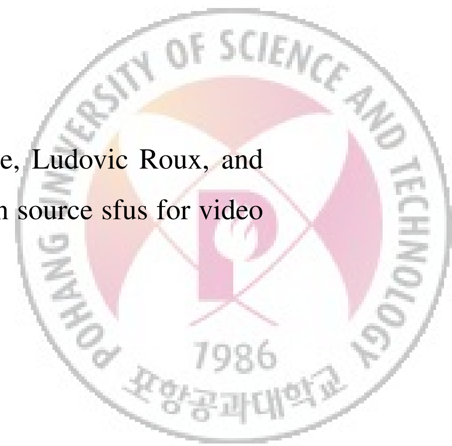
[87] Luca Deri, Maurizio Martinelli, Tomasz Bujlow, and Alfredo Cardigliano. ndpi: Open-source high-speed deep packet inspection. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 617–622. IEEE, 2014.

[88] Suricata. Suricata: Open Source IDS/IPS/NSM engine. `https://suricata-ids.org/`.

[89] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.

[90] Jitsi. Jitsi Meet. `https://jitsi.org/jitsi-meet/`.

[91] Emmanuel André, Nicolas Le Breton, Augustin Lemesle, Ludovic Roux, and Alexandre Gouaillard. Comparative study of webrtc open source sfus for video



conferencing. In *2018 Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 1–8. IEEE, 2018.

[92] Sami Kekki, Walter Featherstone, Yonggang Fang, Pekka Kuure, Alice Li, Anurag Ranjan, Debashish Purkayastha, Feng Jiangping, Danny Frydman, Gianluca Verin, et al. MEC in 5G networks. *ETSI white paper*, 28:1–28, 2018.

[93] Lizhuang Tan, Wei Su, Wei Zhang, Jianhui Lv, Zhenyi Zhang, Jingying Miao, Xiaoxi Liu, and Na Li. In-band Network Telemetry: A Survey. *Computer Networks*, 186:107763, 2021.

[94] Yisel Garí, David A. Monge, Elina Pacini, Cristian Mateos, and Carlos García Garino. Reinforcement Learning-based Application Autoscaling in the Cloud: A Survey, 2020.

Acknowledgements

긴 학위 과정 동안 지도해주신 홍원기 교수님과 유재형 교수님께 감사드립니다. 교수님들의 가르침 아래 좋은 환경에서 다양한 연구를 진행하며 성장할 수 있었습니다. 앞으로도 그러한 가르침을 잊지 않고, 매사에 성실히 임하며 발전하는 사람이 되겠습니다. 그리고, 귀한 시간 할애해주셔서 논문을 심사해주신 서영주 교수님, 황인석 교수님, 주홍택 교수님께도 감사의 말씀 올립니다.

쉽지 않은 대학원 생활동안 힘이 되어주신, 감사드리고 싶은 분들이 참 많습니다. 그 중에서도 언제나 헌신적으로 저를 지지해주시고 응원해주신 어머니, 아버지에게 깊은 감사를 드립니다. 언제나 제 편이 되어주시고 힘을 주셔서 감사합니다. 부모님의 사랑에 보답하여 자랑스러운 아들이 되고 싶었는데, 학위 과정을 마무리하며 조금이나마 부모님께 기쁨을 드릴 수 있어서 다행이라 여깁니다. 어머니, 아버지 사랑합니다 그리고 감사드립니다. 그리고, 하나 뿐인 형에게도 고맙다고 전하고 싶습니다. 같은 학교에서 수학하며 무뚝뚝한 동생을 잘 챙겨주고 격려해줘서 고맙습니다. 친동생마냥 여겨주시고 응원해주신 형수님께도 감사드립니다.

같은 연구실에서 동고동락했던 연구실 선후배님께도 감사드립니다. 좋은 사람들과 같은 공간에서 연구했던 경험은 앞으로도 잊지 못할 좋은 추억이 될 것입니다. 특히, 저와 가장 긴 시간동안 함께 지낸 동갑내기 친구 세연이와 경찬이에게 고맙다고 전하고 싶습니다. 그리고 좋은 선배이자 형으로 많은 조언을 해준 태열이형에게도 감사드립니다.

많이 흔들리는 순간도 있었던 학위 과정이었습니다. 하지만, 그 때마다 힘이 되어주신 많은 분들 덕분에 스스로를 다잡으며 마무리할 수 있었습니다. 이제 새로운 출발점에 선 지금, 제가 받은 은혜에 보답하며 다른 사람에게 선한 영향력을 주는 사람이 되겠습니다. 감사합니다.

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY
1986
포항공과대학교

Curriculum Vitae

Personal Information

Name: **Doyoung Lee**

Position: Ph.D. student

Laboratory: Distributed Processing & Network Management (DPNM) Lab.

Department: Computer Science and Engineering

Education

Degree	Year	University	Department
Ph.D.	2015-2021	POSTECH, Pohang, Korea	Computer Science and Engineering
B.S.	2009-2015	Konkuk University, Seoul, Korea	Computer Science and Engineering

Research Areas of Interest

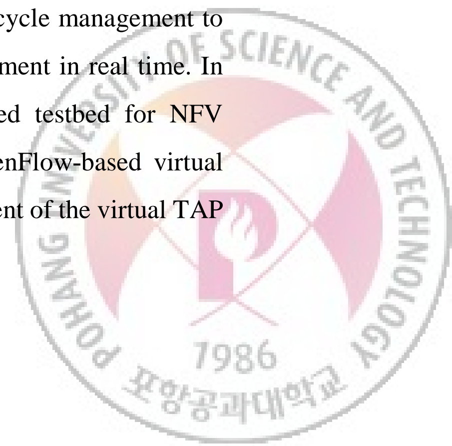
Software-Defined Networking (SDN), Network Function Virtualization (NFV), Network Virtualization, Artificial Intelligence-based Network Management

Research / Project Experiences

1. **Development of virtual network management technology based on artificial intelligence**

   Funded by Institute for Information & Communications Technology Promotion (2018 – 2021)

   This research project aims to study a virtual network management technology using artificial intelligence (AI) for NFV. This study focuses on developing AI-based algorithms, which are essential for NFV life-cycle management to monitor and control various resources of NFV environment in real time. In this project, I designed and built an OpenStack-based testbed for NFV environment. The monitoring system includes an OpenFlow-based virtual TAP managed by ONOS. I contributed to the development of the virtual TAP



and the front-end web UI/UX. Further, I studied a method of optimal service function chaining (SFC) path selection using Q-learning and auto-scaling for SFC using Deep Q-networks (DQN). I developed the algorithms as modules interacting with real OpenStack testbed.

2. **Development of Traffic Engineering based on Artificial Intelligence**

   Funded by Samsung (2020 – 2021)

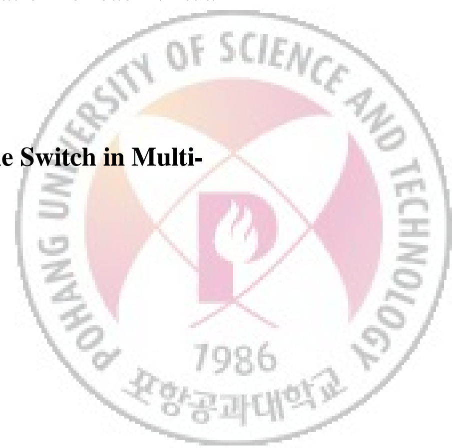
   This research project aims to develop the artificial intelligence (AI)-based fast routing algorithm using network status information monitored in the segment routing environment. This algorithm not only satisfies service requirements (e.g., throughput, latency and packet loss) but also handles network failures. In this project, I built a testbed and survey previous studies and solutions about application of reinforcement learning to segment routing.

3. **Development of Internet Infrastructure System Technologies and R&D Human Resources**

   Funded by Institute for Information & Communications Technology Promotion (2017 – 2021)

   This research project aims to develop an ONOS-based network virtualization platform for CORD environment. It also focuses on training manpower who is a professional in SDN/NFV. This work aims to develop efficient network functions based on the ONOS virtualization subsystem. In this project, I developed an OpenFlow-based virtual gateway using ONOS virtualization subsystem. The virtual gateway provides external connectivity of virtual networks without help of additional softwares such as OpenStack and Quagga. Further, I developed a virtual reactive forwarding application for each virtual network equipped with the virtual gateway.

4. **Development of Core Technologies for Programmable Switch in Multi-Service Networks**



Funded by Institute for Information & Communications Technology Promotion (2017 – 2020)

This research project aims at providing multi-service networks on programmable switch. This project includes extending P4 language and corresponding compiler, designing new programmable switch machine model and multi-service network structure and developing network monitoring, control and management technologies. In this project, I surveyed knowledge-defined networking and designed the overall knowledge plane to interwork with the INT-based network monitoring/control system.

5. **Research and Development of SDN-based Multi-protocol support network virtualization and virtual network snapshot technologies**

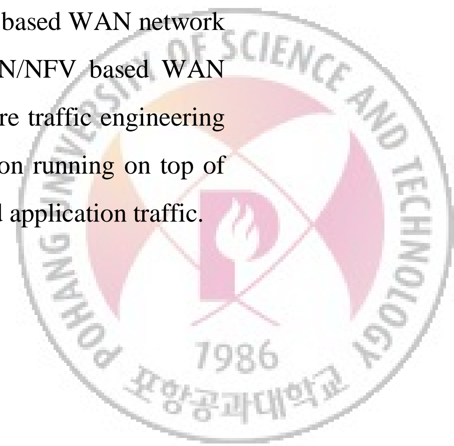
Funded by Institute for Information & Communications Technology Promotion (2017)

This research project aims to implement of SDN-based network virtualization platform. In this project, I surveyed studies to support multi-protocol by network virtualization platform. This platform purposes to support not only OpenFlow but also other protocols (e.g., P4 and NetConf/Yang) through southbound interface and northbound interface. Additionally, I participated in the development of a virtual network snapshotting application, which captures a current virtual network topology and restores it on the physical network.

6. **Korea-US Collaborative Research on SDN/NFV Security/Network Management and Testbed Build**

Funded by Institute for Information & Communications Technology Promotion (2015 – 2017)

This research project aims at studying on the SDN/NFV based WAN network stability/service management and constructing the SDN/NFV based WAN Testbed. In this project, I developed an application-aware traffic engineering system using ONOS and a traffic engineering application running on top of ONOS to allocate different bandwidth for each identified application traffic.



7. **Development of Smart Mediator for Mashup Service and Information Sharing among ICBMS Platform**

Funded by Institute for Information & Communications Technology Promotion (2015 – 2017)

This research project aims to develop a smart mediator to support development of mashup services by interconnecting ICBMS (Internet of Things, Cloud, Big data, Mobile, Security) platforms. In this project, I designed and implemented the core modules including message router, IoT adapter, big data adapter, and web dashboard for mashup service use cases. The message router supports the development of mashup services by providing useful functions such as service chaining, and routing required by developers in cooperation with various platforms according to the request messages transmitted through OpenAPIs. The IoT adapter supports interoperability with IoT platforms. The big data adapter interoperates with data analysis platforms. I worked as a lead developer for the implementation of the smart mediator including JavaScript-based web development using Node.js and AngularJS, and mashup service development using various open source projects.

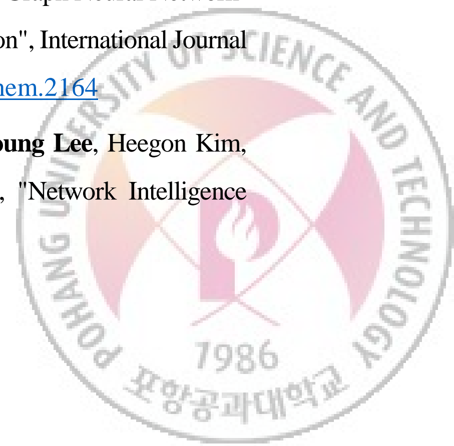
Publications

International Journal Papers

1. **Doyoung Lee**, Seyeon Jeong, Kyungchan Ko, Jae-Hyoung Yoo, James Won-Ki Hong, "Deep Q-network-based auto scaling for service in a multi-access edge computing environment", International Journal of Network Management (IJNM) (SCIE), doi: 10.1002/nem.2176

2. Heegon Kim, Suhyun Park, Stanislav Lange, **Doyoung Lee**, Dongnyeong Heo, Heeyoul Choi, Jae-Hyoung Yoo, James Won-Ki Hong, "Graph Neural Network-based Virtual Network Function Deployment Optimization", International Journal of Network Management (IJNM) (SCIE), doi: 10.1002/nem.2164

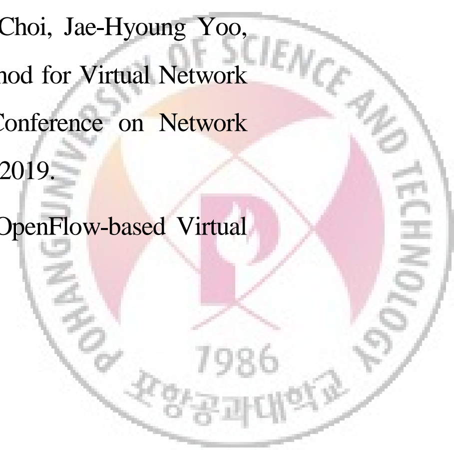
3. Stanislav Lange, Nguyen Van Tu, Seyeon Jeong, **Doyoung Lee**, Heegon Kim, Jibum Hong, Jae-Hyoung Yoo, James Won-Ki Hong, "Network Intelligence



Architecture for Efficient VNF Lifecycle Management", IEEE Transactions on Network and Service Management (TNSM) (SCIE), vol. 18, no. 2, pp. 1476-1490, June 2021, doi: 10.1109/TNSM.2020.3015244

International Conference Papers

1. Petra Gabriela, **Doyoung Lee**, Nguyen Van Tu, James Won-Ki Hong, "Machine Learning-Based Auto-Scaler for Video Conferencing Systems", 7th IEEE Conference on Network Softwarization (Netsoft 2021), Tokyo, Japan, June 28-July 2 2021 (Accepted to appear).
2. **Doyoung Lee**, Jae-Hyoung Yoo, James Won-Ki Hong, "Deep Q-Networks based Auto-scaling for Service Function Chaining", 16th International Conference on Network and Service Management (CNSM 2020), Izmir, Turkey, Nov. 2-6, 2020.
3. **Doyoung Lee**, Jae-Hyoung Yoo, James Won-Ki Hong, "Q-learning Based Service Function Chaining Using VNF Resource-aware Reward Model", 21st Asia-Pacific Network Operations and Management Symposium (APNOMS 2020), Daegu, Korea, Sep. 23-25, 2020.
4. Heegon Kim, Stanislav Lange, Suhyun Park, **Doyoung Lee**, Dongnyeong Heo, Heeyoul Choi, Jae-Hyoung Yoo, James Won-Ki Hong, "Graph Neural Network-based Virtual Network Function Management", 21st Asia-Pacific Network Operations and Management Symposium (APNOMS 2020), Daegu, Korea, Sep. 23-25, 2020 (Student Paper Award).
5. Heegon Kim, Seyeon Jeong, **Doyoung Lee**, Heeyoul Choi, Jae-Hyoung Yoo, James Won-Ki Hong, "A Deep Learning Approach to VNF Resource Prediction using Correlation between VNFs", 2nd International Workshop on Emerging Trends in Softwarized Networks (ETSN 2019), Paris, France, June 28, 2019.
6. Heegon Kim, **Doyoung Lee**, Seyeon Jeong, Heeyoul Choi, Jae-Hyoung Yoo, James Won-Ki Hong, "A Machine Learning-based Method for Virtual Network Function Resource Demand Prediction", 5th IEEE Conference on Network Softwarization (Netsoft 2019), Paris, France, June 24-28, 2019.
7. Seyeon Jeong, **Doyoung Lee**, James Won-Ki Hong, "OpenFlow-based Virtual

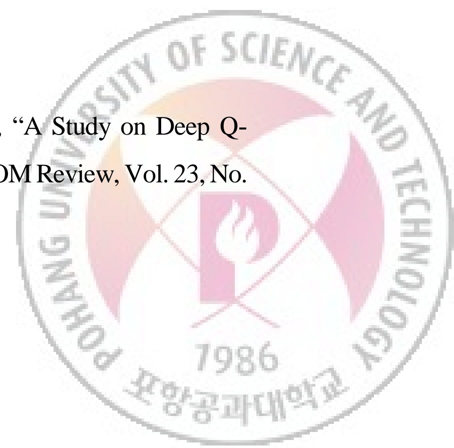


TAP using Open vSwitch and DPDK", 16th IEEE/IFIP Network Operations and Management Symposium (NOMS 2018), Taipei, Taiwan, April 23-27, 2018.

8. **Doyoung Lee**, Yoonseon Han, James Won-Ki Hong, "Design of Virtual Gateway in Virtual Software Defined Networks", 4th International Workshop on Management of SDN and NFV Systems (ManSDN/NFV 2017), Tokyo, Japan, Nov. 26, 2017.
9. **Doyoung Lee**, Seyeon Jeong, James Won-Ki Hong, "OpenAPI-based Message Router for Mashup Service Development", 19th Asia-Pacific Network Operations and Management Symposium (APNOMS 2017), Seoul, Korea, Sep. 27-29, 2017. (Best Paper Award)
10. Seyeon Jeong, **Doyoung Lee**, Jonghwan Hyun, Jian Li, James Won-Ki Hong, "Application-aware Traffic Engineering in Software-Defined Network", 19th Asia-Pacific Network Operations and Management Symposium (APNOMS 2017), Seoul, Korea, Sep. 27-29, 2017.
11. Dongho Son, **Doyoung Lee**, Taeyeol Jeong, James Won-Ki Hong, "A Hybrid Live Streaming Model for a Reliable Service", 19th Asia-Pacific Network Operations and Management Symposium (APNOMS 2017), Seoul, Korea, Sep. 27-29, 2017.
12. **Doyoung Lee**, Seyeon Jeong, Taeyeol Jeong, Jae-Hyoung Yoo, James Won-Ki Hong, "ICBMS SM: A Smart Mediator for Mashup Service Development", 18th Asia-Pacific Network Operations and Management Symposium (APNOMS 2016), Kanazawa, Japan, Oct. 5-7, 2016
13. Seyeon Jeong, **Doyoung Lee**, Junemuk Choi, Jian Li, James Won-Ki Hong, "Application-aware Traffic Management for OpenFlow Networks", 18th Asia-Pacific Network Operations and Management Symposium (APNOMS 2016), Kanazawa, Japan, Oct. 5-7, 2016

Domestic Journal Papers

1. **Doyoung Lee**, Jae-Hyoung Yoo, James Won-Ki Hong, “A Study on Deep Q-Networks based Auto-scaling in NFV Environment”, KNOM Review, Vol. 23, No. 2, Dec.2020, pp. 1-10.



2. Heegon Kim, **Doyoung Lee**, Jae-Hyoung Yoo, James Won-Ki Hong, "A Machine Learning-based Method for Virtual Network Function Resource Demand Prediction", KNOM Review, Vol. 21, No. 2, Dec.2018, pp. 1-9.
3. Jian Li, **Doyoung Lee**, Seyeon Jeong, James Won-Ki Hong, "A study on a Distributed Open Source SDN Controller – ONOS for Service Provider Network", Korean Institute of Communications and Information Sciences (KICS) Information Communications Magazine, Vol. 34, No. 12, Dec. 2017, pp. 10-19.
4. **Doyoung Lee**, Seyeon Jeong, Jonghwan Hyun, Jian Li, James Won-Ki Hong, "Application-aware Traffic Engineering in SDN", KNOM Review, Vol. 19, No. 2, Dec.2016, pp. 1-12.
5. Seyeon Jeong, **Doyoung Lee**, Jian Li, Jae-Hyoung Yoo, James Won-Ki Hong, "A Study of Control Plane Monitoring and Switch Placement Scheme in SDN Multi-controller Environment", KNOM Review, Vol. 18, No. 1, Aug.2015, pp. 11-24

Domestic Conference Papers

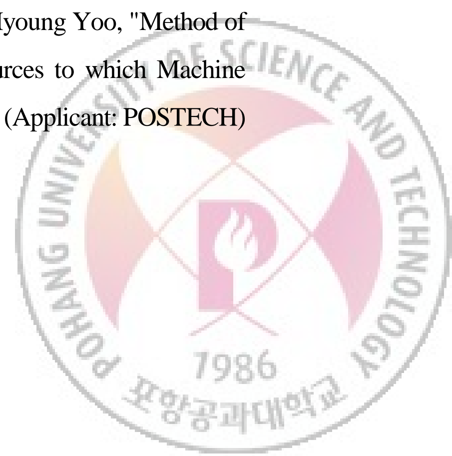
1. **Doyoung Lee**, Jae-Hyoung Yoo, James Won-Ki Hong, "A study on Deep Q-Networks-based Service Function Chaining Composition", KNOM Conference 2021, Pohang, Korea, Apr. 29-30, 2021.
2. **Doyoung Lee**, Jae-Hyoung Yoo, James Won-Ki Hong, "Q-Learning based VNF Resource-aware Service Function Chaining", KNOM Conference 2020, On-line KNOM Conference Venue, Korea, May 15, 2020 (Best Paper Award)
3. Heegon Kim, **Doyoung Lee**, Stanislav Lange, Jae-Hyoung Yoo, James Won-Ki Hong, "The VNF Management System using Machine Learning", KNOM Conference 2020, On-line KNOM Conference Venue, Korea, May 15, 2020
4. Seyeon Jeong, **Doyoung Lee**, Jae-Hyoung Yoo, James Won-Ki Hong, " Design of AI-based NFV Management System and Testbed Construction ", KNOM Conference 2019, Daegu, Korea, May 30-31, 2019, pp. 40-42. (Best Paper Award)
5. **Doyoung Lee**, Heegon Kim, Jae-Hyoung Yoo, James Won-Ki Hong, "OpenFlow-based Virtual Gateway for Virtual Networks", KNOM Workshop 2018, Seoul,

Korea, Nov. 30, 2018.

6. **Doyoung Lee**, Seyeon Jeong, Jae-Hyoung Yoo, James Won-Ki Hong, "Design and Requirements of Artificial Intelligence-based NFV Management Platform", KNOM Conference 2018, Jeju, Korea, May 10-11, 2018.
7. Seyeon Jeong, **Doyoung Lee**, Gayeon Kim, Jae-Hyoung Yoo, James Won-Ki Hong, "Design of Monitoring Framework for NFV Management based on Machine Learning", KNOM Conference 2018, Jeju, Korea, May 10-11, 2018.
8. Jibum Hong, **Doyoung Lee**, Jae-Hyoung Yoo, James Won-Ki Hong, "A Study of Machine Learning-based VNF Live Migration Technology ", KNOM Conference 2018, Jeju, Korea, May 10-11, 2018.
9. **Doyoung Lee**, Seyeon Jeong, James Won-Ki Hong, "A Study on OpenAPI based Message Router for Mashup Service Development", KNOM Conference 2017, Gwangju, Korea, June 2-3, 2017.
10. Dongho Son, **Doyoung Lee**, Taeyeol Jeong, James Won-Ki Hong, "Analysis of Multimedia Quality according to the Streamer's Network Condition in Live Streaming", KNOM Conference 2017, Gwangju, Korea, June 2-3, 2017.
11. **Doyoung Lee**, Seyeon Jeong, Junemuk Choi, James Won-Ki Hong, "A Study on Smart Mediator for Mediating ICBMS platforms and Developing Mashup Services ", KNOM Conference 2016, Chuncheon, Korea, May 12-13, 2016.
12. Seyeon Jeong, **Doyoung Lee**, Junemuk Choi, James Won-Ki Hong, "SDN Traffic Management System using DPI", KNOM Conference 2016, Chuncheon, Korea, May 12-13, 2016. (Best Paper Award)

International Patents

1. James Won-Ki Hong, Heegon Kim, **Doyoung Lee**, Jae-Hyoung Yoo, "Method of Predicting Demand of Virtual Network Function Resources to which Machine Learning is Applied", Patent No.: 16/691,505, 2019.11.21 (Applicant: POSTECH) (Pending)

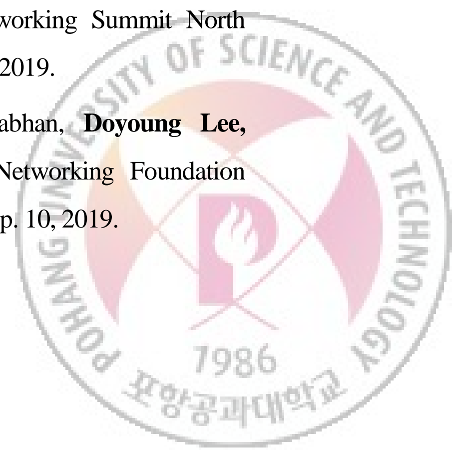


Domestic Patents

1. James Won-Ki Hong, **Doyoung Lee**, Jae-Hyoung Yoo, " Method and Apparatus for Auto Scaling", Patent No.: 10-2020-0158750, 2020.11.24 (Applicant: POSTECH) (Pending)
2. James Won-Ki Hong, **Doyoung Lee**, Jae-Hyoung Yoo, " Apparatus and Method for Selecting Service Function Chaining Route", Patent No.: 10-2020-0138914, 2020.10.26 (Applicant: POSTECH) (Pending)
3. James Won-Ki Hong, Seyeon Jeong, **Doyoung Lee**, Jae-Hyoung Yoo, "Method for Duplicating Packet and Apparatus thereof", Patent No.: 10-2019-0160735, 2019.12.05 (Applicant: POSTECH) (Pending)
4. James Won-Ki Hong, Jae-Hyoung Yoo, Heegon Kim, **Doyoung Lee**, "Method for Prediction Demand of Virtual Network Function Resource", Patent No.: 10-2019-0026890, 2019.03.08 (Applicant: POSTECH) (Pending)
5. James Won-Ki Hong, Heegon Kim, **Doyoung Lee**, Jae-Hyoung Yoo, "Method and Apparatus for Prediction Demand of Virtual Network Function Resource", Patent No.: 10-2018-0146500, 2018.11.23 (Applicant: POSTECH) (Pending)
6. James Won-Ki Hong, Seyeon Jeong, **Doyoung Lee**, "METHOD AND APPARATUS FOR IMPLEMENTING VIRTUAL TEST ACCESS POINT", Patent No.: 10-2018-0012705, 2018.02.01 (Applicant: POSTECH) (Pending)

Talks & Posters & Demos

1. **Doyoung Lee**, “Research of SDN-based Multi-protocol Support Network Virtualization”, Seoul, Korea, Sep. 20-22, 2017.
2. Pingping Lin, **Doyoung Lee**, Wei-Yu Chen, Woojoong Kim, “OMEC Project Overview & COMAC Initiative Launch”, Open Networking Summit North America (ONS NA 2019), San Jose, CA, USA, Apr. 3-5, 2019.
3. Pingping Lin, Hyunsun Moon, Badhrinath Padmanabhan, **Doyoung Lee,** Woojoong Kim, “COMAC EP Deep-Dive”, Open Networking Foundation Connect (ONF Connect 2019), Santa Clara, CA, USA, Sep. 10, 2019.



4. Hyunsun Moon, **Doyoung Lee**, “Hands-On with COMAC-In-A-Box on CloudLab”, Open Networking Foundation Connect (ONF Connect 2019), Santa Clara, CA, USA, Sep. 10, 2019.

Work Experience

Open Networking Foundation

Menlo Park, CA, USA

Research Intern

Jan. 2019 – Sep. 2019

1. Development of a monitoring system that observes Docker container-based COMAC components and visualize those status.
2. Development and testing of Docker container-based COMAC components managed by Kubernetes.

Teaching Assistantship

Title	Courses	Date
Teaching Assistant (Dept. of CSE, POSTECH)	**POSTECH MOOC: SDN-NFV**	2018 Fall
	CSED353: Computer Networks	2018 Spring
	CSED499: Senior Research Projects	2017 Spring
	CSED291: Life Planning and Management for Computer Scientists	2017, 2016 Spring
	CSED800A: Computer Science Colloquium A	2016 Spring

Awards

Title	Organization	Date
Student Paper Prize	APNOMS 2020	09/2020
Best Paper Award	KNOM Conference 2020	05/2020
Best Paper Award	KNOM Conference 2019	05/2019
Winner of Hackathon	ONK 2017	11/2017
Best Paper Award	APNOMS 2017	09/2017
Winner of Hackathon	ONOS Build 2017	09/2017
Best Paper Award	KNOM Conference 2016	05/2016

Skills

Programming Experiences	C, C++, Java, JavaScript, HTML, Python
Platforms Experiences	Docker, Kubernetes, OpenStack, ONOS, etc.

References

Prof. James Won-Ki Hong

Department of Computer Science and Engineering

Pohang University of Science and Technology, Pohang,

Korea Email: jwkhong@postech.ac.kr

Prof. Jae-Hyoung Yoo

Department of Computer Science and Engineering

Pohang University of Science and Technology, Pohang,

Korea E-mail: jhyoo78@postech.ac.kr