Doctoral Dissertation

DRAFAS: Dynamic Resource Allocation for AI-native Services

Nguyen Van Tu (응우옌 반 투)

Department of Computer Science and Engineering

Pohang University of Science and Technology

2025

AI 네이티브 서비스를 위한 동적 리소스 할당

DRAFAS: Dynamic Resource Allocation for AI-native Services

DRAFAS: Dynamic Resource Allocation for AI-native Services

by

Nguyen Van Tu
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
12. 18. 2024
Approved by
James Won-Ki Hong (Signature)
Academic advisor

DRAFAS: Dynamic Resource Allocation for AI-native Services

Nguyen Van Tu

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

12. 18. 2024

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

DCSE 응우옌 반 투 Nguyen Van Tu
20212038
DRAFAS: Dynamic Resource Allocation for AI-native Services.
AI 네이티브 서비스를 위한 동적 리소스 할당.
Department of Computer Science and Engineering ,
2025, 83p.
Advisor : James Won-Ki Hong.
Text in English.

ABSTRACT

With the recent advancements in Artificial Intelligence (AI) and Machine Learning (ML), AI-native services are becoming increasingly popular, serving every aspect of human life. The huge and growing demand for AI-native services requires more and more computing resources, especially GPU resources. Therefore, it is important to allocate resources to AI-native services efficiently to improve resource utilization and reduce infrastructure costs while still ensuring the Service Level Objectives (SLOs).

Recent studies have implemented several frameworks for dynamically allocating GPU resources to multiple ML workloads. However, they deploy all ML workloads under the same inference engine, which increases the efficiency and ease of resource allocation but reduces the isolation in terms of performance and security between ML workloads. In reality, different AI-native services can be deployed and utilized by different users or tenants; therefore, isolation is an important requirement.

This research proposes DRAFAS: a Dynamic Resource Allocation for AI-native Services framework with support for multi-tenants. To ensure isolation between services, we containerized services and managed them with a container orchestration framework. The GPU resources are spatially partitioned and allocated to each container. We then dynamically scaled the number of containers for each service based on the current load.

We developed and evaluated two resource allocation algorithms for DRAFAS: a parameter-optimized rule-based algorithm and a Deep Reinforcement Learning (DRL)-based algorithm. Traditional static or heuristic-based approaches may fall short in coping with the dynamics of network conditions and AI workloads. DRL has been shown to be efficient for resource allocation in such complex and fast-changing environments. The evaluation results showed that the DRL-based algorithm performed better in most cases when compared to the optimized rule-based algorithm and also generalized better when applied to different environment settings.

For AI-native services, while computing power is the most critical resource, network resources are also important. In the operation of an AI-native service, the compute part is in charge of processing the requests, and the network part is in charge of delivering the requests and results. Therefore, we believe that ensuring end-to-end connection quality is necessary to maintain SLO indicators. In DRAFAS, we utilized network slicing to reserve network resources for the AI-native services. The evaluation results showed that using network slicing helped maintain SLOs when there was competing traffic, especially when the AI service had high bandwidth requirements.

Contents

I. Introduction **1**

1.1 Motivation 1

1.2 Problem Statement and Research Goals 2

1.2.1 Problem Statement 2

1.2.2 Research Goals 4

1.3 Organization 5

II. Background and Related Work **7**

2.1 Background 7

2.1.1 GPU Sharing 7

2.1.2 Network Slicing 11

2.1.3 Deep Reinforcement Learning 13

2.2 Related Work 14

2.2.1 GPU resource allocation framework 14

2.2.2 DRL algorithms 16

2.2.3 Network Slicing 19

2.2.4 RL for resource scaling 20

III. Design **23**

3.1 Overall architecture and workflows 23

3.1.1 Deployment phase 25

3.1.2 Operation phase 26

3.2 DRL Algorithm 27

3.2.1 Choice of DRL algorithm 27

3.2.2 State 28

3.2.3 Action . . . 31
3.2.4 Reward . . . 33
3.3 Rule-based algorithms . . . 33
3.4 Simulator . . . 36
3.4.1 Service simulator . . . 37
3.4.2 Client simulator . . . 38

IV. Implementation **40**
4.1 DRL agent and Simulator . . . 40
4.2 End-to-end testbed . . . 41
4.3 AI-native services . . . 44
4.4 Training . . . 45
4.4.1 Dataset . . . 45
4.4.2 Service profiling . . . 47
4.4.3 Training DRL agent . . . 48
4.4.4 Parameter optimization for rule-based algorithms . . . 52

V. Evaluation **57**
5.1 Evaluation with the simulator . . . 57
5.1.1 Performance when using same setup as training . . . 58
5.1.2 Performance when changing request ratio . . . 60
5.1.3 Performance when changing the total resource units . . . 63
5.2 Evaluation on the real testbed . . . 65
5.3 Effect of network slicing . . . 67

VI. Conclusion **70**
6.1 Summary . . . 70
6.2 Future work . . . 71
6.2.1 Dynamic resource allocation for network resources . . . 71
6.2.2 Continue fine-tuning DRL agent in real environment . . . 72

6.2.3 CPU-based AI-native services 72
6.2.4 Support multiple inference request mechanisms 72
6.2.5 CPU-based AI-native services 72

References **74**

List of Tables

2.1 Comparison between GPU sharing mechanisms 11

3.1 Parameters used for modeling AI-native services in DRAFAS simulator 37

4.1 Open-sourced tools and frameworks used for DRAFAS testbed implementation 43
4.2 Statistics of request traces in train set, validation set, and test set. The unit is requests per second. 47
4.3 Computing resource allocated to one container of each service. 48
4.4 Service parameters from profiling (except SLO delay requirement, which is set manually) 49
4.5 Training and validation parameters 51
4.6 Default parameters for MPPO in Stable-Baselines3 52
4.7 Search ranges for parameters optimization for rule-based algorithm v1. Optuna allows dynamic search range. 54
4.8 Search ranges for parameters optimization for rule-based algorithm v2 55
4.9 Parameter-optimized values for rule-based v1 agent for each service type 55
4.10 Parameter-optimized values for rule-based v2 agent for each service type 55
4.11 Comparison of rule-based algorithm v1 and rule-based algorithm v2 on the validation Set 56

5.1 Detailed comparison between DRL algorithm and rule-based algorithm in simulation. The total resource unit is set to 8 and request scaling ratio set to 0.15, 0.4, 0.15 for chatbot, image classification, and text to speech clients, respectively. 60

5.2 Detailed comparison between DRL algorithm and rule-based algorithm in simulation. The total resource unit is set to 8 and request scaling ratio set to 0.3, 0.8, 0.3 for chatbot, image classification, and text to speech clients, respectively. 62
5.3 Detailed comparison between DRL algorithm and rule-based algorithm in simulation. The total resource unit is set to 128 to simulate large cluster. Request scaling ratios are set to 2, 6, 2 for chatbot, image classification, and text to speech clients, respectively, to simulate high workload. 64
5.4 Detailed comparison between DRL algorithm and rule-based algorithm in the real testbed. 66

List of Figures

1.1 AI as a service market size prediction, 2022 to 2032 [1] 1

2.1 Temporal Sharing vs. Spatial Sharing on GPUs 9
2.2 All possible MIG configurations on Nvidia A100 40GB GPU. No vertical overlap are allowed [2]. 10
2.3 Network slicing for different services [3]. 12
2.4 Overview of Deep Reinforcement Learning interacting with the environment. 13
2.5 The workflow of (a) independent PPO (IPPO) and (b) Multi-Agent PPO (MAPPO) [4]. 18

3.1 Architecture overview of DRAFAS for multiple AI-native services. . . 23
3.2 Detailed architecture and workflow of DRAFAS for an AI-native service. 25

4.1 The components used for DRAFAS testbed implementation. 41
4.2 Number of request per second over time in train set, validation set, and test set. 46
4.3 DRAFAS client and service simulator interacting with single service (Chatbot) and its corresponding DRL agent. 50
4.4 Episode mean reward of three agents during the training. 53
4.5 Training value loss of three agents during the training. 53
4.6 Validation mean reward of three agents during the training. 54

5.1 DRAFAS client and service simulator interacting with multiple services and DRL agents in evaluation. 58

5.2 Reward comparison between DRL algorithm and rule-based algorithm in simulation. The total resource unit is set to 8 and request scaling ratio set to 0.15, 0.4, 0.15 for chatbot, image classification (IC), and text to speech (TTS) clients, respectively. 59
5.3 Reward comparison between DRL algorithm and rule-based algorithm in simulation. The total resource unit is set to 8 and request scaling ratio set to 0.8, 0.3, 0.3 for chatbot, image classification (IC), and text to speech (TTS) clients, respectively. 61
5.4 Reward comparison between DRL algorithm and rule-based algorithm in simulation. The total resource unit is set to 16, 32, 64, and 128. IC stands for image classification, TTS stands for text to speech. 63
5.5 Reward comparison between DRL algorithm and rule-based algorithm in the real testbed. 66
5.6 Evaluation setup with network slicing. 67
5.7 SLO violation rate and average processing time of image classification service with and without dedicated network slice. 68
5.8 Average processing time of text to speech service with and without dedicated network slice. 69

I. Introduction

1.1 Motivation

Nowadays, AI-native services are popular and widely used in every aspect of human life across various sectors such as advertisement, healthcare, finance, transportation, manufacturing, education, communication, arts and creativity, and entertainment. Gartner predicted that the AI software market will grow to $297.9 billion with a compound annual growth rate of 19.1% [5].

The huge and growing demand for AI-native services requires increasing network and compute resources for deployment and operation. Precedence Research predicted that the market for AI as a service will reach $31.28 billion in 2027 and $155.31 billion in 2032 [1], as shown in Fig. 1.1. Next-generation communication networks (B5G and 6G networks) will have built-in support for AI-native services [6].

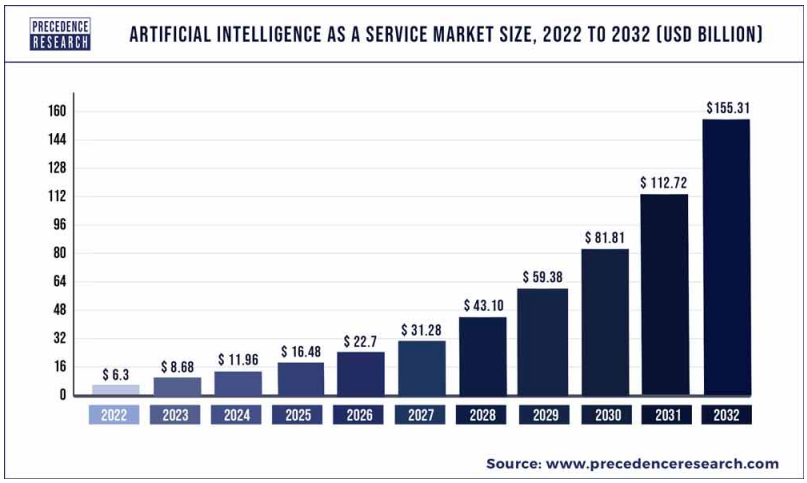


Figure 1.1: AI as a service market size prediction, 2022 to 2032 [1]

Because of the increasing demand for AI-native services, there is a need for a framework for deploying and managing AI-native services efficiently. Since the users (tenants) of each AI service can be different, the framework should natively support multi-tenants, i.e., each service should be well isolated from other services in terms of performance and security.

The framework needs to have an efficient resource allocation method for AI-native services. The resource allocation should dynamically adjust the resources allocated to each service on demand, thereby improving resource utilization, increasing the number of deployable services, and reducing operation costs while still ensuring the Service Level Objectives (SLOs). Additionally, a dynamic resource allocation framework can help accelerate the deployment of AI-native services. Furthermore, since AI has been known to consume a large amount of energy [7, 8], an efficient resource allocation solution can help reduce energy usage significantly.

1.2 Problem Statement and Research Goals

In this section, we clearly define the problem we aim to address and our final research goal.

1.2.1 Problem Statement

With the increasing demand for AI-native services, there is a need for a framework to deploy and manage AI services efficiently. The framework should provide good isolation in terms of performance and security between services, which is a key requirement for multi-tenant support. The framework should have an efficient algorithm to dynamically adjust the computing resources, especially GPU resources, allocated to each service based on demand. The target is to maximize resource utilization, increase the number of deployable services, and reduce operation costs while still maintaining SLO indicators. With the huge computational power of modern GPUs, many ML models cannot fully utilize a whole GPU, leading to a waste of computing

resources. Therefore, the framework should be able to allocate fractional GPUs. To the best of our knowledge, there is currently no open-source solution that satisfies all the above requirements.

- **Lack of available frameworks**: Currently, there is no open-source framework that satisfies all the requirements mentioned above. Previous work focused on managing multiple ML workloads by deploying them into the same inference engine, thereby overlooking the isolation requirements needed for multi-tenant use cases. While cloud-based solutions such as Microsoft Azure [9], Amazon AWS [10], and Google Cloud [11] support multi-tenants, they are not open-source and therefore cannot be applied by small to medium-sized companies that have their own GPU clusters. Additionally, these cloud-based services currently support only time-slicing (or temporal-slicing) for fractional GPU allocation, which reduces the overall throughput compared to spatial-slicing fractional GPU allocation.

- **Limitations of rule-based resource allocation algorithms**: Traditionally, static, rule-based, or heuristic-based approaches are used for such allocation tasks. However, these approaches may not adapt well to the fast-changing user requests, the dynamics of the network, or the fluctuating compute demands of AI workloads. DRL has been shown to learn hidden patterns in complex and fast-changing environments and make better control decisions compared to traditional approaches. DRL can also continuously learn and adapt to environmental changes over time.

- **The need for end-to-end SLO guarantees**: While computing resources are the most critical component of an AI-native service, network resources are also important, especially for real-time critical services such as autonomous driving, medical applications, or financial AI services. Different AI-native services may have varying requirements, such as capacity, latency, reliability, and security.

For example, a typical chat service does not require very low latency or high reliability, but a real-time video/image security service may demand low latency and high bandwidth usage. A remote surgery service may not require a huge amount of bandwidth and compute resources, but this type of service requires low latency and extreme reliability. The framework should be able to guarantee SLOs for end-to-end connections between the user and the service computing cluster for these services.

1.2.2 Research Goals

In this work, we present **DRAFAS**: **D**ynamic **R**esource **A**llocation **F**or **A**I-native **S**ervices framework that solved all three problems mentioned in Section 1.2.1. Our main contributions are listed as follows:

- **We present DRAFAS: a Resource Allocation For AI-native Services framework**. To ensure the isolation needed for multi-tenant support, we choose to deploy AI services inside containers. Each service may have multiple replicas, each has a fixed GPU resource. DRAFAS then dynamically scales the number of replicas for each service on-demand.

- **We developed two algorithms for service scaling in DRAFAS: a rule-based algorithm with optimized parameters, and a DRL-based algorithm**. DRL is expected to improve resource utilization compared to traditional rule-based or heuristic-based approaches while still ensuring the SLOs. We optimize the rule-based algorithm using the same dataset used for the DRL-based algorithm and the same reward function. For training the DRL-based algorithm and optimizing parameters for the rule-based algorithm, we developing an emulator to emulate the AI services and the clients. To make the simulator close to the real environment, we developed and deployed real AI services into GPU clusters and profiled each service to get the parameters needed for the simulation. Both algorithms are trained and evaluated using public real-world ML inference traces

from Microsoft [12].

- **We implemented and evaluated DRAFAS in both simulated and real environments**. We use Kubernetes [13] as the container orchestration framework. We use spatial slicing to create fractional GPU for allocation. Compared to temporal-slicing GPU, spatial-slicing can improve overall GPU throughput as well as provide better performance isolation. However, one challenge of GPU spatial-slicing is that there is no current mechanism to monitor how much GPU utilization for one particular GPU slice, i.e., we cannot monitor GPU utilization of each container. To overcome this challenge, we propose to measure container utilization indirectly by profiling the container beforehand and then measuring the current number of serving requests in real-time during the operation. The evaluation showed that the DRL-based algorithm performed better in most cases in both simulated and real environments, and generalized better to un-seen environment setups when compared to the rule-based algorithm.

- **We created and evaluated end-to-end AI-native services in DRAFAS by using network slicing**. We implemented network slicing using Open5GS [14] and UERANSIM [15] to create a different network slices from clients to the AI-services with bandwidth isolation. The evaluation results showed that network slicing helps maintain SLOs when there are competing traffic flows.

- We made the source code of DRAFAS available online [16] for future users and researchers.

1.3 Organization

The rest of the thesis is organized as follows. Firstly, we introduce the background and related work, which includes the technologies required for the development of DRAFAS as well as related work on compute resource allocation and network slicing, in Chapter II. Secondly, we present the detailed design of our proposed framework in

Chapter III. Thirdly, the detailed implementation of DRAFAS and the training process are presented in Chapter IV. Next, we present the evaluation results in Chapter V. Finally, we conclude this work in Chapter VI.

II. Background and Related Work

2.1 Background

In this section, we introduce the core technologies needed for our framework: GPU sharing for compute resource allocation and network slicing for network resource allocation. We also provide a brief introduction to Deep Reinforcement Learning (DRL).

2.1.1 GPU Sharing

The Need for GPU Sharing

Fine-grained GPU sharing is an important mechanism for efficient resource allocation in AI-native services. Most ML models cannot saturate modern GPUs [17, 18], leading to under-utilized GPUs if they are not shared among AI workloads. However, unlike CPU sharing, which has been available for decades, GPU sharing has only matured in recent years with the explosion of Deep Learning. Additionally, unlike CPUs, which usually have several physical cores, GPUs typically have thousands of physical cores, requiring significantly different sharing mechanisms.

We surveyed technologies enabling fine-grained compute resource allocation. There are four main mechanisms for Nvidia GPU sharing: CUDA Stream, Time Slicing, Multi-Process Service, and Multi-Instance GPU. CUDA Stream requires programming by developers, while the other three mechanisms are transparent to applications. In this section, we focus on Nvidia-based GPUs as they dominate the GPU market for deep learning and provide the most comprehensive and mature solutions for GPU sharing.

CUDA Stream

A CUDA Stream [19] is a sequence of operations that execute on the GPU in a specific order. The operations can combine computation commands and memory copies. Multiple streams can run simultaneously on different compute cores. This mechanism enables high levels of parallelism, where different models can run in parallel using different sets of CUDA streams, thus increasing GPU utilization.

One disadvantage of CUDA Stream is that it is not transparent to applications, as it must be used via programming APIs. Fortunately, most deep learning and serving frameworks, such as PyTorch or TensorFlow, already utilize CUDA Stream in their core. Another disadvantage is that the APIs can only be used within a single application, meaning all resources are shared, which limits isolation for performance and security.

Time Slicing

Time Slicing is the classic CPU sharing mechanism, which is now also available for Nvidia GPUs via compute preemption [19]. GPU resources between processes or applications are shared using a time-sharing scheduler, also referred to as temporal GPU sharing, as illustrated in Fig. 2.1.

Time Slicing is simple and does not require explicit user configuration. It is transparent to applications and does not limit the number of applications sharing the same GPU. However, it incurs a cost for context-switching between different CUDA applications. Moreover, with Time Slicing, only one application can use the GPU at a time, which may cause under-utilization since most models cannot saturate modern GPUs.

Multi-Process Service

To address the under-utilization problem of Time Slicing, Multi-Process Service (MPS) [20] allows different CUDA kernels from different processes or applications

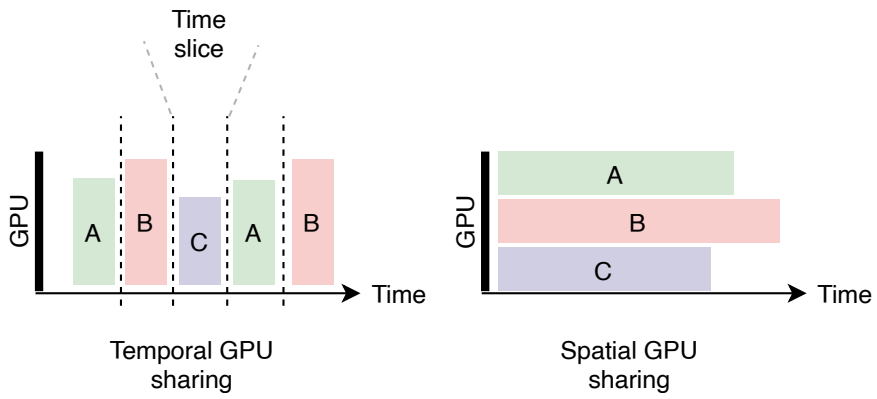


Figure 2.1: Temporal Sharing vs. Spatial Sharing on GPUs

to run concurrently on the same GPU. As the different applications run in parallel on different processing cores of the GPU, GPU sharing with MPS is considered spatial sharing, as illustrated in Fig. 2.1. While Time Slicing is managed by the system, MPS requires explicit configuration by the user. After configuration, each MPS can be treated as a single GPU and used transparently by CUDA applications. Most modern Nvidia GPUs support MPS.

Compared to Time Slicing, MPS avoids the cost of context-switching. However, GPU hardware is still shared among all MPS applications, so MPS does not provide strong isolation. Additionally, MPS only allows a maximum of 48 partitions.

Multi-Instance GPU

Multi-Instance GPU (MIG) [2] is the newest GPU sharing mechanism that provides complete hardware isolation, offering the strongest performance and security isolation. Using MIG, the GPU is physically partitioned into different GPU instances. However, due to strong physical isolation, MIG has a strict and limited set of supported configurations. For example, Fig. 2.2 shows that only four configurations are available for the Nvidia A100 GPU, and no vertical overlap configurations are allowed (e.g., a configuration with two *2g.10gb* and one *3g.20gb* is not permitted).

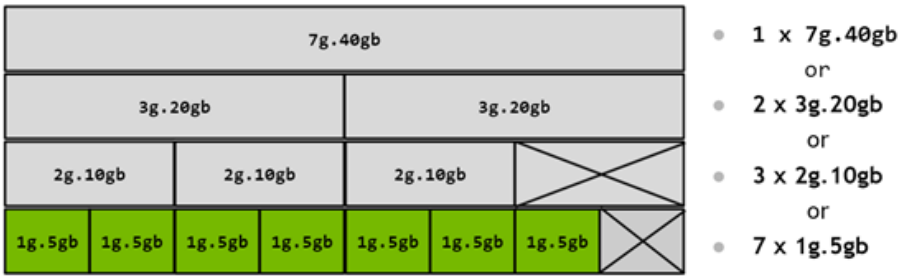


Figure 2.2: All possible MIG configurations on Nvidia A100 40GB GPU. No vertical overlap are allowed [2].

Table 2.1 summarizes and compares the above four GPU sharing mechanisms. It is important to note that the above four mechanisms can work together to provide flexible GPU sharing as needed. When the GPU is partitioned with MPS, CUDA streams and time slicing can be used within each MIG. When the GPU is partitioned with MIG, CUDA streams, time slicing, and MPS can be used within each MIG instance.

Selecting fraction GPU mechanism for DRAFAS

In our work, we choose Multi-Process Service as the mechanism for fractional GPU sharing. MPS is more flexible and available in most GPUs, unlike MIG which is not as flexible and only available in the high-end expensive ML-specific GPUs. Compared to Time-slicing, MPS provides better GPU utilization leading to higher throughput overall. With Memory protection, MPS also provides better isolation compared to time slicing. The downside of MPS is that it is not possible to measure GPU utilization for the fractional GPU allocated to the running workload or instance (e.g., container). With time-slicing, because there is only one process that can occupy the GPU at a time, there is a workaround by counting how much time a workload is scheduled to run. However, with MPS, the GPU is spatially partitioned, therefore the workaround applied to time-slicing is not working for MPS. We addressed this issue with MPS by indirectly approximating container utilization (which considers both GPU and CPU

utilization) using the number of in-flight requests that the container is currently processing. The detailed solution will be presented in Section 2.2.2.

Table 2.1: Comparison between GPU sharing mechanisms

	CUDA Streams	**Time slicing**	**MPS**	**MIG**
Partition Type	Single process	Temporal	Logical	Physical
Max Partitions	Unlimited	Unlimited	48	7
Performance Isolation	No	Percentage	Percentage	Yes
Memory Protection	No	Yes	Yes	Yes
Memory Bandwidth QoS	No	No	No	Yes
GPU Management (telemetry)	No	No	Limited Metrics	Full

2.1.2 Network Slicing

This section introduces Network Slicing, which allows end-to-end network resource allocation from end-users to compute servers, with strong isolation on performance and security. Strong isolation is required because different AI-native services have different network requirements on latency, reliability, and security, where the best-effort model of traditional network sharing mechanisms does not fit.

Network slicing creates multiple logical networks for different services on top of a shared physical network, as shown in Fig. 2.3. Each network slice uses a part of the physical infrastructure, such as the radio spectrum or network link. Each logical network is isolated from others in terms of performance, reliability, and security to serve different types of services. Network slicing is commonly discussed in the context of 5G [21] and is expected to be enhanced in 6G [6], but network slicing also applies to other types of networks in general [22]. In this work, we use network slicing within the 5G and 6G context because it contains all the necessary components for an end-to-

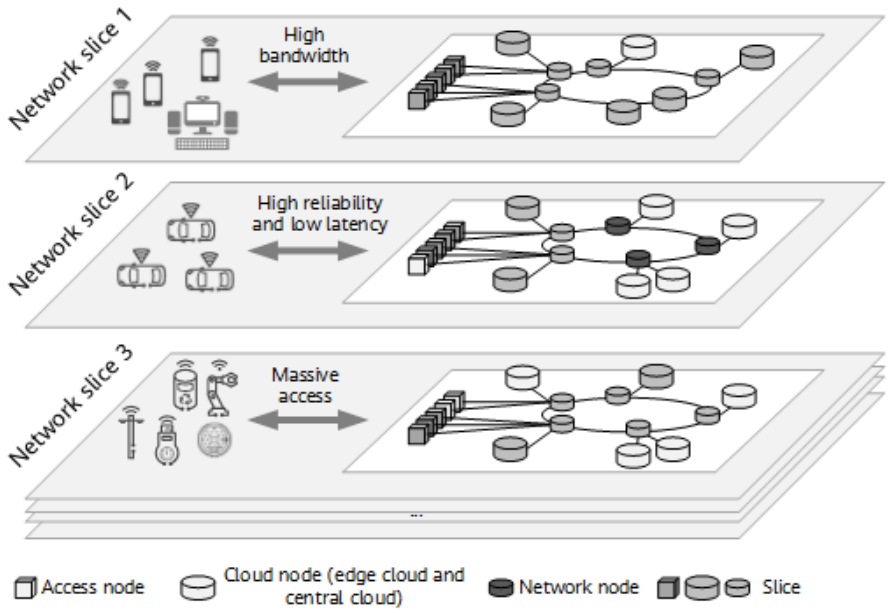


Figure 2.3: Network slicing for different services [3].

end connection: Radio Access Network (RAN), transport network, and core network. Our work can also be applied to wired networks by excluding the RAN and using the same mechanism for the transport networks.

5G defines three general categories of network services [21]: enhanced Mobile Broadband (eMBB) services, ultra-reliable and low-latency (uRLLC) services, and massive Machine-Type Communication (mMTC) services. 6G [23] will expand the service categories with two more types: long-distance and high-mobility communications (LDHMC), and extremely low-power communications (ELPC). In each category, network operators can further define different types of services with different specific QoS values. Examples of eMBB services are video on-demand, virtual reality, or augmented reality services. uRLLC services include automation, surgery, finance, or autonomous driving. mMTC is meant for low-power but massive large-scale services such as metering, sensors, and Internet of Things. All these aforementioned services

leverage AI, and some services such as autonomous driving only exist with the advancements of AI. With the broad application of AI and the need for network resource allocation for AI-native services, 5G and 6G networks are envisioned to take advantage of network slicing to provide first-class support for AI-native services [23, 6].

2.1.3 Deep Reinforcement Learning

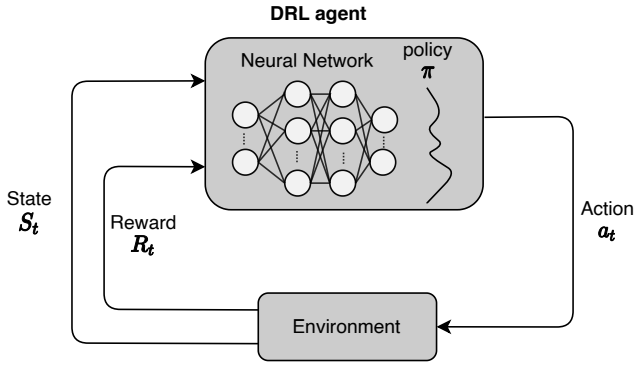


Figure 2.4: Overview of Deep Reinforcement Learning interacting with the environment.

Deep Reinforcement Learning (DRL) [24] is a combination of Deep Learning and Reinforcement Learning (RL), therefore taking advantage of both. Figure 2.4 shows the overview of a DRL agent interacting with the environment. In RL, an agent learns to make decisions automatically by interacting with the environment, making actions, and then observing the outcome of the actions. The performance of the agent is improved over time via that learning process, eventually able to produce optimal actions.

DRL uses neural networks as the core component for training and decision-making. With neural networks, DRL can handle raw data directly with high-dimensional and complex inputs, therefore freeing the developers from hand-crafting features. This ability also allows DRL to work well in environments where hand-crafted features are

hard or even impossible to define.

In this work, we use DRL as the core component for resource allocation. DRL can adapt to the dynamic and complex nature of the network and the AI workloads. The user demand, network status, and AI workloads also change over time, therefore requiring solutions that can continuously learn and adapt to the change. DRL is a suitable solution for these requirements.

2.2 Related Work

In this section, we cover related work on compute resource allocation, focusing on GPU allocation; and network resource allocation, particularly network slicing. We also cover the related work on using DRL to scale computing resources based on real-time demand.

2.2.1 GPU resource allocation framework

With the rapid development of AI and ML, compute resource allocation for AI workloads is also an important research topic. In this research, we only focus on the inference phase of AI workloads instead of the training phase, where the workload characteristics and resource requirements are totally different and should be its own research topic [18].

In [25], authors observed that ML inference workloads are becoming more dynamic and unpredictable with bursty queries. Therefore, they proposed Irina - an online scheduler that takes completion time under unpredictable workloads as the primary optimization metric. Irina utilizes batching, job stacking (i.e., running multiple jobs in parallel on one GPU), and dynamic time-slicing.

GSLICE [26] is an inference framework that virtualizes the GPU and allocates the GPU resources to different inference functions. GSLICE has self-learning and adaptive GPU resource allocation and requests batching to maximize GPU utilization while still keeping the inference latencies below the SLOs. Compared to the default MPS

configuration on GPUs, GSLICE offers a 60 to 800% improvement in GPU utilization and achieves a 2 to 13 times improvement in throughput.

SHEPHERD [27] is another inference framework for DL models. The key idea of SHEPHERD is that while individual requests can be unpredictable, the predictability is greatly improved after aggregating request streams into groups with moderately sizes. The predictability then allows the scheduler to achieve high resource utilization and high scalability. SHEPHERD also profiles models to have an optimal request batch-size for each model.

In [28], authors proposed an SLA-Driven ML inference framework for the cloud. The authors profile models to find optimal parameters for each of them. The framework also uses ML workload-specific metrics for better resource allocation decisions. The framework focuses on serverless functions on the cloud and utilizes MPS as the GPU sharing mechanism.

Nvidia Triton Inference server [29] is an open-sourced ML inference framework developed by Nvidia. The advantage of Triton is being mature and production-ready. Triton supports multiple ML frameworks such as PyTorch or TensorFlow. Triton uses CUDA streams for GPU sharing between models.

While DRL has been applied for task scheduling during the training phase [30, 31], the aforementioned works have not yet utilized DRL for resource allocation in the inference phase. However, with the increasing volume and dynamic nature of ML inference requests as well as the increasing complexity of computing infrastructure for ML inferencing, DRL can benefit the resource allocation task for AI inference workloads. Most of the aforementioned work serves platforms where multiple models run on the same software, which may not be suitable for multi-tenant environments where strong isolation is needed. In our work, we provide strong isolation by using different container sets for different workloads. Each service can run the model directly or use an inference platform on top of its container set.

2.2.2 DRL algorithms

We survey the recent work on DRL algorithms. We categorize DRL algorithms into single-agent DRL and multi-agent DRL algorithms.

Single-Agent DRL algorithms: In recent years, several deep reinforcement learning (DRL) algorithms, such as DQN and its variants [32, 33], Advantage Actor-Critic (A2C) [34], and Proximal Policy Optimization (PPO) [35], have gained wide popularity.

Proximal Policy Optimization (PPO) is a well-known policy optimization algorithm in the family of reinforcement learning algorithms developed by OpenAI. The policy optimization methods presented before PPO had several problems. Vanilla Policy Gradient (VPG) and REINFORCE [36], for example, work well only in simple environments and suffer from high variance and inefficiency. A2C [34] addressed high variance by introducing the advantage function but faced issues related to data efficiency and convergence speed. TRPO [37] improved performance and stability compared to previous algorithms but was complicated to implement. To address these problems, PPO introduced several concepts.

PPO aims to achieve TRPO's performance with a simpler implementation. TRPO restricts updates within a trust region for stable changes and solves the problem using KL divergence. PPO also seeks stable changes but does so by using a clipping-based surrogate objective function to prevent large gradient updates. The detailed explanation of PPO is presented in [35].

PPO has been demonstrated to outperform other DRL algorithms in most cases, including vanilla policy gradient, A2C, and TRPO, in the MuJoCo [38] environment. Additionally, in the Atari domain [39], PPO won against A2C and ACER [40] in many games. Despite the introduction of several new algorithms, such as SAC [41], TD3 [42], etc., PPO remains widely used because it performs well in most environments and is straightforward to implement.

Maskable PPO (MPPO) [43] is a variant of PPO that allows masking of invalid ac-

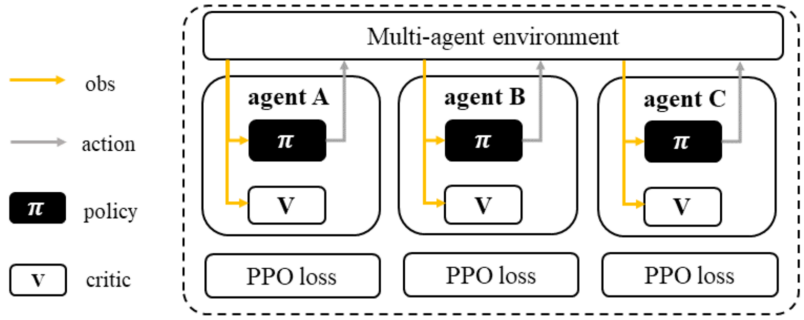
tions. During the action sampling phase, MPPO only allows valid action (not masked) to be sampled. Due to action masking, MPPO is limited to discrete action space only. However, the action mask is built into the agent and environment, effectively improving the convergence speed and performance of the algorithm.

Multi-Agent RL algorithms: In many environments, there are multiple agents working together instead of a single agent, therefore Multi-Agent Reinforcement Learning (MARL) [44, 4] is important. Most MARL algorithms are the extension of the equivalent single-agent DRL, designed to handle environments involving multiple interacting agents coexisting in a shared environment. The agents can work in independent, collaborative, or competitive mode. This introduces new challenges, such as non-stationary due to the dynamic behaviors of other agents, scalability due to large numbers of agents in some environments, and communication or coordination among agents.

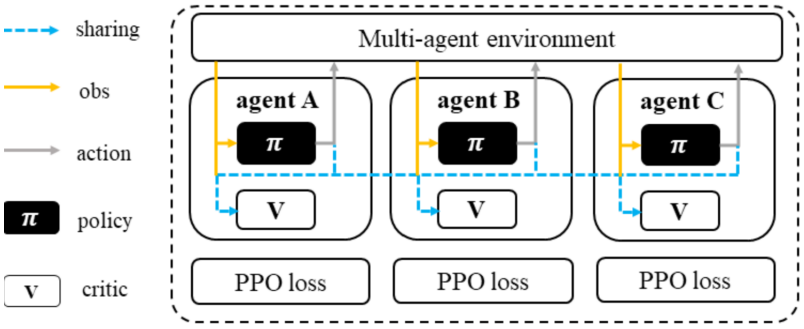
A simple method to make single-agent RL work in Multi-agent environments is to consider multiple agents as one big agent with state as the combined state of all agents, action as the combined action of all agents, and reward as the combined reward of all agents. While this method is simple to implement, it does not scale well when the number of agents increases due to the huge size of combined states and actions. Additionally, it is difficult to add or remove agents during the operation.

Another simple class of MARL is independent MARL, where we assign a single-agent RL to each agent in the environment and let each agent learn and work on its own [45, 4]. There are multiple independent RL algorithms such as IA2C - the independent MARL version of A2C, ITRPO - the independent MARL version of TRPO, and IPPO - the independent MARL version of PPO [4]. There is no difference between an independent MARL algorithm and the equivalent single-agent RL algorithm except that the agents in the independent MARL algorithm coexist in the same environments where they may or may not affect each other. The advantages of independent MARL algorithms are scalable, simple to implement, and the number of agents can change dynamically. However, independent MARL does not work well if the agent's operations

affect each other, either in a collaborative or competitive way.



(a) Independent PPO



(b) Multi-agent PPO

Figure 2.5: The workflow of (a) independent PPO (IPPO) and (b) Multi-Agent PPO (MAPPO) [4].

In the case the agents work together in collaborative or competitive fashion, the information such as state, action, and reward can be shared between agents during the training and operating phase, therefore improving the agent performances [4]. There are multiple MARL algorithms doing so, such as MAA2C, MATRPO, and MAPPO, which are the MARL version of A2C, TRPO, and PPO, respectively [4]. While a MARL algorithm can have a better performance compared to the equivalent independent MARL algorithm, it is difficult to change the number of agents dynamically during the operation. Figure 2.5 shows the differences between IPPO and MAPPO: IPPO agents work independently with each other, while MAPPO agents share the state and

action information with each other.

2.2.3 Network Slicing

Network slicing, particularly for 5G networks, has been an important research topic in the networking area to provide isolated virtual networks for multiple services. In [46], authors proposed a resource allocation framework for network slicing using an iterative algorithm. The numerical evaluation shows that the algorithm has fast convergence in a wide range of static and dynamic environments.

In [47], authors proposed using DRL for RAN slicing in drone-assisted applications and aimed to maximize the number of slices within a given request set. The model monitors the current states of the substrate network and the requested slice and then decides the resource allocation for that slice. The authors conducted a simulation for evaluation, and the results show a high rate of successfully deployed slices in a resource-limited substrate network.

In [48], authors presented a Deep Q-Learning algorithm for end-to-end network slicing in 5G networks. The authors first formulated the problem as a mixed-integer programming problem and then proposed a Deep Q-Learning algorithm that considers both RAN slices and core network slices to dynamically allocate resources to maximize the number of users. The simulation results show improvement in the average access rate compared to the optimal allocation scheme that works only on the access side.

Atlas [49] is a solution for online network slicing configuration. Atlas focuses on reducing the simulation-to-reality discrepancy by using a learning-based simulator for the offline training phase. The simulator uses Bayesian optimization to optimize the simulation parameters to be close to real environments. During online training and operation, they proposed a safe exploration policy. The authors implemented and evaluated Atlas on a 5G testbed built on open-source projects. The evaluation results show a significant reduction in resource usage and an improvement in quality of experience.

In [50], authors focused on network slicing at the edge. In 5G and beyond 5G networks, Multi-Access Edge computing (MEC) is becoming increasingly important. A MEC has fewer computing resources compared to data centers but can provide low latency for particular users that are currently near that MEC, making it suitable for low-latency services. The paper proved that the resource allocation problem on MECs is NP-hard and proposed a near-optimal algorithm leveraging the similarities between edge nodes.

OnSlicing [51] is an end-to-end network slicing using DRL. OnSlicing has agents to slice for RAN, transport network, core network, and edge. The authors proposed a constraint-aware policy update method to reduce violations when the DRL chose unoptimal actions. OnSlicing also uses a proactive baseline-switching algorithm to switch to a traditional rule-based algorithm when actions from the DRL model deviate too much from the optimal. The evaluation results show that OnSlicing reduces resource usage by 12.5% compared to state-of-the-art online DRL solutions without service violations.

Authors in [6] propose a conceptual architecture for 6G networks with built-in support for AI-native services. The authors also propose leveraging AI for 6G network management tasks, including resource allocation for network slicing. The authors present an example use case that uses DRL to create network slicing for an AI workload. The numerical simulation shows the feasibility of the proposed approach.

2.2.4 RL for resource scaling

Researchers have utilized Reinforcement Learning (RL) as well as Deep Reinforcement Learning (DRL) for resource scaling, particularly computing resources thanks to its efficiency in dealing with complex and dynamic environments.

In [52], State–action–reward–state–action (SARSA) and Q-learning are used to do auto-scaling for cloud workloads. The authors evaluate two algorithms using OpenStack [53], which is an open-sourced framework for deploying and managing cloud

computing infrastructure.

Authors in [54] propose a safe RL-based auto-scaling for scaling containers in multi-access edge computing environments. Authors formulated the latency requirements as Linear Temporal Logic and showed that it can be used to guide the RL agent to learn auto-scaling algorithm in a safe way that guarantees convergence.

In [55], the authors propose to use Deep Q-Network, a simple DRL algorithm, to do auto-scaling for Virtual Network Functions (VNFs). VNFs are the network functions (such as router, firewall, load balancer, and deep packet inspection) implemented entirely in software. In this work, each VNFs has a specific amount of CPU, RAM, and storage, with a specific processing capacity. The Deep Q-Network algorithm monitored the CPU, memory, and storage usage in real-time and made scaling decisions. Similarly, in [56], authors also propose Deep Q-Network to dynamically scale the number of VNFs, targeting 5G use case. The authors also utilize mean CPU usage among active VNFs as on metric for the input state.

In [57], the authors utilize Deep Q-Network to do auto-scaling specifically for video conferencing systems. In video conferencing systems, the media servers in-change of processing and forwarding video streams of participants require a high amount of CPU, memory, as well as network bandwidth. As the number of participants changed over time, the Deep Q-Network monitored the state of the media servers and the video conferences to dynamically increase or decrease the number of media servers.

In [58], the authors leverage Deep Q-Network and Double Dueling Deep Q-Network (D3QN) to dynamically provision computing resources for VMs in cloud environments. The models are trained and evaluated using CloudSim [59] simulator, targeting Amazon AWS environment.

In [60], the authors use Proximal Policy Optimization (PPO), a new DRL algorithm that supports continuous action space for auto-scaling container workloads in edge computing. The authors conduct training and evaluation using simulation and compare PPO with Q-learning for this particular problem, with PPO achieving better

performance compared to Q-learning.

All the above work focuses on traditional computing resources with CPU, memory, and storage, using CPU utilization as the key metric for auto-scaling. Most of the work utilizes simple RL and DRL for resource scaling. While we cannot apply these researches directly to DRAFAS, they prove the effectiveness of RL and DRL for resource scaling, inspiring us to use DRL for resource allocation in DRAFAS.

III. Design

In this section, we first introduce the overall architecture and workflows of our proposed framework in detail. Secondly, we present the DRL algorithm in detail, including how we chose the DRL algorithm, the state, action, and reward. Thirdly, we present the two classes of rule-based algorithms.

3.1 Overall architecture and workflows

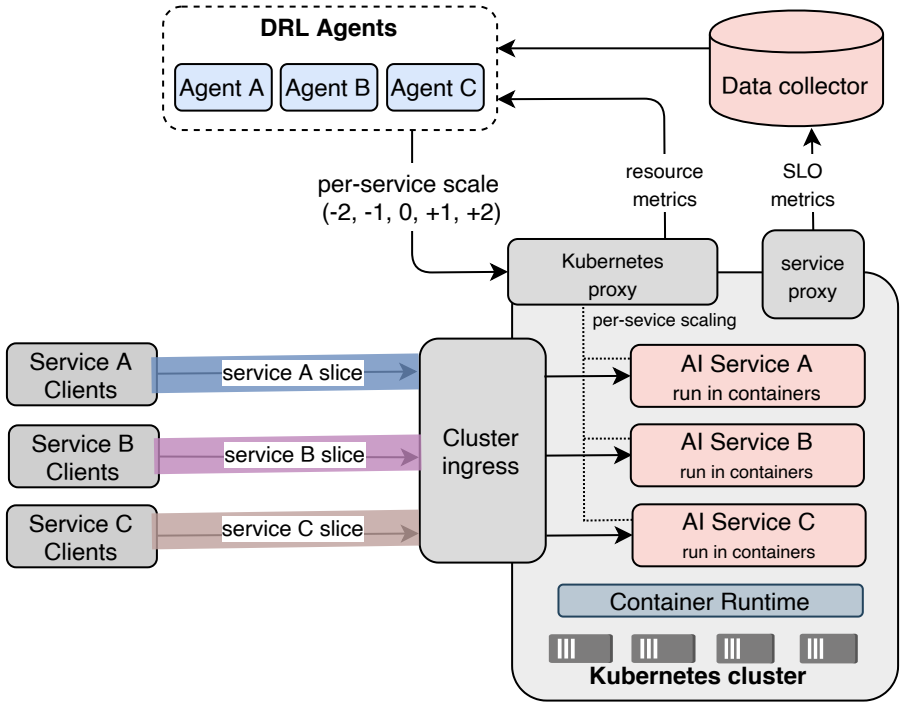


Figure 3.1: Architecture overview of DRAFAS for multiple AI-native services.

Figure 3.1 shows the architecture overview of DRAFAS with multiple AI-native

services being served in the same computing cluster. Each service can optionally have one network slice to preserve network resources. Different services can also share the same network slice. The detailed architecture and workflow of each service is shown in Figure 3.2.

Figure 3.2 shows the detailed architecture and workflow of DRAFAS applied to one AI-native service. The architecture and workflow are similar for all AI-native services. The computing resources include multiple servers with CPUs, memory, storage, GPU, and GPU memory. The AI-native service is deployed inside the container to ensure the security isolation needed in a multi-tenant environment. Each container has a fixed dedicated computing resource: the amount of CPU, memory, and storage, and especially, the amount of GPU and GPU memory. The computing resources are isolated between containers to ensure performance isolation. Each service can have one or multiple replications (replicas) and the incoming inference requests are distributed evenly between replicas so that the load is evenly distributed. The number of replicas is controlled and adjusted dynamically by the DRL agent.

Before the client request arrives at the service computing cluster, the request needs to go through the network infrastructure. For some critical services, there is a need to ensure that the connection between the client and the service cluster is maintained with some specific requirements such as high availability, high bandwidth, and/or low latency. For example, an AI service for stock trading may not need high bandwidth but it needs high availability and very low latency. An AI service for real-time object detection and segmentation used in traffic or security surveillance needs both high bandwidth and low latency. For such services, it is necessary to ensure network resources available to them. DRAFAS addresses this requirement by providing optional network slicing for such services.

DRAFAS workflow includes two phases: deployment and operation.

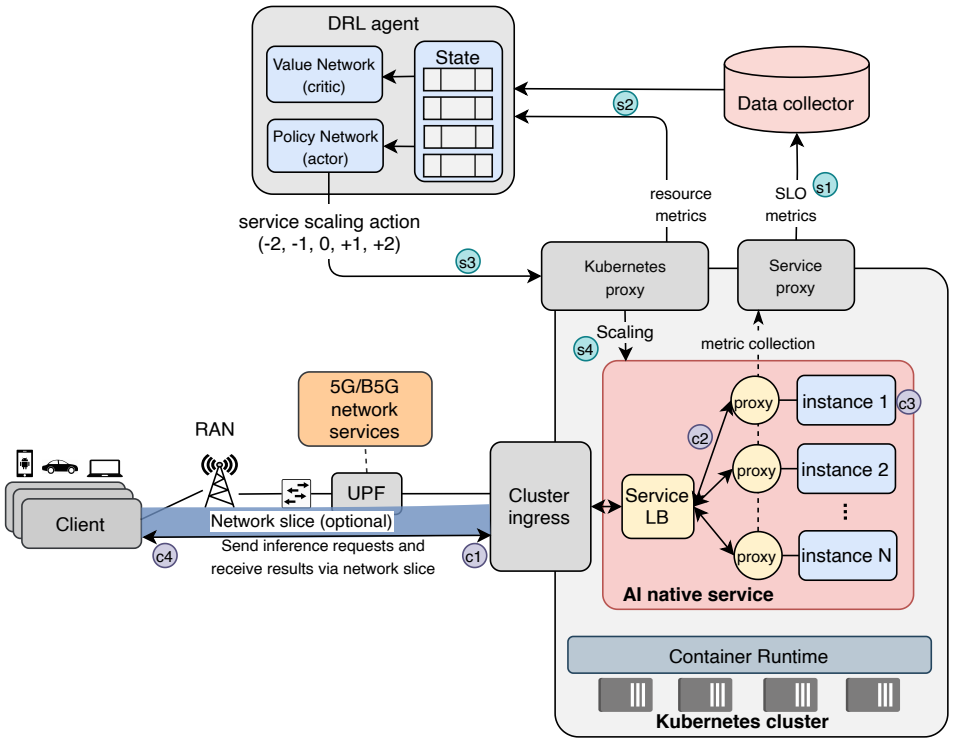


Figure 3.2: Detailed architecture and workflow of DRAFAS for an AI-native service.

3.1.1 Deployment phase

In the deployment phase, one instance (container) of the AI-native service is deployed into the cluster for profiling. The purpose of profiling is to get the service performance metric to feed into DRAFAS simulator for training the DRL agent and parameter optimization for the rule-based resource allocation algorithms. The details of this process and the metric collected will be presented in Section 4.4.

After the training (for the DRL agent) and parameter optimization (for the rule-based algorithms), The model is deployed if using the DRL agent, or the optimized rule-based logarithm is deployed. The AI service is now deployed with the desired number of initial replicas, ready for serving inference requests from clients.

In the deployment phase, the network slice can also be deployed for the AI-native

service if required. The network slicing parameters such as bandwidth and QoS parameters are configured via the 5G/B5G (beyond 5G) configuration interface. The configuration parameters are then propagated to the network components such as Radio Access Network (RAN) and User Plane Function (UPF) to create an end-to-end network slice with desired network resources. Note that while we use 5G/B5G architecture for network slicing, the same principle is also applied to other types of networks such as broadband wired network connections. DRAFAS computing resource allocation works independently with the network slicing method.

3.1.2 Operation phase

In the operation phase, clients send inference requests to the services based on their own needs. The number of requests can change dynamically over time, and the time needed to process each request can also be different between requests. The DRL agent / rule-based algorithm continuously monitors the current state of the cluster and makes scaling decisions if needed.

The workflow of the client is as follows:

- A client sends an inference request to the service, optionally via the network slicing (c1).
- The request arrives at the service load balancer and is distributed to one of the running replicas. The load balancer works in a round-robin mechanism, evenly distributing the workload to all the replicas. There is one proxy for each replica that intercepts requests forwarded from the service load balancer, for the purpose of metric collection, such as the number of requests per second. The proxy should be transparent to both the service instance and the client (c2).
- The inference request is processed in the selected replica (c3).
- The inference result is sent back to the original client. The inference result also goes through the proxy, again for the purpose of SLO metric collection and

calculation, such as the number of in-flight requests (i.e., number of requests currently in processing), and request processing time (c4).

The workflow of the DRL agent / rule-based algorithm is as follows:

- While the service is running, the inference requests from clients change, causing changes in the resource usage and SLO metrics. The resource usage metrics are exposed via Kubernetes proxy, while the SLO metrics is collected periodically and stored in a data collector (s1). The period for collecting SLO metrics is one second.
- Every 30 seconds, The DRL agent / Rule-based agent collects the SLO metrics over the last 30 seconds as well as current available resource information (s2).
- The DRL agent / Rule-based decides the scaling action, checks if it is valid, and calls the scaling API via Kubernetes proxy (s3).
- The Kubernetes scale the service according to the received action (s4). The cycle repeats every 30 seconds until the service and the according DRL agent / rule-based agent is removed.

3.2 DRL Algorithm

When designing the DRL agent, there are two important aspects: selecting the suitable DRL algorithm, and designing the state, action, and reward.

3.2.1 Choice of DRL algorithm

To select the suitable DRL algorithm for DRAFAS, we first surveyed the available DRL algorithms as presented in Section 2.2.2. We analyze the requirements and characteristics of the agents and environment in DRAFAS to select a suitable one.

Firstly, there can be multiple AI-native services deployed, and each needs to have its own DRL agent, which means that we should use a multi-agent DRL algorithm.

Secondly, new services can be added at any time, and an existing service can be removed at any time; requiring that the DRL algorithm should be able to add or remove agents at any time. Thirdly, while an AI service can affect the performance of other AI services to some extent, the effect is minimal by ensuring performance isolation. Therefore, the agent of each AI service works independently with each other with one constraint that the total resource usage of all services should not exceed the total resource available in the cluster.

Given the above requirements and characteristics, independent MARL is suitable for DRAFAS: each AI-native service has its own DRL agent, independent MARL can scale well, agents can be added or removed on the fly, the algorithm works well when each agent works fairly independently and does not affect each other. To overcome the resource constraint requirements, we create an action mask during runtime and block any action that allocates to the AI service more resources than the currently available free resources.

Additionally, with the advantages of MPPO over other DRL algorithms, we use Independent Maskable PPO (IMPPO) as the DRL algorithm for resource allocation in DRAFAS. Since we controlling the number of container replicas for each AI service, the action space is discrete; therefore, Maskable PPO allows masking out invalid actions such as scaling up when there is not enough free resources, or scaling down when there is only one remaining replica. The action mask is also needed for a rule-based algorithm to filter out invalid action during operation.

3.2.2 State

In DRAFAS, we use Independent Maskable PPO, which means that each AI-native service has its own DRL agent or rule-based agent. The agent only has information about its service and overall resource information of the whole cluster. It does not have any information about the other services.

The state at time t for a service is the combination of five metrics: the average

request per second, the average instance (container) utilization, the action mask, the average SLO violation rate, and the number of instances. All of the metrics are normalized to the range between 0 and 1. We maintain the history of each metric with a history size of 5, which is equal to 2.5 minutes considering the model calculates new action every 30 seconds. Below are the explanations of each metric. While the meaning of average request per second, average SLO violation rate, and number of instances are intuitive, the average instance utilization will be explained in more detail later in this Section. The action mask metric will be explained in more detail in Section 3.2.3.

- **Average request per second**: number of requests to the AI-native service from last 30 seconds, average to each second. To normalize the value, we divided it to the maximum number of requests per second in the trace file from Microsoft Azure ML inference dataset [12]. During the operation, the value can still get bigger than 1, so we need to cap the normalized value to 1.
- **Average instance utilization**: the container utilization percentage, measured indirectly. Naturally, the average instance utilization value is ranged between 0 and 1.
- **Action mask: the value of action mask**, indicating which action is valid and which action is invalid.
- **Average SLO violation rate**: the average SLO violation rate in last 30 second. A request is considered to have SLO satisfied if it is successful and the request processing time is under a certain threshold. The threshold is different for different AI-native services. Naturally, the SLO violation rate is ranged between 0 and 1.
- **Number of instances**: the current number of instances. Normalized by dividing the current number of instances by the max possible number of instances.

We use average instance utilization instead of GPU, GPU memory, CPU, memory, and storage utilization due to the technical limitation of fractional GPU monitoring.

Intuitively, GPU, GPU memory, CPU, memory, and storage can be used as the input metrics for the DRL agent, which is the case in many previous work [55, 57, 56, 58, 60]. However, fractional GPU monitoring is not available in DRAFAS. DRAFAS uses Multi-Process Service (MPS) for fractional GPU sharing between containers to take many benefits of MPS such as good performance and error isolation, high GPU utilization, and high total throughput, as well as flexibility and high availability in many GPUs. However, MPS has one downside that it does not support fractional GPU utilization monitoring. Furthermore, in AI workload, GPU is usually the most expensive and important resource and is usually saturated before other resources such as CPU and memory.

To address the issue, we replacing the GPU, GPU memory, CPU, memory, and storage utilization as one abstract metric called instance (or container) utilization. In DRAFAS, we use MPS to divide GPU into partitions with equal GPU processing power and GPU memory, and allocate one partition to each container. For performance isolation, the amount of CPU, memory, and storage is also fixed for each container. Because of that, different replicas of a same AI-native service have equal processing capability. We model a replica of a service with processing capability P, which means that it can process P requests in parallel at any given time. By putting a proxy in between the replica and the service load balancer, as shown in Figure 3.2, we can monitor how many in-flight requests (i.e., requests are currently in processing) at a time. By sampling the value every second between each scaling action (i.e., 30 seconds duration), we have an approximate value of instance utilization.

Using instance utilization brings several benefits. Firstly, it addresses the problem of lacking fractional GPU utilization monitoring. Secondly, it bundles five metrics: GPU, GPU memory, CPU, memory, and storage into one abstract metric, greatly reducing the monitoring overhead. Thirdly, as we use a simulator to train and validate both the DRL agent and the rule-based agent before deploying into real environments, simplifying metrics helps reduce the simulation complexity and improve the simulation accuracy.

3.2.3 Action

In DRAFAS, we designed the action to have five discrete values: -2, -1, 0, 1, 2, which are equivalent to reducing the number of replicas by 2, reducing the number of replicas by 1, maintain (do nothing), increasing the number of replicas by 1, and increasing the number of replicas by 2, respectively.

However, not all action is available at all times. We limit the minimum number of replicas to 1. Additionally, the total number of resource usage by all services cannot exceed the total resources of the cluster, i.e., we cannot scale up the service if the current free available resource is not enough. Additionally, if the last action is scale-up, the actions during next three minutes cannot be scale-down to ensure the stability.

One solution is to punish invalid actions with negative rewards. However, this only applies to the DRL agent, as the rule-based agent can still produce invalid action and possibly cause system error. Additionally, this does not guarantee that actions are always valid in the DRL agent, it only reduces the possibility of producing invalid actions.

To solve these issues, we utilize Maskable PPO [43] or MPPO, which allows putting action masking into the model operation, making the model to provide only valid action. MPPO accepts an action mask function that returns an array indicating which action is valid and which is not. MPPO then uses action mask to re-sampling the output, effectively making the model to output only one of the valid actions. In DRAFAS, we also used the action mask to filter out invalid actions in the rule-based agent before applying the action to the service. Algorithm 1 shows the detailed logic of action mask calculation. The action mask is an array with a size of 5, and the value at index 0, 1, 2, 3, 4 represents the mask for action -2, -1, 0, 1, 2, respectively. A value of 0 means the action is invalid, and a value of 1 means the action is valid.

Algorithm 1 Valid Action Mask Calculation

```
function ValidActionMask
    Initialize action_masks as an array of ones with size 5
    remain_resources ← GET_REMAIN_RESOURCE()
    if service.num_inst = 2 then
        action_masks[0] ← 0
    end if
    if service.num_inst = 1 then
        action_masks[0] ← 0
        action_masks[1] ← 0
    end if
    if scale_down_cooldown > 0 then
        action_masks[0] ← 0
        action_masks[1] ← 0
    end if
    if remain_gpus = 1 then
        action_masks[4] ← 0
    end if
    if remain_gpus = 0 then
        action_masks[3] ← 0
        action_masks[4] ← 0
    end if
    return action_masks
end function
```

We also include the action mask as one of the input state metrics for the DRL agent. This helps the agent learn from the action mask information to identify which actions are valid. The action mask is encoded as a one-hot binary value. Equation 3.1 shows the formula to calculate the normalized value of the action mask. Since the maximum value of the action mask is $0b11111$ when all actions are valid, we normalize it by dividing the value by $0b11111$.

$$\text{action\_mask} = \frac{M[0] \ll 4 \,|\, M[1] \ll 3 \,|\, M[2] \ll 2 \,|\, M[3] \ll 1 \,|\, M[4]}{0b11111} \tag{3.1}$$

3.2.4 Reward

Since the DRL algorithm is trained to maximize the reward function, we design the reward to decrease when the SLO violation rate increase and the number of replicas increase. Specifically, we formula the reward in Equation 3.2, where α and β are the weight for slo_violation_rate and num_inst_norm, respectively. We empirically set $\alpha = 0.9$ and $\beta = 0.1$ so that the slo_violation_rate is prioritized but the num_inst_norm is also considered, making the DRL agent try balancing between maintaining low SLO violation rate and keeping low resource usage (i.e., low number of replicas).

$$\text{Reward} = -\left(\alpha \times \text{slo\_violation\_rate} + \beta \times \text{num\_inst\_normalized}\right) \tag{3.2}$$

We also use the reward function in Equation 3.2 for parameter optimization in the rule-based algorithms.

3.3 Rule-based algorithms

In DRAFAS, beside the DRL agent, we also present two classes of rule-based algorithms. The rule-based agent interacts with the environment in the same way as the DRL agent: it observes the current service and resource states as well as the SLO metrics, then produces scaling action every 30 seconds. The logic of two rule-based algorithms is presented in Algorithm 2 and Alogorithm 4.

Algorithm 2 Rule-based scaling v1

```
1: function RULE_BASED_SCALING_V1(inst_util, action_mask,
   inst_util_high_2, inst_util_high_1, inst_util_low_1, inst_util_low_2)
2:     action ← 0
3:     if inst_util > inst_util_high_2 then
4:         action ← 2
5:     else if inst_util > inst_util_high_1 then
6:         action ← 1
7:     else if inst_util < inst_util_low_2 then
8:         action ← −2
9:     else if inst_util < inst_util_low_1 then
10:        action ← −1
11:    end if
12:    action ← ACTION_ADJUSTMENT(action_mask)
13:    return action
14: end function
```

In Rule-based algorithm v1, presented in Alogirthm 2, intuitively, we use instance utilization as the metric to determine the suitable action. There are four threshold values of instance utilization for action decision. After the action is decided, the action is adjusted with action mask if necessary, following the logic in Algorithm 3.

For the rule-based agent, the action mask does not block the agent from generating an invalid action, the action mask only prevents the action from being applied to the cluster. Therefore, in the rule-based agent, it is possible that the action is scaling up 2 replicas, and the cluster only has enough resources to scale up 1 replicas, and the agent ends up doing nothing because scaling 2 replicas is an invalid action and is blocked. However, a more suitable action is to scale up 1 replica. Similarly, the agent can decide to scale down by 2 replicas when it is only possible to scale down 1 replicas. In this case, the better action is to scale down by 1 replica instead of doing nothing. Algorithm 3 shows how action can be adjusted in these cases.

Algorithm 3 Action Adjustment Based on Action Mask

if `action` $= -2$ **and** `action_mask`$[0] = 0$ **then**
    if `action_mask`$[1] = 1$ **then**
        `action` $\leftarrow -1$
    else
        `action` $\leftarrow 0$
    end if
else if `action` $= -1$ **and** `action_mask`$[1] = 0$ **then**
    `action` $\leftarrow 0$
else if `action` $= 2$ **and** `action_mask`$[4] = 0$ **then**
    if `action_mask`$[3] = 1$ **then**
        `action` $\leftarrow 1$
    else
        `action` $\leftarrow 0$
    end if
else if `action` $= 1$ **and** `action_mask`$[3] = 0$ **then**
    `action` $\leftarrow 0$
end if

In Rule-based algorithm v2, presented in Algorithm 4, besides the instance utilization metric, we also use SLO violation rate as the second metric. With two metrics, it is difficult to design the rule-based algorithm with four actions, therefore we omit the two actions +2 and -2, only considering actions -1, 0, and 1. After the action is decided, the action is also adjusted with action mask if necessary, using the logic in Algorithm 3.

Algorithm 4 Rule-based scaling v2

function RULE_BASED_SCALING_V2(slo_violation_rate, inst_util_usage, action_mask, slo_violation_high, slo_violation_low, inst_util_high, inst_util_low)
    action $\leftarrow 0$
    if slo_violation_rate $>$ slo_violation_high **or** inst_util_usage $>$ inst_util_high **then**
        action $\leftarrow 1$
    else if slo_violation_rate $<$ slo_violation_low **and** inst_util_usage $<$ inst_util_low **then**
        action $\leftarrow -1$
    end if
    action $\leftarrow$ ACTION_ADJUSTMENT(action_mask)
    return action
end function

The threshold values used in both rule-based algorithms are service-specific, i.e., they are optimized for each type of service via parameter search. The detailed process of parameter optimization will be presented in Section 4.4.

3.4 Simulator

In DRAFAS, each step takes 30 seconds in real environments. Considering that 100,000 steps will take around 35 days, training DRL agent will take a few hundred thousand steps, it is impractical to train the DRL agent in the real environment. To solve this issue, we develop a simulator to train the DRL agent as well as do parameter optimization for the rule-based agents. The simulator needs to simulate the clients and the services in the computing cluster.

3.4.1 Service simulator

To simulate the AI-native services, we model an AI service with the parameters listed in Table 3.1. A service is deployed with `initial_inst` instances. During the operation, the DRL agent of the rule-based agent can scale up or down the service, effectively changing the `current_inst`.

Table 3.1: Parameters used for modeling AI-native services in DRAFAS simulator

Parameter Name	**Description**
`initial_inst`	The initial number of instances when the service is deployed.
`current_inst`	Current number of instances.
`resource_per_inst`	The amount of resource allocated to each instance.
`capacity_per_inst`	The processing capacity of each instance, i.e., how many request each service instance can process at the same time.
`queue_size_per_inst`	The size of waiting queue for each instance.
`static_proc_time_range`	The range of static processing times for a request, defined as a minimum and maximum value. The processing time of each request during simulation is randomed between the provided range.
`req_delay_slo`	The service-level objective (SLO) for request delay, specifying the maximum allowed response time for an inference request.
`startup_time`	The time required to start up a new instance and make it operational.

The request processing workflow is as follows. When a new request arrives at the service, the request is distributed to an available instance using the round-robin

mechanism, similar to the real environment. If the selected instance is processing with all capacity, the request is placed in a queue and waits for its turn to be processed. If the waiting queue size is equal to the `queue_size_per_inst`, the request is dropped.

When the request exits the queue and makes it to the processing state, the instance will wait for a random time in the range of `static_proc_time_range`, simulating the request processing time. Then, the request is marked as processed. The instance continues to process the next request in the queue.

When the service is scaled up, new instance(s) are added to the service. The instance(s) needs to wait for `startup_time` before it can start processing a request. This simulates the real environment where the instance needs time to startup as well as loading necessary components (e.g., ML models) before it can start serving inference requests. When the service is scaled down, the instance(s) on the last of the instance list is removed after all the queue requests are finished, effectively simulating gracefully shutdown of the container in the real environment.

With the simulator, we can calculate exactly every metric needed for the input state of DRL agent. However, in the real environment, the metrics reported are measured using a sampling mechanism, i.e., the metric is scraped every second, then the value is averaged over the monitoring duration, which is 30s in DRAFAS. Therefore, in the simulator, we also use the sampling mechanism to measure the metrics.

3.4.2 Client simulator

The simulation of the client is much simpler. The client simply sends requests following a set of provided trace files. A trace file includes multiple lines, each has two variables: one is the duration in second, and the other is the number of inference requests during that duration. Each client type (i.e., client of which service) has its own request scaling parameter, which is used to adjust the number of requests ratio compared to the trace file. After multiplying the number of requests from the trace file, the clients send these requests to the service. The requests are spread evenly

during the duration. After the duration is finished, the client continues to process with the next line in the trace file regardless of the status of the requests in the previous duration.

With the simulation of client and AI services, we can train the DRL agent as well as do parameter optimization for the rule-based agent much faster. For example, 100,000 steps in the real environment will take around 35 days but only take more than 10 minutes in DRAFAS simulator in our experiment.

IV. Implementation

In this chapter, we present how we implemented the components of DRAFAS: the end-to-end testbed with client emulator, computing cluster, and network slicing. We also present the implementation of three AI-native services for evaluation: a chatbot service based on Large Language Model, an image classification service, and a text to speech services.

4.1 DRL agent and Simulator

For the DRL agent, we implement it using Stable-baseline3 [61], which is one of the most popular frameworks for building, training, and deploying DRL models. We set the policy network and value network to both have two layers with 64 neurons in each layer. We use ReLU [62] as the activation function. The input layer has 5 metrics with a history size of 5, making a total 25 input features.

The Simulator includes a lot of asynchronous processes, therefore it is best suited to be implemented with a discrete event framework. Therefore, we use SimPy [63] to implement the simulator. Each service and client is implemented as a SimPy process. Additionally, the metric monitoring function is also implemented as SimPy process. We set the time unit in SimPy equal to 1 ms in the real environment. Every time SimPy clock advances, it looks for the expected event to process at that time point, and then processes to the next clock cycle. The simulation speed of SimPy depends on the complexity of the simulated process and the clock speed of the CPU core.

4.2 End-to-end testbed

The end-to-end DRAFAS testbed is deployed using multiple open-sourced tools and frameworks. Figure 4.1 shows how function blocks in the DRAFAS architecture are implemented with open-sourced tools and frameworks.

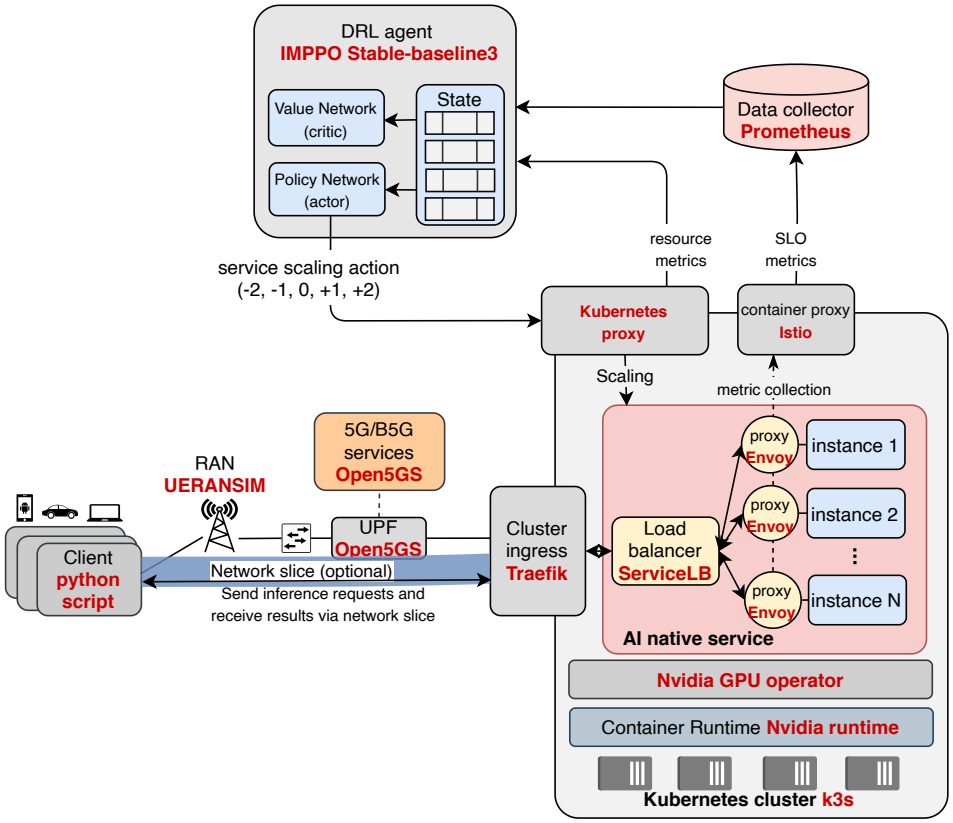


Figure 4.1: The components used for DRAFAS testbed implementation.

Table 4.1 gives an overview of these tools and frameworks. UERANSIM and Open5GS are used to deploy 5G infrastructure and create network slicing via Open5GS configuration web GUI. We use K3s Kubernetes distribution as the container orchestrator. Nvidia container runtime and Nvidia GPU operator are used to allow containers to use fraction GPUs. Envoy is used as the proxy to intercept and monitor inference

requests. Envoy is transparent to the client and service and does not require any code modification from services. Istio is the tool to automatically deploy Envoy proxy along with each container and provide the APIs for Prometheus to collect data.

Table 4.1: Open-sourced tools and frameworks used for DRAFAS testbed implementation

Component	**Description**
UERANSIM [15]	A user equipment (UE) and radio access network (RAN) simulator for 5G networks.
Open5GS [14]	An open-source 5G core network framework, providing support for 5G and LTE networks.
Traefik [64]	A cloud-native HTTP reverse proxy and load balancer designed to simplify deployment of services and APIs.
ServiceLB [65]	A lightweight load balancer integrated into K3s for managing service traffic in Kubernetes clusters.
K3s [65]	A lightweight, certified Kubernetes distribution designed for resource-constrained environments.
Nvidia Container Runtime [66]	A runtime for enabling GPU acceleration in containerized environments, specifically designed for NVIDIA GPUs.
Envoy Proxy [67]	A high-performance proxy for cloud-native applications, often used in service meshes and API gateways.
Istio [68]	An open-source service mesh that provides traffic management, observability, and security for microservices.
Kubernetes Proxy [69]	A component of Kubernetes that expose Kubernetes control and metric APIs.
Prometheus [70]	An open-source systems monitoring and alerting toolkit, commonly used for metrics collection in cloud-native environments.
Stable-Baseline3 [61]	A collection of reliable reinforcement learning algorithms implemented in Python, widely used for DRL research and applications.

We develop a client emulator using Python asyncio [71] - an asynchronous Python framework. Similar to the client simulator in the DRAFAS simulator, the client emulator reads request trace from a trace file to decide how many requests to send each time duration. The requests are asynchronously sent and they are evenly distributed during the time duration. However, different from the simulator, the client emulator sends real ML inference requests to the real AI-native services and receives actual inference results.

4.3 AI-native services

To evaluate DRAFAS, we developed and deployed three AI-native services: a Chatbot service, an image classification service, and a text to speech services. The implementation of these services is as follows:

- **Chatbot service with Large Language Models**: We deployed a chatbot service using ollama [72] - an open-sourced framework to deploy Large Language Models (LLMs). We deployed Llama3.2:1b [73] version on Ollama, which then serves the model via REST APIs.

- **Image classification service**: we developed an image classification service using Pytorch [74] and the pre-trained Resnet152 model [75]. The service accepts a batch of four images, does preprocessing, and then returns the classification results. The service serves inference requests vis REST APIs.

- **Text to speech service**: we deployed text to speech service using Coqui open-sourced framework [76]. The service accepts text with variable length and converts it into audio. Similar to the above two services, Coqui text to speech service also serves inference requests via REST APIs.

For each service, we also develop the equivalent code in the client emulator to send inference request to that service.

4.4 Training

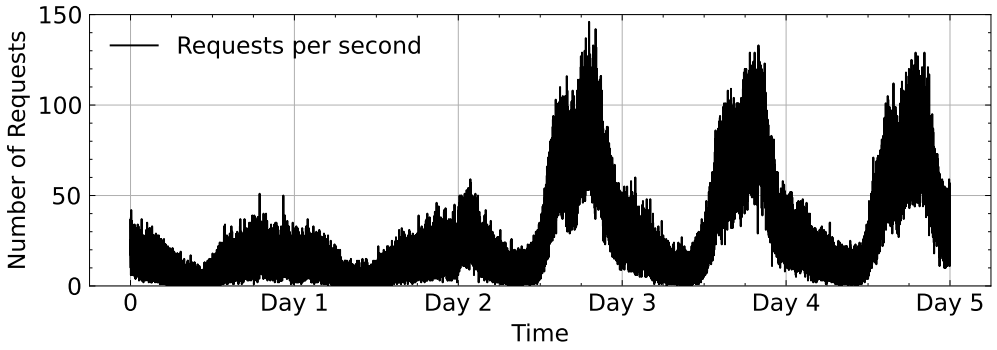
In this section, we cover in detail how the DRL agent is trained and how to optimize parameters for the rule-based scaling algorithms. To so do, we use public ML inference dataset. We deployed the three AI-native services into DRAFAS cluster and profiled them to get the service parameters for the simulation.

4.4.1 Dataset

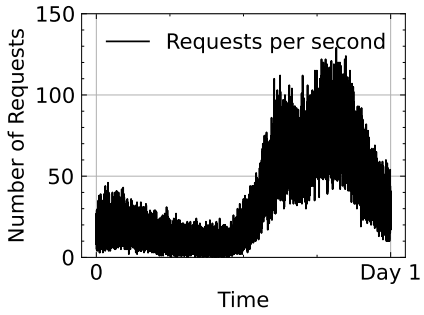
To make the client emulator behave close to the real world, we use public ML inference traces, specifically LLM inference trace from Microsoft [12]. The trace consists of seven days trace log collected from multiple LLM inference services in Azure, from 10^{th} to 19^{th} May 2024. We processed the trace to get the number of requests per second for every second in the trace. The extracted trace file includes 604800 data points, spread over 7 days in the original trace. We divided the trace into three sets: 5 days (432000 data points) for the train set, 1 day (86400 data points) for the validation set, and 1 day (86400 data points) for the test set. Figure 4.2 shows the distribution of requests per second over time in the train set, validation set, and test set. Intuitively, the traffic pattern seems to be repeated every day with different scaling.

Table 4.2 shows the minimum, maximum, average, and standard deviation of the number of requests per second in the train set, validation set, and test set. The minimum value is 0, which means that there are periods when there is no request from clients. The high standard deviation shows that while there is a traffic pattern in a large time scale (e.g., day), the number of requests per second fluctuates a lot at a small time scale (e.g., second).

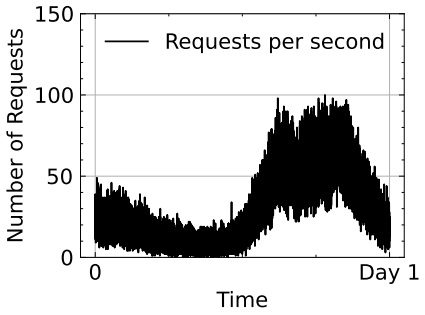
While the dataset of request traces above is satisfied for training and testing in the DRAFAS simulator, it is not enough for testing in the real environment. In the real environment, we deployed three real AI-native services presented in Section 4.3 and therefore we need a dataset of request contents for each service. The client emulator sends a different number of requests per second following the traffic trace, and the



(a) Train set



(b) Validation set



(c) Test set

Figure 4.2: Number of request per second over time in train set, validation set, and test set.

content of each request is chosen from the dataset specified generated for each service type. The details is as follows:

- **Chatbot service**: The client sends a random text question to the Chatbot service and receives the answer. The text query is selected from a set of 500 queries generated procedurally.
- **Image classification service**: The client sends a batch of random four images to the image classification service and receives classification results. The images are selected randomly from Cifar10 dataset [77]. The Cifar10 images are tiny,

Table 4.2: Statistics of request traces in train set, validation set, and test set. The unit is requests per second.

	Min	**Max**	**Average**	**Std. Dev.**
Train set	0	146	26.33	24.75
Validation set	0	129	34.00	26.39
Test set	0	100	28.82	20.78

therefore we scale up each image to the size of 512 before sending, which is closer to the typical image size.

- **Text to speech service**: The client sends a text string with different lengths and words to the text-to-speech service and receives the equivalent audio of that text string. The text string is selected from a set of 100 text strings, generated with the help of ChatGPT [78].

4.4.2 Service profiling

To make the training on DRAFAS simulator useful, we need to model the services in the simulator as close to the real environment as possible. Therefore we did profiling the service in the real DRAFAS computing cluster. The cluster consists of two servers, each server has an Intel Xeon 4114 CPU with 10 cores, 64 GB of RAM, and 1 Nvidia RTX 5000 GPU with 16 GB of VRAM. Each GPU is spatially partitioned into 4 MPS partitions, effectively creating a total of 8 vGPUs in the cluster. The resource allocated to one replica of each service is shown in Table 4.3. For image classification and text to speech services, they need more CPUs for pre-processing images as well as post-processing generated audio data. Since the GPU is partitioned equally both in GPU cores and VRAM, the VRAM amount for each vGPU is fixed at 4GB. In the three AI-native services we developed, there is a negligible need for long-term storage.

For each service, we deployed one container into the cluster. We wrote a bash

	Chatbot	Image classification	Text to speech
CPU cores	1	2	2
Memory	2GB	2GB	2GB
vGPU	1	1	1
VRAM	4GB	4GB	4GB

Table 4.3: Computing resource allocated to one container of each service.

script to measure the startup time, from the time we start deploying the container to the time the container is ready to serve incoming requests. The startup time includes container various processes such as container startup, service initialization, and ML model loading (which can take a lot of time). We then use the client simulator to send different amounts of requests at a fixed rate to determine the capacity (i.e., how many requests the service can process in parallel). Then we sent the requests in continuous mode (i.e., send a new request immediately after the previous request succeeded) to measure the processing time of each request without counting the queuing time. The results after profiling are shown in Table 4.4. The SLO delay requirements are chosen manually considering the average processing time as well as the characteristics of each service. Intuitively, we expect fast responses from the image classification service, but we are more tolerant to the chatbot and text to speech services.

4.4.3 Training DRL agent

Training setup

After service profiling, the DRL agent now can be trained using DRAFAS service and client simulator. There are two methods for training: train all the agents of all the services together, or train each agent with each service independently with each other. Considering that during the evaluation, we will evaluate all services and agents at the same time sharing the same computing clusters, training all agents at the same time

	Chatbot	Image classification	Text to speech
Capacity (number or request can be processed at a time)	1	1	1
Processing time range (ms)	140 - 420	60 - 80	130 - 390
SLO delay requirement (ms)	2500	500	2500
Startup time (ms)	13000	11000	21000

Table 4.4: Service parameters from profiling (except SLO delay requirement, which is set manually)

can produce a better performance. However, in a real environment, this is impractical because the number of services can be huge and we cannot retrain all agents when there is a new service.

Considering the above issue, we chose to train each agent of each service independently. Figure 4.3 shows how the DRAFAS client simulator, the service simulator, and the DRL agent / rule-based agent interact with each other to form a training session. Because the performance of each container is mostly independent of each other (i.e., the performance of one container is not affected by the workload of the other containers), the only remaining issue of training agents independently is the dynamic change of remaining computing resources. To simulate this, we randomize the remaining resources between zero and the actual remaining resources. The actual remaining resources equals total resources minus the resources currently used by the service.

The training is done on CPU on a machine with Intel Core i9-9880H and 64 GB RAM. We set the maximum number of training steps to 300,000, which is equivalent to more than 100 days in the real environment. We set the learning rate to linear learning rate, i.e., the learning rate starts with the value of 0.0002 and reduces as the training goes on. Linear learning rate can potentially help the model to learn quickly at the beginning (with a large learning rate) and stabilize toward the end of the training process (with now small learning rate). The total resources unit is set to the total

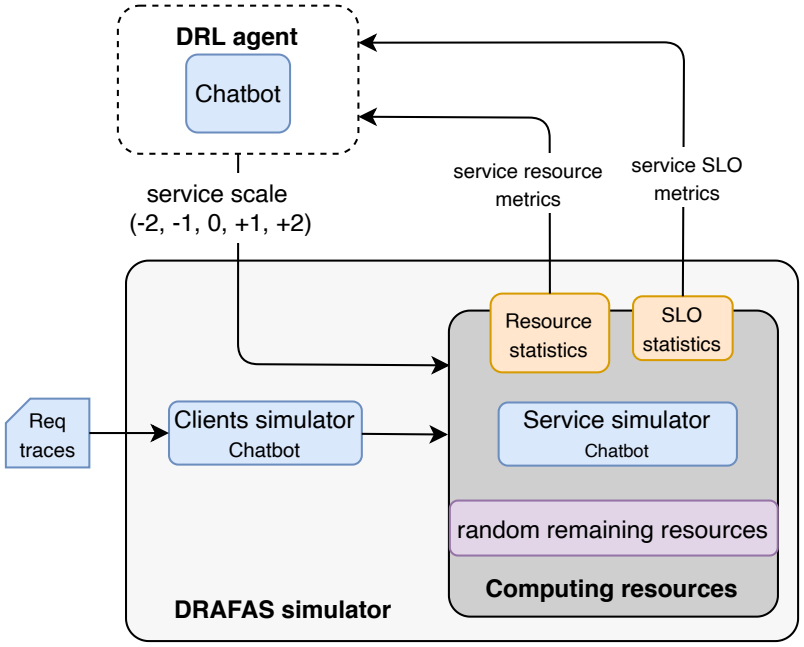


Figure 4.3: DRAFAS client and service simulator interacting with single service (Chatbot) and its corresponding DRL agent.

number of GPU slices because the GPU is the only constrained resource in the system.

The parameter setup for the training is summarized in Table 4.5. We periodically evaluate the trained model with the validation set every 14400 steps, which is equal to 5 days in the real environment. The validation is repeated three times to get the average reward, and the model with the best reward is saved. The training can terminated early before step 300,000 if there is no reward improvement after 4 validations. To make the training effective, we only check for no reward improvement after 8 initial validation cycles. Additionally, since we use the same training set for training all three DRL agents, we need to modify the request rate ratio to be suitable for each service. During the training, we set the request rate ratio to 0.15, 0.4, and 0.15 for chatbot, image classification, and text to speech services, respectively.

The other MPPO-specific parameters are kept default from Stable-baseline3 im-

Table 4.5: Training and validation parameters

Parameter	**Value**
Total resource units	8
Max Training Steps	300,000 (more than 100 days)
Learning Rate	$0.0002 \times remaining\_progress$
Validation Frequency (steps)	14400 (5 days)
Validation repeat	3 times
Early Stopping Criteria	4 cycles without improvement
Request scaling ratio	chatbot: 0.15, image classification: 0.4, text to speech: 0.15

plementation, which are shown in Table. 4.6.

Training results

Figure 4.4 shows the episode mean reward of three agents during the training. The mean rewards of all three agents increase quickly in the early training process and slow down as the training approaches the end. Figure 4.5 shows the value losses of three agents during the training. The losses are high at the start of the training process and reduced quickly, although with fluctuation. The trends of mean reward and value loss indicate that the training process went as expected and the model learned correctly to improve the reward function. This is further confirmed by the trend of validation mean reward, as shown in Figure 4.6.

The time to run training DRL agents for chatbot, image classification, and text to speech services are 23 minutes, 45 minutes, and 23 minutes, respectively, which is much faster than training in the real environment. The training time of image classification DRL agents is twice as much compared to the other two because the request scaling ratio of the image classification client is also twice the scaling ratio of the

Table 4.6: Default parameters for MPPO in Stable-Baselines3

Parameter	Value	Description
n_steps	2048	Number of steps to run for each environment per update.
batch_size	64	Minibatch size used for training.
gamma	0.99	Discount factor for rewards.
clip_range	0.2	Clipping range for the PPO objective function.

chatbot client and the text to speech client.

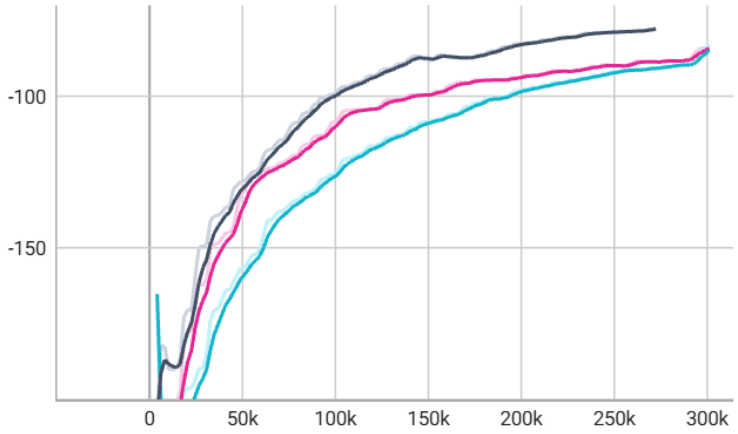
4.4.4 Parameter optimization for rule-based algorithms

To optimize the parameters for rule-based algorithms presented in Section 3.3, we use the same setup for training DRL agent as shown in Figure 4.3, but replacing the DRL agent with the rule-based agent. We optimize the rule-based agents on the validation set. We use Optuna [79], a hyper parameter optimization framework for better parameter optimization considering large search space.

The search ranges for each parameter in rule-based algorithm v1 are presented in Table 4.7, and the search ranges for each parameter in rule-based algorithm v2 are presented in Table 4.8. The optimization reward is the same as the reward function in the DRL agent, which is shown in Equation 3.2.

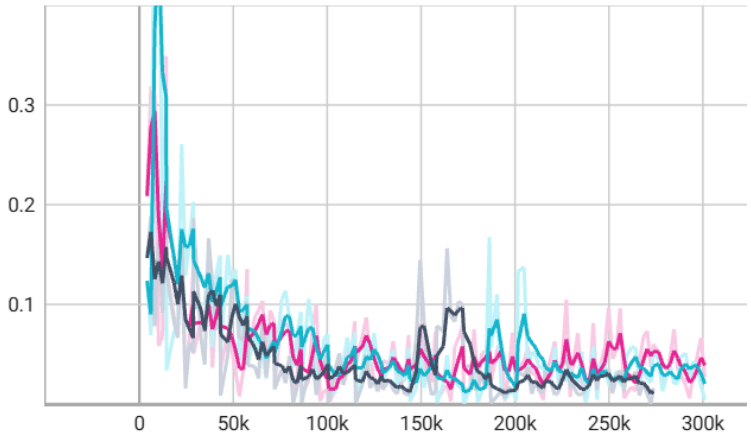
We ran the optimization for 100 iterations, which is equal to 100 days in real environments and similar to the training time. The result is shown in Table 4.9 for the rule-based scaling v1 algorithm, and in Table 4.10 for the rule-based scaling v2 algorithm.

We evaluated and compared rule-based algorithms v1 and v2 on the validation set. The results are shown in Table 4.11. Rule-based v2 is better than rule-based v1 for all three AI services by providing better reward value. Therefore, we chose rule-based v2 for final deployment on DRAFAS.



Run ↑	Smoothed	Value	Step	Relative
chatbot	-84.8143	-82.8835	301,056	22.75 min
image_classification	-77.8623	-77.4704	272,384	43.26 min
text_to_speech	-84.1774	-83.0042	301,056	22.46 min

Figure 4.4: Episode mean reward of three agents during the training.



Run ↑	Smoothed	Value	Step	Relative
chatbot	0.0203	0.0039	301,056	22.75 min
image_classification	0.0127	0.0162	273,600	44.43 min
text_to_speech	0.0388	0.0266	301,056	22.46 min

Figure 4.5: Training value loss of three agents during the training.

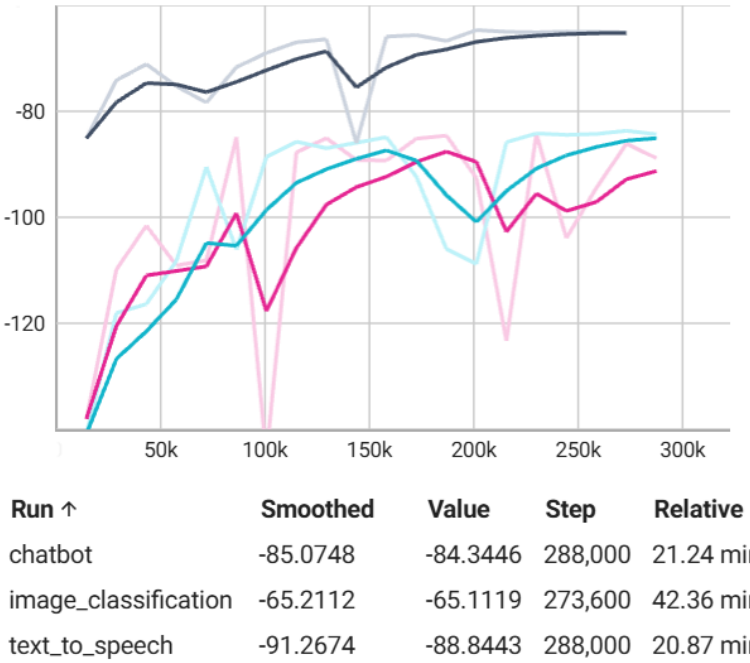


Figure 4.6: Validation mean reward of three agents during the training.

Table 4.7: Search ranges for parameters optimization for rule-based algorithm v1. Optuna allows dynamic search range.

Parameter	**Range Start**	**Range End**	**Step Size**
`inst_util_high_1`	0.6	0.8	0.01
`inst_util_high_2`	`inst_util_high_1`	0.95	0.01
`inst_util_low_2`	0.05	0.3	0.01
`inst_util_low_1`	`inst_util_low_2`	0.5	0.01

Table 4.8: Search ranges for parameters optimization for rule-based algorithm v2

Parameter	**Range Start**	**Range End**	**Step Size**
`sla_high`	0.005	0.1	0.002
`sla_low`	0.0001	0.001	0.0001
`inst_util_high`	0.6	0.9	0.02
`inst_util_low`	0.2	0.5	0.02

Table 4.9: Parameter-optimized values for rule-based v1 agent for each service type

Parameter	Chatbot	Image Classification	Text to Speech
`inst_util_high_2`	0.82	0.93	0.85
`inst_util_high_1`	0.8	0.8	0.8
`inst_util_low_1`	0.44	0.47	0.31
`inst_util_low_2`	0.2	0.17	0.14

Table 4.10: Parameter-optimized values for rule-based v2 agent for each service type

Parameter	Chatbot	Image Classification	Text to Speech
`sla_high`	0.081	0.095	0.091
`sla_low`	0.0007	0.0008	0.001
`inst_util_high`	0.9	0.9	0.89
`inst_util_low`	0.32	0.38	0.22

Table 4.11: Comparison of rule-based algorithm v1 and rule-based algorithm v2 on the validation Set

Category	**Metric**	**rule-based v1**	**rule-based v2**
`chatbot`	SLO Violation Rate	0.0166	**0.0150**
	Number of Instances	1.8875	**1.8760**
	Reward	-0.03849	**-0.03695**
`image classification`	SLO Violation Rate	**0.0182**	**0.0182**
	Number of Instances	1.4542	**1.4309**
	Reward	-0.03452	**-0.03423**
`text to speech`	SLO Violation Rate	0.0154	**0.0144**
	Number of Instances	**1.8722**	1.9007
	Reward	-0.03723	**-0.03675**

V. Evaluation

In this chapter, we evaluate and compare the performance of the DRL agent and rule-based agent using the DRAFAS simulator in different environment setups. We consider three performance metrics: average SLO violation rate, average number of instances (resource usage), and most importantly, the reward function, which is the metric the agent tries to optimize that balances the SLO violation rate and resource usage. We also evaluated the DRL agent and the rule-based agent in the real DRAFAS environment. Finally, we evaluate the benefit of network slicing for the AI-native services, especially when there is competing traffic. Note that because we chose the rule-based v2 algorithm for DRAFAS due to its superior performance compared to the rule-based v1 algorithm, from now on, when we mention the rule-based algorithm, it means the rule-based v2 algorithm.

5.1 Evaluation with the simulator

Figure 5.1 shows how the client simulator, service simulator, and DRL agent / rule-based agent interact with each other to form a setup for evaluation. The setup is similar to the training phase, except that in the evaluation, the client simulator for each service uses its own test set and all three services are evaluated together, possibly competing for available resources. In terms of the test set, we have one day of trace (86400 data points). To create a more diverse request trace for each service, we shifted the trace by 0 hours, 8 hours, and 16 hours for chatbot, image classification, and text to speech clients, respectively.

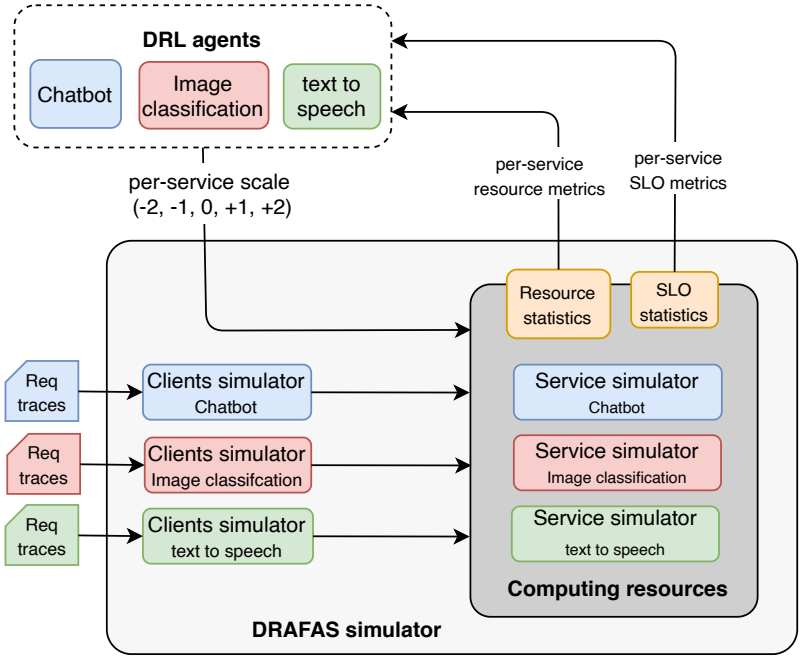


Figure 5.1: DRAFAS client and service simulator interacting with multiple services and DRL agents in evaluation.

5.1.1 Performance when using same setup as training

In this test case, we use the same setup as the training with a total number of resources set to 8 and the request scaling ratio set to 0.15, 0.4, and 0.15 for chatbot, image classification, and text to speech clients, respectively.

Figure 5.2 shows the reward achieved by DRL agent and rule-based algorithm for each AI-native service. In all cases, DRL achieved better reward value compared to the rule-based algorithm. Specifically, the reward achieved by the DRL agent is better than the rule-based agent by 9.22%, 7.61%, and 0.22% for chatbot, image classification, and text to speech services, respectively. The detailed metrics value for the average SLO violation rate and average number of instances is presented in Table. 5.1. Generally, we can see that there are tradeoffs between the SLO violation rate and the average

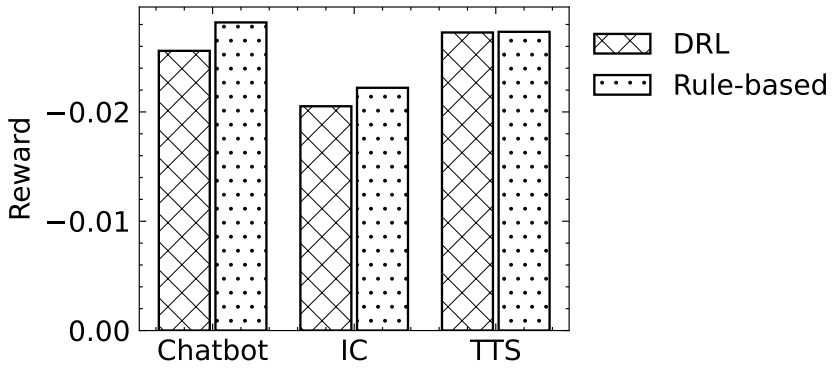


Figure 5.2: Reward comparison between DRL algorithm and rule-based algorithm in simulation. The total resource unit is set to 8 and request scaling ratio set to 0.15, 0.4, 0.15 for chatbot, image classification (IC), and text to speech (TTS) clients, respectively.

number of instances, as shown in the case of image classification service and text to speech service. In the case of the chatbot, the DRL agent performed better than the rule-based agent in both SLO violation rate and number of instances. The difference in processing time is negligible.

Use Case	Metric	DRL	Rule-based
Chatbot	SLO Violation Rate	**0.0024** (**29.41%**)	0.0034
	Num Instances	**1.8771** (**6.63%**)	2.0104
	Processing Time	0.2956	**0.2955** (**0.03%**)
	Reward	**-0.0256** (**9.22%**)	-0.0282
Image Classification	SLO Violation Rate	0.0018	**0.0017** (**5.56%**)
	Num Instances	**1.5115** (**8.60%**)	1.6538
	Processing Time	0.0707	0.0707
	Reward	**-0.0205** (**7.61%**)	-0.0222
Text to Speech	SLO Violation Rate	0.0064	**0.0029** (**54.69%**)
	Num Instances	**1.7219** (**12.88%**)	1.9764
	Processing Time	0.2735	**0.2672** (**2.30%**)
	Reward	-0.0273	-0.0273

Table 5.1: Detailed comparison between DRL algorithm and rule-based algorithm in simulation. The total resource unit is set to 8 and request scaling ratio set to 0.15, 0.4, 0.15 for chatbot, image classification, and text to speech clients, respectively.

5.1.2 Performance when changing request ratio

We evaluate and compare the performance of the DRL agent and the rule-based agent when the request scaling ratio changed significantly, i.e., while the relative pattern of requests per second throughout the day is kept the same, the number of requests per second changed significantly. We changed the request scaling ratio for chatbot, image classification, and text to speech clients to 0.3, 0.8, and 0.3, respectively, which is doubled compared to the previous test case.

The reward results are shown in Figure 5.3. In this test case, the DRL agent improved the reward score significantly compared to the rule-based agent. Specifically,

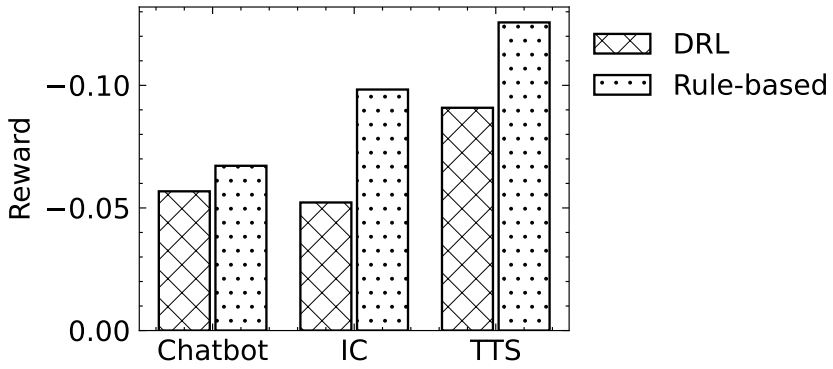


Figure 5.3: Reward comparison between DRL algorithm and rule-based algorithm in simulation. The total resource unit is set to 8 and request scaling ratio set to 0.8, 0.3, 0.3 for chatbot, image classification (IC), and text to speech (TTS) clients, respectively.

when compared to the rule-based agent, the DRL agent improved reward score by 15.49%, 46.86%, and 27.7% for chatbot, image classification, and text to speech service, respectively. The amount of improvement is also significantly higher than in the previous test. When looking into the detailed comparison presented in Table 5.2, we can see that the DRL agent achieved better metric value in almost all categories over all three AI services, except for the number of instance metrics in the image classification service.

Use Case	Metric	DRL	Rule-based
Chatbot	SLO Violation Rate	**0.0215** (**32.39%**)	0.0318
	Num Instances	**2.9951** (**2.84%**)	3.0826
	Processing Time	**0.3645** (**3.26%**)	0.3768
	Reward	**-0.0568** (**15.49%**)	-0.0672
Image Classification	SLO Violation Rate	**0.0259** (**67.75%**)	0.0803
	Num Instances	2.3139	**2.0830** (**9.98%**)
	Processing Time	**0.0811** (**21.19%**)	0.1029
	Reward	**-0.0522** (**46.86%**)	-0.0983
Text to Speech	SLO Violation Rate	**0.0659** (**34.49%**)	0.1006
	Num Instances	**2.5198** (**10.34%**)	2.8104
	Processing Time	**0.3304** (**5.90%**)	0.3511
	Reward	**-0.0909** (**27.70%**)	-0.1257

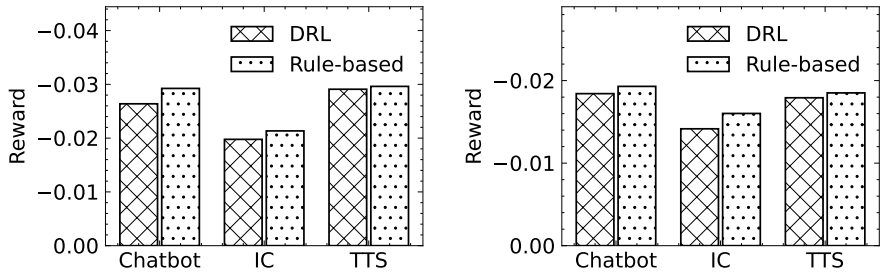
Table 5.2: Detailed comparison between DRL algorithm and rule-based algorithm in simulation. The total resource unit is set to 8 and request scaling ratio set to 0.3, 0.8, 0.3 for chatbot, image classification, and text to speech clients, respectively.

A possible reason for the significant differences between the DRL agent and and rule-based agent in this test case is because of resource contention. The increased request rate makes services compete for computing resources more frequently. The rule-based algorithm does not have the mechanism to cope with the resource competition in a fair way, it only has action masking that blocks an invalid action and adjusts it into a valid action. The DRL agent, on the other hand, is trained with a random resource constraint mechanism, which helps the DRL to learn how to deal better with the environment with frequent resource competition. Another possible reason is that the parameter optimization method tried to optimize parameters for the validation set. However, the mechanism of the rule-based agent is too simple to be generalized well

to different environment setups.

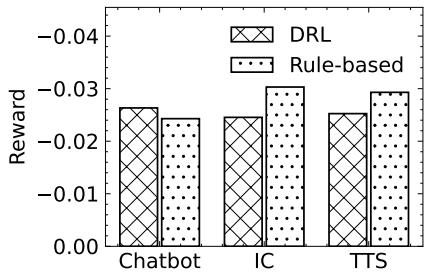
5.1.3 Performance when changing the total resource units

In this test, we changed the total number of resource units to 16, 32, 64, and 128. For the case of 16 and 32 resource units, we set the scaling ratio to 0.3, 0.8, and 0.3, for chatbot, image classification, and text to speech clients, respectively. For the case of 64 resource units, we set the equivalent scaling ratios to 1, 3, and 1; and for the case of 128 resource units, we set the equivalent scaling ratios to 2, 6, and 2.

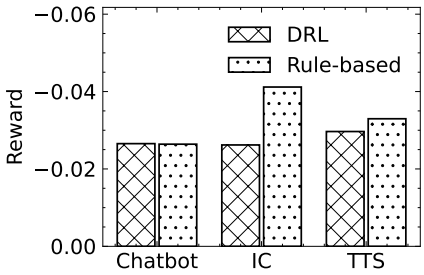


(a) Result with 16 total resource units

(b) Result with 32 total resource units



(c) Result with 64 total resource units



(d) Result with 128 total resource units

Figure 5.4: Reward comparison between DRL algorithm and rule-based algorithm in simulation. The total resource unit is set to 16, 32, 64, and 128. IC stands for image classification, TTS stands for text to speech.

The reward scores comparison between DRL and rule-based agents in these four test cases are shown in Figure 5.4. In most cases, the DRL agent achieved a better

reward score compared to the rule-based agent, except for the case of chatbot with 64 total resource units, where the DRL agent performed worse than the rule-based algorithm; and the case of chatbot with 128 total resource units, where the DRL agent performed similarly to the rule-based algorithm.

Use Case	**Metric**	**DRL**	**Rule-based**
Chatbot	SLO Violation Rate	**0.0040 (52.94%)**	0.0085
	Num Instances	29.3646	**23.9944 (18.29%)**
	Processing Time	**0.2985 (6.19%)**	0.3182
	Reward	-0.0265	**-0.0264 (0.38%)**
Image Classification	SLO Violation Rate	**0.0009 (97.29%)**	0.0332
	Num Instances	32.4649	**14.4868 (55.38%)**
	Processing Time	**0.0719 (29.65%)**	0.1022
	Reward	**-0.0262 (36.36%)**	-0.0412
Text to Speech	SLO Violation Rate	**0.0081 (39.55%)**	0.0134
	Num Instances	28.6958	**26.8042 (6.59%)**
	Processing Time	**0.2736 (2.81%)**	0.2815
	Reward	**-0.0297 (10.01%)**	-0.0330

Table 5.3: Detailed comparison between DRL algorithm and rule-based algorithm in simulation. The total resource unit is set to 128 to simulate large cluster. Request scaling ratios are set to 2, 6, 2 for chatbot, image classification, and text to speech clients, respectively, to simulate high workload.

Table 5.3 shows the detailed results for the case of 128 total resource units and request scaling ratios set to 2, 6, 2 for chatbot, image classification, and text to speech clients, respectively. In large clusters with high workloads like this, the DRL agent achieved significantly better reward scores in the case of image classification service and text to speech services, with 36.36% and 10.01% improvement over the rule-based

agent, respectively. For the chatbot service, the reward is slightly lower than the rule-based agent (0.53%), which is negligible. We can also see the trade-off between keeping low SLO violation rate and low resource usage.

5.2 Evaluation on the real testbed

We deployed three AI services: chatbot, image classification, and text to speech into our real DRAFAS environment. The DRAFAS cluster is the same as the cluster used in profiling: we have two servers, and each server has one Intel Xeon 4114 CPU with 10 cores, 64GB of RAM, and one Nvidia RTX 5000 GPU with 16 GB of RAM. We use Multi-process Service to spatially partition each GPU into four equal slices, effectively creating 8 vGPUs in the cluster, which is equivalent to 8 resource units.

Each service started with one replica. We deployed three DRL agents using the models trained from the training step. The DRL agents run on CPU in a separate machine. We also deployed three client emulators to send inference requests to the AI services. Each service sends requests following its own test set request trace, and the request content is selected randomly from per-service content dataset, as described in Section 4.4.1. The request scaling ratios for chatbot, image classification, and text to speech services are set to 0.15, 0.4, and 0.15, respectively.

Figure 5.5 shows the reward comparison between the DRL agent and the rule-based agent for AI services deployed in the real environment. For the chatbot service and image classification service, DRL achieved better reward results compared to the rule-based agent at 3.21% and 31.81%, respectively. For the text to speech service, DRL also achieved better reward than the rule-based algorithm, however, the difference is negligible. Table 5.4 shows the detailed results. We can see the trade-off between the SLO violation rate and the number of instances in the case of chatbot and text to speech services. In the case of image classification, DRL improved both the SLO violation rate and resource usage compared to the rule-based agent.

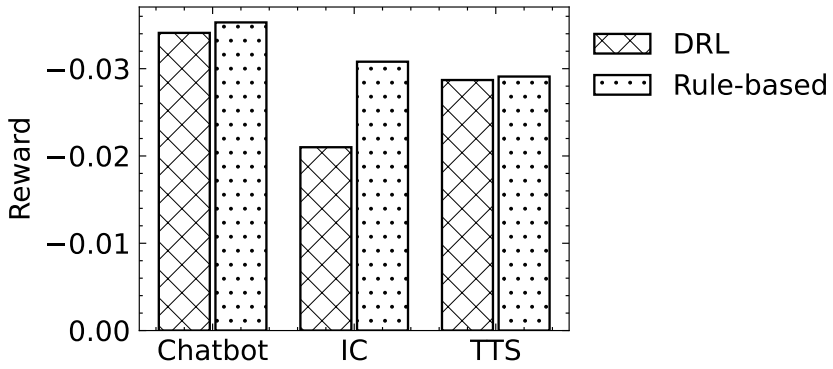


Figure 5.5: Reward comparison between DRL algorithm and rule-based algorithm in the real testbed.

Use Case	**Metric**	**DRL**	**Rule-based**
Chatbot	SLO Violation Rate	0.0067	**0.0036** (**46.27%**)
	Num Instances	**2.2484** (**12.32%**)	2.5642
	Processing Time	0.4625	**0.3994** (**13.64%**)
	Reward	**-0.0341** (**3.21%**)	-0.0353
Image Classification	SLO Violation Rate	**0.0026** (**75.78%**)	0.0106
	Num Instances	**1.4946** (**12.21%**)	1.7024
	Processing Time	**0.0874** (**17.13%**)	0.1023
	Reward	**-0.0210** (**31.81%**)	-0.0308
Text To Speech	SLO Violation Rate	0.0034	**0.0012** (**64.7%**)
	Num Instances	**2.0472** (**8.80%**)	2.2448
	Processing Time	0.3752	**0.3397** (**9.46%**)
	Reward	**-0.0287** (**1.63%**)	-0.0291

Table 5.4: Detailed comparison between DRL algorithm and rule-based algorithm in the real testbed.

5.3 Effect of network slicing

In the real environment, the inference request traffic might need to compete with other types of traffic such as video streaming, which might take a lot of network resources and cause excessive delay or even packet loss. For critical AI services, DRAFAS provides optional network slicing to create an end-to-end network slice from clients to the cluster where the AI service is deployed. In this test, we created a such a network slice for all three AI network services with the reserved bandwidth of 20 Mbps. We cap the total available bandwidth to 100 Mbps, and let the other non-AI services use another network slice with the remaining bandwidth.

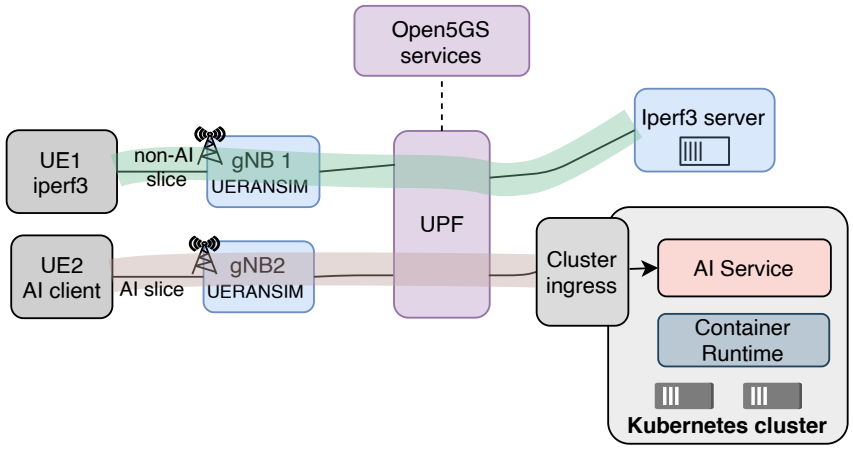
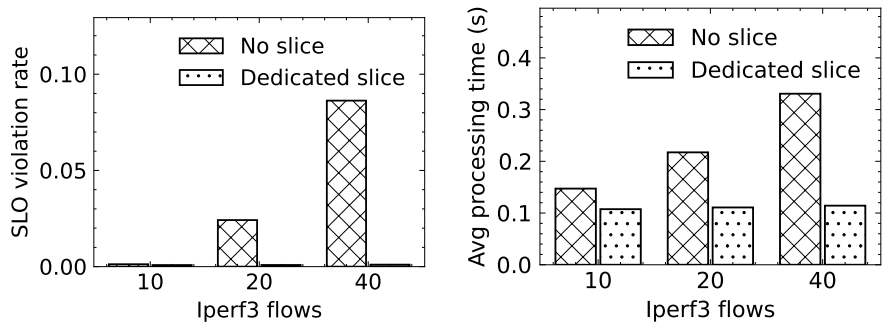


Figure 5.6: Evaluation setup with network slicing.

We test each AI service independently by deploying each service into the real testbed with 6 replicas. The computing cluster is the same as presented in Section 5.2. The client emulator is ran inside an emulated UE using UERANSIM [15], sending inference requests at a constant rate of 10, 30, and 10 requests per second for chatbot, image classification, and text to speech services, respectively. We emulated traffic from other non-AI services by running multiple iperf3 [80] flows between an emulated UE and a server outside the DRAFAS cluster. The number of iperf3 flows is changed

between 10, 20, and 40 flows. We measured the average processing time and SLO violation rate from the client emulator. Figure 5.6 shows the evaluation setup with network slicing.

For the image classification service, there are significant improvements in terms of SLO violation and request processing time when using network slicing compared to no network slicing. Figure 5.7 shows the SLO violation rate and average processing time of image classification service with and without dedicated network slicing. When the number of iperf3 flows increased, the SLO violation rate and average processing time significantly increased in the case there is no dedicated network slice for the image classification service. When there is a dedicated network slice for the image classification services, the SLO violation rate and average processing time are almost not affected by the iperf3 flows.



(a) SLO violation rate

(b) Average processing time

Figure 5.7: SLO violation rate and average processing time of image classification service with and without dedicated network slice.

For the text to speech service, there is a small difference when the number of iperf3 flows is high (40 flows) as shown in Figure 5.8. In terms of the SLO violation rate, there is no difference between having no slice and having a dedicated slice. For the chatbot service, there is no difference in both the SLO violation rate and request processing time in both cases.

Because the image classification service requires more significant bandwidth com-

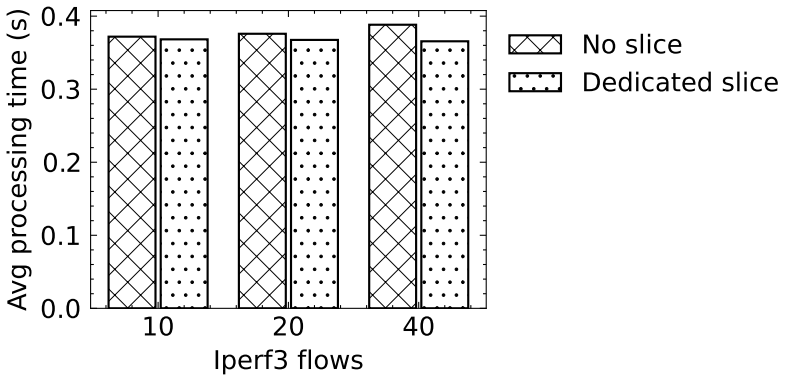


Figure 5.8: Average processing time of text to speech service with and without dedicated network slice.

pared to chatbot and text to speech services, the image classification service is more affected by competing traffic flows. Additionally, The SLO delay requirement of the image classification service is much lower than the other two services, it is more sensitive to additional delay caused by competing traffic.

VI. Conclusion

In this chapter, we summarize the content of this thesis and highlight the contribution. We also discussed the limitations of the current work and directions for future work.

6.1 Summary

The rapid advancements in Artificial Intelligence (AI) and Machine Learning (ML) have led to the widespread adoption of AI-native services across various domains of human life. With the growing demand for AI-native services, there is a need for efficient resource allocation frameworks to optimize GPU utilization, reduce infrastructure costs, and maintain Service Level Objectives (SLOs) within acceptable values. The framework should also provide multi-tenant support, i.e., the framework should provide performance and security isolation between AI-native services.

To address these issues, we propose DRAFAS: a Dynamic Resource Allocation Framework for AI-native Services with multi-tenant support. DRAFAS ensures both performance and security isolation by containerizing services and managing them using a container orchestration framework. DRAFAS allocates fractional GPU to service containers using Multi-process Service mechanism.

We implemented DRAFAS simulator, including the AI service simulator and client simulator. To make the simulator close to the real environment, the service can be deployed in a real DRAFAS testbed for performance profiling. DRAFAS simulator helps accelerate the development and evaluation of new algorithms for resource allocation of AI-native services.

In DRAFAS, we designed and implemented two resource allocation algorithms: a parameter-optimized rule-based algorithm, and a Deep Reinforcement Learning (DRL)-

based algorithm. We train and optimize the algorithms using the DRAFAS simulator following real-world ML inference traces. The evaluation results on the simulator showed that the DRL-based resource allocation algorithm performed better than the rule-based algorithm in most cases. The DRL-based agent maintains lower resource usage while still keeping the SLO violation rate low. The DRL-based agent also works better in network setups that are not presented in the training, especially when there are tight resource constraints.

We implemented DRAFAS framework and three example AI-native services, and deployed them in a GPU cluster for evaluation. The evaluation on the real environment shows that the DRL agent trained with the simulator worked well in the real testbed. The DRL agent outperformed the rule-based algorithm in both the simulation and real testbed.

We also implemented optional network slicing in DRAFAS to reserve bandwidth for AI-native services, effectively creating end-to-end network slice from the clients to the services. The evaluation results showed that network slicing effectively maintains SLOs in the presence of competing traffic, especially for services with high bandwidth demands.

6.2 Future work

While DRAFAS shows promising evaluation results, there are several limitations and areas that can be improved.

6.2.1 Dynamic resource allocation for network resources

In DRAFAS, we only do dynamic resource allocation for the computing resources, which is the most important resource for AI-native services. For the network resources, DRAFAS allows configuring a fixed bandwidth for the network slice. However, because the number of inference requests changed over time, it is more efficient to adjust the network resources allocated to the slice according to the change of the inference

requests. Existing work [51] has been tried to apply DRL to dynamically adjust network resources for slicing. DRAFAS can implement a similar approach for dynamic network resource allocation.

6.2.2 Continue fine-tuning DRL agent in real environment

Despite effort to make the simulator behave close to the real environment, there will be always a gap between simulation and and real deployment. The trained DRL agent can be further fine-tuned in the real environment for a better performance. This is another advantage of the DRL agent compared to the rule-based agent: there is no clear approach to further fine-tuning the rule-agent agent on the real environment.

6.2.3 CPU-based AI-native services

While it is usually best to deploy an AI service on GPU, GPU resources are expensive and might not be always available. In such cases, the AI service or some replica of the AI service can be run in the available CPU cores. This would improve the number of requests that can be served and improve SLOs.

6.2.4 Support multiple inference request mechanisms

In DRAFAS, we assumed that the AI-native services use HTTP or gRPC as the protocol to send inference requests and receive results. While these are the two most popular protocols for AI services, there can be other protocols used, such as plain TCP stream. In these cases, the service might need to be modified to expose necessary metrics to DRAFAS so that the resource allocation agent can work correctly.

6.2.5 CPU-based AI-native services

Nowadays, the Muti-Access Edge Computing (MEC) architecture can be applied to move computing resources closer to the clients, reducing latency for critical AI

services. DRAFAS can be improved by placing newly created containers close to the clients for such AI services.

References

[1] Precedence Research. Artificial Intelligence-as-a-service Market. https://www.precedenceresearch.com/artificial-intelligence-as-a-service-market. Accessed on 2024-12-10.

[2] Nvidia. Multi-Instance GPU. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html. Accessed on 2024-12-10.

[3] HUAWEI. What Is Network Slicing. https://info.support.huawei.com/info-finder/encyclopedia/en/Network+Slicing.html. Accessed on 2024-12-10.

[4] Siyi Hu, Yifan Zhong, Minquan Gao, Weixun Wang, Hao Dong, Xiaodan Liang, Zhihui Li, Xiaojun Chang, and Yaodong Yang. Marllib: A scalable and efficient multi-agent reinforcement learning library. *Journal of Machine Learning Research*, 2023.

[5] Gartner. Forecast Analysis: Artificial Intelligence Software, 2023-2027, Worldwide. https://www.gartner.com/en/documents/4925331. Accessed on 2024-12-10.

[6] Wen Wu, Conghao Zhou, Mushu Li, Huaqing Wu, Haibo Zhou, Ning Zhang, Xuemin Sherman Shen, and Weihua Zhuang. Ai-native network slicing for 6g networks. *IEEE Wireless Communications*, 29(1):96–103, 2022.

[7] Stefanos Georgiou, Maria Kechagia, Tushar Sharma, Federica Sarro, and Ying Zou. Green ai: do deep learning frameworks have different costs? In *Proceedings*

of the 44th International Conference on Software Engineering, ICSE '22, page 1082–1094, New York, NY, USA, 2022. Association for Computing Machinery.

[8] Eva García-Martín, Crefeda Faviola Rodrigues, Graham Riley, and Håkan Grahn. Estimation of energy consumption in machine learning. *Journal of Parallel and Distributed Computing*, 134:75–88, 2019.

[9] Microsoft azure: Cloud computing services. `https://azure.microsoft.com/`. Accessed: 2024-12-10.

[10] Amazon web services (aws): Cloud computing services. `https://aws.amazon.com/`. Accessed: 2024-12-10.

[11] Google cloud platform: Cloud computing services. `https://cloud.google.com/`. Accessed: 2024-12-10.

[12] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing llm inference clusters for performance and energy efficiency, 2024.

[13] Kubernetes: Production-grade container orchestration. `https://kubernetes.io/`, 2024. Accessed: 2024-12-10.

[14] Open5gs: Open source 5g core network. `https://open5gs.org/`. Accessed: 2024-12-10.

[15] Ueransim: Open source 5g ue and ran (gnodeb) implementation. `https://github.com/aligungr/UERANSIM`. Accessed: 2024-12-10.

[16] Nguyen Van Tu. Drafas: Dynamic resource allocation for ai-native services. `https://github.com/tu-nv/drafas`. Accessed: 2024-12-10.

[17] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep learning applications. *Proceedings of Machine Learning and Systems*, 2:98–111, 2020.

[18] Zhisheng Ye, Wei Gao, Qinghao Hu, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. Deep learning workload scheduling in gpu datacenters: A survey. *ACM Comput. Surv.*, 56(6), jan 2024.

[19] Nvidia. Improving GPU utilization in Kubernetes. `https://developer.nvidia.com/blog/improving-gpu-utilization-in-kubernetes`. Accessed on 2024-12-10.

[20] Nvidia. Multi-Process Service. `https://docs.nvidia.com/deploy/mps/index.html`. Accessed on 2024-12-10.

[21] Xenofon Foukas, Georgios Patounas, Ahmed Elmokashfi, and Mahesh K. Marina. Network slicing in 5g: Survey and challenges. *IEEE Communications Magazine*, 55(5):94–100, 2017.

[22] Adrian Farrel, John Drake, Reza Rokui, Shunsuke Homma, Kiran Makhijani, Luis M. Contreras, and Jeff Tantsura. A Framework for Network Slices in Networks Built from IETF Technologies. RFC 9543, March 2024. Accessed on 2024-12-10.

[23] Li-Hsiang Shen, Kai-Ten Feng, and Lajos Hanzo. Five facets of 6g: Research challenges and opportunities. *ACM Comput. Surv.*, 55(11), feb 2023.

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

[25] Xiaorui Wu, Hong Xu, and Yi Wang. Irina: Accelerating dnn inference with efficient online scheduling. In *4th Asia-Pacific Workshop on Networking*, APNet '20, page 36–43, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the*

11th ACM Symposium on Cloud Computing, SoCC '20, page 492–506, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association.

[28] Junguk Cho, Diman Zad Tootaghaj, Lianjie Cao, and Puneet Sharma. Sla-driven ml inference framework for clouds with heterogeneous accelerators. *Proceedings of Machine Learning and Systems*, 4:20–32, 2022.

[29] NVIDIA. Triton Inference Server. `https://github.com/triton-inference-server/server`. Accessed on 2024-12-10.

[30] Yunteng Luan, Xukun Chen, Hanyu Zhao, Zhi Yang, and Yafei Dai. Sched²: Scheduling deep learning training via deep reinforcement learning. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7, 2019.

[31] Haoyu Wang, Zetian Liu, and Haiying Shen. Job scheduling for large-scale machine learning clusters. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 108–120, New York, NY, USA, 2020. Association for Computing Machinery.

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

[33] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, 2015.

[34] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous

methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[35] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *CoRR*, abs/1707.06347, 2017.

[36] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pages 5–32, 1992.

[37] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[38] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[39] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

[40] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay, 2017.

[41] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[42] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[43] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms. *CoRR*, abs/2006.14171, 2020.

[44] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. *Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms*, pages 321–384. Springer International Publishing, Cham, 2021.

[45] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

[46] Mathieu Leconte, Georgios S. Paschos, Panayotis Mertikopoulos, and Ulaş C. Kozat. A resource allocation framework for network slicing. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 2177–2185, 2018.

[47] Linh Le, Tu N. Nguyen, Kun Suo, and Jing (Selena) He. 5g network slicing and drone-assisted applications: a deep reinforcement learning approach. In *Proceedings of the 5th International ACM Mobicom Workshop on Drone Assisted Wireless Communications for 5G and Beyond*, DroneCom '22, page 109–114, New York, NY, USA, 2022. Association for Computing Machinery.

[48] Taihui Li, Xiaorong Zhu, and Xu Liu. An end-to-end network slicing algorithm based on deep q-learning for 5g network. *IEEE Access*, 8:122229–122240, 2020.

[49] Qiang Liu, Nakjung Choi, and Tao Han. Atlas: automate online service configuration in network slicing. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '22, page 140–155, New York, NY, USA, 2022. Association for Computing Machinery.

[50] Salvatore D'Oro, Leonardo Bonati, Francesco Restuccia, Michele Polese, Michele Zorzi, and Tommaso Melodia. Sl-edge: network slicing at the edge. In *Proceedings of the Twenty-First International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Com-*

puting, Mobihoc '20, page 1–10, New York, NY, USA, 2020. Association for Computing Machinery.

[51] Qiang Liu, Nakjung Choi, and Tao Han. Onslicing: online end-to-end network slicing with reinforcement learning. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, page 141–153, New York, NY, USA, 2021. Association for Computing Machinery.

[52] Hamid Arabnejad, Claus Pahl, Pooyan Jamshidi, and Giovani Estrada. A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 64–73, 2017.

[53] Openstack: Open source cloud computing infrastructure. `https://www.openstack.org/`. Accessed: 2024-12-10.

[54] Kaustabha Ray and Ansuman Banerjee. Horizontal auto-scaling for multi-access edge computing using safe reinforcement learning. *ACM Trans. Embed. Comput. Syst.*, 20(6), October 2021.

[55] 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, 2020.

[56] Paola Soto, Danny De Vleeschauwer, Miguel Camelo, Yorick De Bock, Koen De Schepper, Chia-Yu Chang, Peter Hellinckx, Juan F. Botero, and Steven Latré. Towards autonomous vnf auto-scaling using deep reinforcement learning. In *2021 Eighth International Conference on Software Defined Systems (SDS)*, pages 01–08, 2021.

[57] Petra Gabriela, Doyoung Lee, Nguyen Van Tu, and James Won-Ki Hong. Machine learning-based auto-scaler for video conferencing systems. In *2021 IEEE*

7th International Conference on Network Softwarization (NetSoft), pages 142–150, 2021.

[58] Zhiguang Wang, Chul Gwon, Tim Oates, and Adam Iezzi. Automated cloud provisioning on aws using deep reinforcement learning, 2017.

[59] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[60] Zhaoyu Gan, Rongheng Lin, and Hua Zou. Adaptive auto-scaling in mobile edge computing: A deep reinforcement learning approach. In *2022 2nd International Conference on Consumer Electronics and Computer Engineering (ICCECE)*, pages 586–591, 2022.

[61] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[62] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

[63] Simpy: Discrete event simulation for python. `https://gitlab.com/team-simpy/simpy`. Accessed: 2024-12-10.

[64] Traefik: The cloud-native application proxy. `https://traefik.io/`. Accessed: 2024-12-10.

[65] K3s: Lightweight kubernetes distribution. `https://k3s.io/`. Accessed: 2024-12-10.

[66] Nvidia container toolkit: Build and run containers leveraging nvidia gpus. https://github.com/NVIDIA/nvidia-container-toolkit. Accessed: 2024-12-10.

[67] Envoy proxy: Cloud-native high-performance edge/middle/service proxy. https://www.envoyproxy.io/. Accessed: 2024-12-10.

[68] The istio service mesh. https://istio.io/. Accessed: 2024-12-10.

[69] kube-proxy: The kubernetes network proxy. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/. Accessed: 2024-12-10.

[70] Prometheus: Monitoring system & time series database. https://prometheus.io/. Accessed: 2024-12-10.

[71] asyncio: Asynchronous i/o framework. https://docs.python.org/3/library/asyncio.html. Accessed: 2024-12-10.

[72] Ollama: Get up and running with large language models. https://ollama.com/. Accessed: 2024-12-10.

[73] Llama: Open-source ai models. https://ai.meta.com/. Accessed: 2024-12-10.

[74] . Pytorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://github.com/pytorch/pytorch. Accessed on 2024-12-10.

[75] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[76] Coqui tts: Open-source text-to-speech framework. https://coqui.ai/. Accessed: 2024-12-10.

[77] Alex Krizhevsky. Learning multiple layers of features from tiny images. *Technical Report, University of Toronto*, 2009.

[78] Chatgpt: Language model for conversational ai. `https://chatgpt.com`. Accessed: 2024-12-10.

[79] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *The 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631, 2019.

[80] Iperf3: A tcp, udp, and sctp network bandwidth measurement tool. `https://iperf.fr/`. Accessed: 2024-12-10.

Acknowledgements

It has been a long journey, finally comes to an end.

I would like to thank Prof. James Won-Ki Hong for being my supervisor during my Master and Ph.D. at POSTECH. His guidance helped me a lot in improving my skills and knowledge. His patient with me helped me through the difficult time while doing research. I also want to thank him to give opportunities to go to many conferences in many countries.

I want to say thank to all of my labmates at DPNM. We have been together for so long, doing project together, enjoying Ph.D. life together. We had so much memories. Many have been graduated, few will graduate in a few years. I wish all of you a happy life.

I want to say thank to all my friends at POSTECH. We have been together for so long, enjoying beautiful life in Pohang together, encourage each other though difficult times. I will remember. I wish you all the best.

I wish to thank my mother and my father. They have raised me, taught me, and loved me. I wish to thank my wife and my son, who has been with me in every steps of my Ph.D. journey. I love them more than I can say. This thesis is dedicated to them.

Curriculum Vitae

Personal Information

Name : **Nguyen Van Tu**

Position : Ph.D. student

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

Department : Computer Science and Engineering

Education

2021. 03. – 2025. 02. Department of Computer Science and Engineering, Pohang University of Science and Technology (Ph.D.)

2016. 09. – 2018. 08. Department of Computer Science and Engineering, Pohang University of Science and Technology (M.S.)

2010. 09. – 2015. 08. School of Electronic and Telecommunication, Hanoi University of Science and Technology (B.Sc.)

Research Areas of Interest

Network Function Virtualization (NFV), Cloud Computing, Video conferencing, Applied-AI in Networking

Research / Project Experiences

1. **Development of core technologies for programmable switches in multi-service networks**

   Funded by the Institute for Information & Communications Technology Planning & evaluation (IITP) (2018 – 2020)

   This project aims to support multi-service networks based on programmable switches. The research goals focus on extending P4 languages and defining a new switch machine model, compiler, and multi-service network architecture. My contribution to this project is developing In-band Network Telemetry support for ONOS controller, and develop INTCollector - a high performance collector for In-band Network Telemetry.

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

   Funded by Institute for Information & Communications Technology Planning & evaluation (IITP) (2018 – 2023)

   This research project aims to study virtual network management, including the development of AI-based algorithms essential for VNF life-cycle management to monitor and control resources in the NFV environment in real-time. My contribution to this project involved building an OpenStack-based NFV framework and APIs with monitoring and orchestrating functions to support AI-based management modules. I developed a method for fine-grained monitoring of packet processing time in VNFs and SFCs. I also developed an architecture for building high-performance VNFs using eBPF and XDP.

3. **Development of integrated intelligent plane technology for 6G networks**

Funded by Institute for Information & Communications Technology Planning & evaluation (2024 – 2025)

This research project aims to develop intelligent plane technology that provides AI-based optimal inference and control policies through efficient end-to-end data collection in a 6G integrated domain (RAN/Transport/Core/Data/OAM) mobile network. My contribution to this project is proposing and developing DRAFAS: a Dynamic Resource Allocation framework for AI-native Services. DRAFAS use a DRL-based algorithm to dynamically scaling AI-native services to optimize both SLO requirements and resources usage. I implemented a simulator to accelerate DRL training and validation. I also implemented three AI-native services and DRAFAS testbed for real-world evaluation.

4. **Development of LLM-Based Network Configuration Management Automation Technology**

   Funded by Samsung Electronics (2024 - 2025)

   This research project aims to develop automation technology for network configuration and management using Large Language Models (LLMs). My contribution to this project is proposing and implementing a general and effective approach to perform the network intent translation task using large language models with fine-tuning, dynamic in-context learning, and continuous learning.

International Journal Papers

1. **Van Tu, Nguyen**; Ryu, Sangwoo; Ko, Kyungchan; Yoo, Jae-Hyoung; Hong, James Won-Ki; Muno: Improved Bandwidth Estimation Scheme in Video Conferencing using Deep Reinforcement Learning; International Journal of Network Management (IJNM, SCIE) (accepted for publication).

2. **Van Tu, Nguyen**; Yoo, Jae-Hyoung; Hong, James Won-Ki; Intent-based Network Configuration using Large Language Models; International Journal of Network Management (IJNM, SCIE), vol. 35, Jan 2025, e2313.

3. **Van Tu, Nguyen**; Yoo, Jae-Hyoung; Hong, James Won-Ki; PPTMon: Real-time and fine-grained packet processing time monitoring in virtual network functions; IEEE Transactions on Network and Service Management (TNSM, SCIE), vol. 18, Dec 2021, pp. 4324-4336.

4. **Van Tu, Nguyen**; Yoo, Jae-Hyoung; Hong, James Won-Ki; Accelerating virtual network functions with fast-slow path architecture using eXpress Data Path; IEEE Transactions on Network and Service Management (TNSM, SCIE), vol. 17, Sep 2020, pp. 1474-1486.

5. Lange, Stanislav; **Van Tu, Nguyen**; Jeong, Se-Yeon; Lee, Do-Young; Kim, Hee-Gon; Hong, Jibum; Yoo, Jae-Hyoung; Hong, James Won-Ki; A network intelligence architecture for efficient VNF lifecycle management; IEEE Transactions on Network and Service Management (TNSM, SCIE), vol. 18, Jun 2021, pp. 1476-1490.

6. Khizar, Abbas; Hong, Jibum; **Van Tu, Nguyen**; Yoo, Jae-Hyoung; Hong, James Won-Ki; Autonomous DRL-based energy efficient VM consolidation for cloud data centers; Physical Communication, vol. 55, Dec 2022, 101925.

7. Hyun, Jonghwan; **Van Tu, Nguyen**; Yoo, Jae-Hyoung; Hong, James Won-Ki; Real-time and fine-grained network monitoring using in-band network telemetry; International Journal of Network Management (IJNM, SCIE), vol. 29, Dec 2019, e2080.

International Conference Papers

1. **Van Tu, Nguyen**; Yoo, Jae-Hyoung; Hong, James Won-Ki; Towards Intent-based Configuration for Network Function Virtualization using In-context Learning in Large Language Models, 2024-2024 IEEE/IFIP Network Operations and Management Symposium (NOMS 2024), Seoul, Korea, Republic of, 2024, pp. 1-8

2. **Van Tu, Nguyen**; Ko, Kyungchan; Ryu, Sangwoo; Ha, Sangtae; Hong, James Won-Ki; Improve Video Conferencing Quality with Deep Reinforcement Learning, 2023-2023 IEEE/IFIP Network Operations and Management Symposium (NOMS 2023), Miami, FL, USA, 2023, pp. 1-5

3. Sangwoo Ryu, Kyungchan Ko, **Nguyen Van Tu**, James Won-Ki Hong; Towards Effective Reinforcement Learning in Video Conferencing using Network Status Data and Model Analysis, 2024-2024 IEEE/IFIP Network Operations and Management Symposium (NOMS 2024), Seoul, Korea, Republic of, 2024, pp. 1-9

4. Kyungchan Ko, Sangwoo Ryu, **Nguyen Van Tu**, James Won-Ki Hong; Optimizing Video Conferencing QoS: A DRL-based Bitrate Allocation Framework, 2024 IEEE/IFIP Network Operations and Management Symposium (NOMS 2024), Seoul, Korea, Republic of, 2024, pp. 1-10

5. Gabriela, Petra; Lee, Doyoung; **Van Tu, Nguyen**; Hong, James Won-Ki; Machine learning-based auto-scaler for video conferencing systems, 2021 IEEE 7th International Conference on Network Softwarization (NetSoft 2021), Tokyo, Japan, 2021, pp. 142-150

6. Jeong, Seyeon; **Van Tu, Nguyen**; Yoo, Jae-Hyoung; Hong, James Won-Ki; Proactive live migration for virtual network functions using machine learning, 2021 17th International Conference on Network and Service Management (CNSM 2021), Izmir, Turkey, 2021, pp. 335-339

7. Pandey, Suman; **Van Tu, Nguyen**; Yoo, Jae-Hyoung; Hong, James Won-Ki; EdgeDQN: multiple SFC placement in edge computing environment, 2021 17th International Conference on Network and Service Management (CNSM 2021), Izmir, Turkey, 2021, pp. 301-309

8. **Van Tu, Nguyen**; Yoo, Jae-Hyoung; Hong, James Won-Ki; Measuring end-to-end packet processing time in service function chaining, 2020 16th International Conference on Network and Service Management (CNSM 2020), Izmir, Turkey, 2020, pp. 1-9

9. **Van Tu, Nguyen**; Yoo, Jae-Hyoung; Hong, James Won-Ki; Real-time Monitoring of Packet Processing Time for Virtual Network Functions, 2020 21st Asia-Pacific Network Operations and Management Symposium (APNOMS 2020), Daegu, Korea (South), 2020, pp. 138-143

10. **Van Tu, Nguyen**; Yoo, Jae-Hyoung; Hong, James Won-Ki; eVNF-Hybrid Virtual Network Functions with Linux eXpress Data Path, 2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS 2019), Matsue, Japan, 2019, pp. 1-6

11. **Van Tu, Nguyen**; Yoo, Jae-Hyoung; Hong, James Won-Ki; Building hybrid virtual network functions with eXpress Data Path, 2019 15th International Conference on Network and Service Management (CNSM 2019), Halifax, NS, Canada, 2019, pp. 1-9

12. **Van Tu, Nguyen**; Hyun, Jonghwan; Kim, Ga Yeon; Yoo, Jae-Hyoung; Hong, James Won-Ki; Intcollector: A high-performance collector for in-band network

telemetry, 2018 14th International Conference on Network and Service Management (CNSM 2018), Rome, Italy, 2018, pp. 10-18

13. Hyun, Jonghwan; **Van Tu, Nguyen**; Hong, James Won-Ki; Towards knowledge-defined networking using in-band network telemetry, 2018-2018 IEEE/IFIP Network Operations and Management Symposium (NOMS 2018), Taipei, Taiwan, 2018, pp. 1-7

14. **Van Tu, Nguyen**; Ko, Kyungchan; Hong, James Won-Ki; Architecture for building hybrid kernel-user space virtual network functions, 2017 4th International Workshop on Management of SDN and NFV Systems (ManSDN/NFV 2017), Tokyo, Japan, 2017, pp. 1-6

15. **Van Tu, Nguyen**; Hyun, Jonghwan; Hong, James Won-Ki; Towards onos-based sdn monitoring using in-band network telemetry, 2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS 2017), Seoul, Korea (South), 2017, pp. 76-81

16. **Van Tu, Nguyen**; Thien, Vu Tang; Kim, Son Nguyen; Ngoc, Nam Pham; Huu, Thanh Nguyen; A high throughput pipelined hardware architecture for tag sorting in packet fair queuing schedulers, 2015 International Conference on Communications, Management and Telecommunications (ComManTel 2015), Danang, Vietnam, 2015, pp. 41-45

17. Nguyen, Xuan-Nghia; **Nguyen, Van-Tu**; Pham, Ngoc-Nam; Le, Minh-Tuan; Tran, Xuan-Nam; Ngo, Vu-Duc; High throughput modified MMSE hardware detector for high-rate spatial modulation systems, 2016 IEEE Sixth International Conference on Communications and Electronics (ICCE 2016), Halong, Vietnam, 2016, pp. 211-216

Awards

Title	Organizations	Date
Student Trave Grant	NOMS 2023 (USA)	2023.05
POSCO Asia Fellowship	POSCO (Korea)	2016.09 - 2018.06
Hackathon Winner team	Open Network Korea (ONK)	2017.11

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