Doctoral Dissertation

LLM-based AI Agent for Virtual Network Function Deployment

Sukhyun Nam (남 석 현)

Department of Computer Science and Engineering

Pohang University of Science and Technology

2025

대규모 언어 모델 기반의 가상 네트워크 기능 배포 자동화 에이전트

LLM-based AI Agent for Virtual Network Function Deployment

LLM-based AI Agent for Virtual Network Function Deployment

by

Sukhyun Nam
Department of Computer Science and Engineering
Pohang University of Science and Technology

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

Pohang, Korea
06. 27. 2025
Approved by
Young-Joo Suh (Signature)
Academic advisor

LLM-based AI Agent for Virtual Network Function Deployment

Sukhyun Nam

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

06. 27. 2025

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

DCSE 남 석 현 Sukhyun Nam
20212397
LLM-based AI Agent for Virtual Network Function Deployment.
대규모 언어 모델 기반의 가상 네트워크 기능 배포 자동화 에이전트 .
Department of Computer Science and Engineering , 2025, 79p
Advisors : Prof. Young-Joo Suh, Prof. James Won-Ki Hong.
Text in English.

ABSTRACT

As an initial step toward intent-based networking (IBN), we conducted a study on automating Virtual Network Function (VNF) deployment based on natural language input. In particular, we aimed to develop a system that leverages recent advances in Large Language Models (LLMs), which demonstrate strong performance not only in natural language understanding but also in reasoning and code generation, to process natural language descriptions of VNF deployment. We have proposed, implemented, and validated an LLM-based AI Agent that accepts natural language inputs describing the desired VNF deployment and generates a corresponding automated workflow with high success rates.

The developed AI Agent employs a few-shot learning approach to generate VNF installation workflows. Before applying the generated workflow to a real network en-

vironment, it is first validated within a Kubernetes-based Network Digital Twin (NDT) to ensure correctness. In the event of errors, the system extracts logs from the NDT and supplies them to the LLM for iterative refinement of the workflow. To enhance this process, we integrated Retrieval-Augmented Generation (RAG) and Multi-Agent Debate (MAD) techniques, enabling the LLM to incorporate external knowledge when resolving issues.

Experimental results demonstrate that even LLMs that were initially unable to achieve fully automated VNF deployment via prompt-only interaction were capable of generating functional workflows for the NDT when guided by the proposed AI Agent. These findings validate our approach as a meaningful contribution to the translation and activation phases of IBN.

Contents

List of Tables

List of Figures

I. Introduction

1.1 Motivation

Since the commercialization of 4G/5G, the number of connected devices and the volume of data processed by networks have increased significantly. As a result, networks have become more complex and larger in scale. With the growing demands on network infrastructure, both the installation and maintenance costs of network equipment have increased accordingly. To address these challenges and reduce associated costs, Software-Defined Networking (SDN) [1] and Network Function Virtualization (NFV) [2] were proposed. Over the past decade, these two technologies have become central to network management. However, it is also true that they have contributed to increased complexity in network operations. Traditionally, network management has depended on the expertise and manual operations of skilled network administrators. However, as networks become increasingly intricate, it is becoming more difficult for human operators to fully understand network topologies and virtualized functions protocols while managing, monitoring, and troubleshooting using device/function-specific commands [3].

In 5G and beyond (B5G) environments, manual network management introduces significant risks of human errors, such as misconfigurations, and impedes the ability to respond promptly to dynamic network conditions. In addition, human errors in network management have increased over the years. In 2020, such errors accounted for less than 26% of network-related problems, rising to nearly 30% by 2024 [4]. In addition, according to a report from a large service provider (ByteDance) [5], 34.4% of incidents in the network were caused by misconfiguration. In such cases, operators must identify the root cause of the issue and manually apply a fix, and 16.6% of incidents require more than 30 minutes to resolve, with some cases taking up to 5 hours.

They reported that this delay is attributed to the inherent complexity of distributed networks. Such prolonged resolution times result in significant losses for the networking industry. Therefore, it is imperative to conduct research aimed at fully automating the network configuration process and incorporating self-healing mechanisms to address issues when they arise.

Given this context, there is growing interest in Intent-Based Networking (IBN, or Intent-Driven Networking, IDN), a closed-loop network management and operations paradigm where intents are used to define desired outcomes, and the system automatically enforces, monitors, and adjusts configurations and policies to maintain those outcomes [6]. In IBN, the system then automatically interprets and implements the intent without requiring low-level configuration.

An intent refers to "a declarative expression of the operational goals that a network should achieve and the outcomes it should deliver, without specifying how to achieve or implement them" [6]. Intents can be expressed in natural language or structured formats. Research on IBN is still in its early stages, and no fully commercial implementations currently exist.

Historically, implementing IBN has been challenging, especially in accurately interpreting user intent. However, recent advances in Artificial Intelligence (AI) have rekindled interest in IBN, with intent-based management technologies increasingly recognized as promising solutions for managing complex B5G/6G networks. A key area of research in this context is the application of Large Language Models (LLMs) [7–11].

LLMs are AI models trained on massive text datasets to perform Natural Language Processing (NLP) tasks. They have demonstrated strong performance in document summarization, translation, question answering, and text generation. Recently, they have also been explored for their potential in automating network operations, particularly in interpreting and acting on user intent—a core capability required for IBN [12–14].

Despite the development of VM orchestration frameworks such as OpenStack

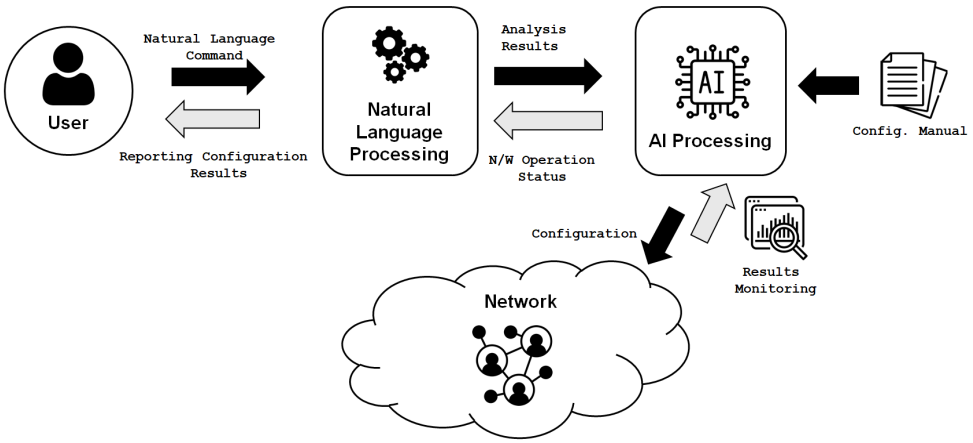

Figure 1.1: AI-based Network Configuration Management Concept

[15] and Kubernetes [16], automating VNF deployment based on natural language input remains a complex problem. Each VNF has unique configuration requirements that must be tailored to the user's intent. Addressing this challenge requires AI with a deep understanding of both VM provisioning and VNF deployment processes. This thesis aims to explore LLM-based automation for managing VNFs through intent, with the ultimate goal of enabling users to deploy and manage VNFs/CNFs using simple natural language instructions.

Fig. 1.1 illustrates the concept of VNF deployment within an IBN environment. The ultimate goal of IBN is to enable fully autonomous network management by accepting natural language inputs, performing continuous monitoring, and providing feedback or reports to the user based on the outcomes. For example, a long-term goal of IBN for NFV is to directly execute user commands like: "Deploy a VM with a single CPU and 1GB of RAM, then configure a load balancer using HAProxy to route traffic from the 10.10.10.X subnet to 10.10.10.4 and from 10.10.20.X to 10.10.20.5." However, since a general LLM does not possess omniscient knowledge, executing such instructions reliably remains a considerable technical challenge. Therefore, when utilizing an LLM, it is essential to provide relevant contextual knowledge depending

on the specific situation. In the conceptual diagram, this is represented as the configuration manual (Config. Manual). In this work, we introduce the use of Method of Procedure (MOP) [1] documents—which describe detailed procedures for specific tasks—as an additional input for LLMs.

Although MOPs include detailed configuration instructions, they still require human understanding and manual execution, which introduces the potential for error. It is expected that LLMs can be leveraged to fully automate this process. For instance, Samsung Electronics maintains internal MOP documentation for configuring and operating VNFs in its commercial SDN/NFV systems. These documents are currently used manually by engineers and are prone to human error, making them strong candidates for automation through LLMs.

1.2 Problem Statement and Research Goals

This thesis focuses on automating Virtual Network Function (VNF) deployment on a Kubernetes environment. Here, 'deployment' refers to both the creation of containers and the configuration of VNFs. Existing research on IBN has primarily focused on basic command execution, such as link disconnection, IP assignment, or intent translation. However, as techniques for leveraging LLMs more effectively continue to advance, it is becoming increasingly feasible to enable LLMs to comprehend and execute more complex commands. In our work, we have also implemented complex tasks such as VNF deployment based on LLMs.

Automating VNF deployment based on MOPs is a non-trivial task. Even for LLMs specialized in code generation, accurately configuring a network environment without errors remains a significant challenge, particularly in complex deployment scenarios. Moreover, these models cannot often debug and correct their own outputs when failures occur autonomously. To address these limitations, we enhanced the

[1] Document describing detailed procedures for performing specific tasks. https://whatfix.com/blog/method-of-procedure/

LLM's coding performance using several techniques: few-shot learning with example code, Retrieval-Augmented Generation (RAG) [17] to provide relevant documentation, and Multi-Agent Devate (MAD) [18] to incorporate perspectives from multiple LLMs.

Through the integration of these methods, we were able to develop a practical automation framework for VNF deployment that can be reliably applied in real-world network management environments. The contributions of this thesis are as follows:

- *AI Agent Framework*: We designed and implemented an AI Agent that facilitates the automated deployment of VNFs based on LLMs. The proposed AI Agent takes as input an MOP that contains the user's request along with descriptions of the target VNF, and incorporates multiple techniques to assist the LLM in generating correct automation workflows.

- *RAG and MAD*: To enhance the LLM's ability to handle complex or ambiguous scenarios, we integrated both RAG and MAD techniques. In particular, a domain-specific knowledge base related to networking was constructed to support RAG. The RAG pipeline was structurally designed and evaluated to retrieve documents relevant to the errors currently encountered during execution.

- *NDT and Test Module*: The proposed AI Agent incorporates a Network Digital Twin (NDT) environment to validate the workflow generated by the LLM before its deployment in the actual network. By executing and verifying the workflow within the NDT, the system ensures that only those workflows that successfully install the target VNF are returned for application to the real network environment.

- *MOP Dataset and Evaluation*: To evaluate the proposed AI Agent, we constructed a custom MOP dataset and conducted experiments across various VNFs. We assessed the ability of multiple LLMs to generate valid automation workflows and analyzed the extent to which our proposed techniques contributed to

improving success rates.

1.3 Organization

The remainder of this thesis is organized as follows. Chapter II introduces the background and related work that form the foundation of this thesis. It presents key concepts such as IBN and NFV, which support the overall design of the proposed system. Chapter III describes the overall architecture of the proposed AI Agent, including the use of in-context learning with LLMs and the integration of techniques such as RAG and MAD. Chapter IV provides implementation details of the AI Agent, elaborating on each module introduced in Chapter III. Chapter V outlines the experimental environment, presents the results of the evaluation, and provides comparative analyses. Finally, Chapter VI concludes the thesis and discusses directions for future work.

II. Background and Related Work

2.1 Large Language Model (LLM)

Large Language Models (LLMs) [19] are massive artificial intelligence models containing hundreds of millions to trillions of parameters designed to process and generate human language by pre-training large-scale text datasets. Primarily based on transformer architectures [20], these models are trained to recognize complex linguistic patterns, syntactic structures, and semantic relationships. LLMs have shown remarkable performance across a wide range of NLP tasks, including text generation, summarization, and translation. Examples of prominent LLMs include models like GPT (Generative Pre-trained Transformer) [19], LLaMA (Large Language Model Meta AI) [21], Gemma [22], DeepSeek [23], and Qwen [24], all of which have made significant strides in handling complex linguistic structures and providing contextually relevant responses. These models' ability to comprehend nuanced language patterns has revolutionized fields ranging from conversational AI to content creation and data analysis.

Notably, LLMs are known to suffer from the hallucination problem—generating plausible but factually incorrect information—making such task-specific adaptation strategies particularly important. Understanding how to best utilize LLMs for specific tasks remains a central research question.

More recently, as base model capabilities have advanced, research has expanded into efficient and specialized LLM variants. This includes lightweight models such as Phi [25] and Mistral [26], which aim to maintain performance while reducing parameter counts; task-specific models like CodeGemma [27] and Qwen-coder [28] for code generation; RAG-based systems that supply external knowledge to LLMs; and multimodal LLMs that accept heterogeneous input types beyond text. These trends reflect

the ongoing effort to make LLMs more effective, efficient, and adaptable to a wider range of use cases.

Since most LLMs are trained primarily on general-purpose knowledge, specialized techniques are required to apply them effectively to domain-specific tasks that demand expert-level understanding. Among these, in-context learning offers a means of guiding pretrained models by providing task-specific prompts; fine-tuning adapts pretrained models to a particular task using labeled datasets; reinforcement fine-tuning (e.g., with human feedback) improves response quality through iterative reward-based optimization; and Retrieval-Augmented Generation (RAG) retrieves relevant information from an external knowledge base when knowledge beyond what the LLM has been pretrained on is required. These approaches have been widely studied as mechanisms to extend the utility of LLMs across diverse application domains.

2.1.1 In-context Learning

This approach leverages the pre-trained model as is, without modifying the model's parameters. It involves incorporating a few examples of the desired task into the input prompt (One-shot or Few-shot methods) [29] or providing instructions for the task without any examples (Zero-shot method). The model is then guided to produce the desired output based on these prompts. In this process, prompt engineering, the technique of designing input prompts to elicit desired responses, plays a critical role. While this method allows for rapid deployment across various tasks without additional training time, the quality of results is heavily influenced by how well the prompts are designed, and it may not perform optimally for complex tasks.

LLMs are known to exhibit a phenomenon referred to as 'lost in the middle' [30], wherein the model tends to better retain and attend to information presented at the beginning and end of the input, while overlooking content positioned in the middle. As such, prompt design that accounts for this characteristic is critical for maximizing LLM performance.

In recent discourse—both in academic and popular contexts—there has been growing interest in how to effectively utilize LLMs, with widespread discussions on best practices for prompt construction. Strategies such as prioritizing key instructions at the beginning of the prompt or emphasizing corrective feedback over praise have emerged as practical heuristics. These techniques can be broadly categorized under the umbrella of prompt engineering, which aims to systematically guide LLM behavior through structured input design.

2.1.2 Fine-Tuning

In contrast, fine-tuning involves further training the LLM's parameters on a specific dataset tailored to a particular task [29]. This enables the model to learn and retain task-specific patterns, thereby optimizing performance for that task with high accuracy and consistency. However, fine-tuning comes with challenges, including the significant time and computational resources required due to the vast number of parameters in LLMs and the need for large, task-specific datasets.

Recently, full-parameter tuning of LLMs has become increasingly impractical due to the associated computational cost. As a result, Parameter-Efficient Fine-Tuning (PEFT, [31]) methods have gained significant attention. Among these, Low-Rank Adaptation (LoRA, [32]) has emerged as a widely adopted approach, which simplifies the update process by applying low-rank factorization to the linear layers of LLMs.

These lightweight adaptation techniques enable Supervised Fine-Tuning (SFT) with smaller datasets, thereby lowering the resource requirements for domain-specific model customization. Consequently, a growing body of research has explored the use of PEFT methods—particularly LoRA—to efficiently fine-tune LLMs across a variety of specialized tasks.

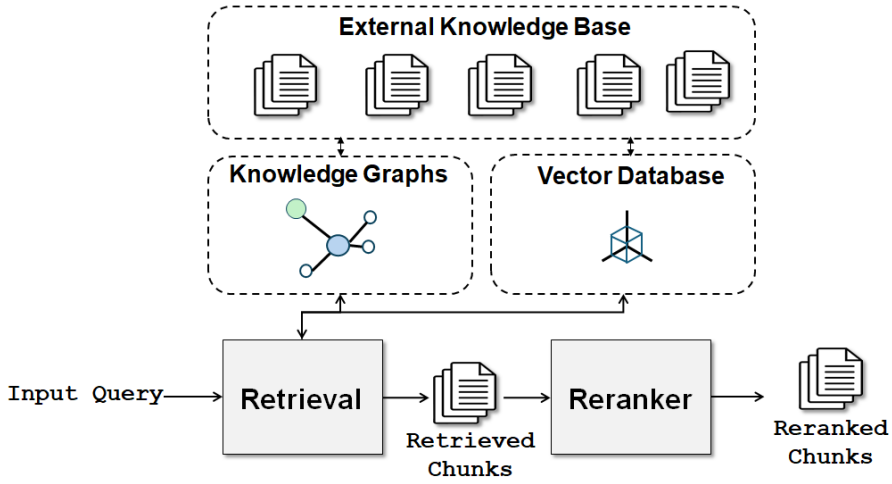

Figure 2.1: General RAG Structure

2.1.3 Retrieval-Augmented Generation (RAG)

Since an LLM is trained primarily on general-purpose knowledge, it exhibits inherent limitations when applied to domain-specific tasks that require specialized expertise. To address this challenge, Lewis et al. [17] introduced the Retrieval-Augmented Generation (RAG) framework, which integrates a retrieval module into the sequence-to-sequence generation process. RAG dynamically retrieves relevant documents from a large corpus during inference, enabling the model to ground its outputs in external textual evidence.

At the core of RAG lies a dense vector retrieval mechanism, typically utilizing pretrained embedding models such as DPR (Dense Passage Retrieval) to encode queries and corpus documents into a shared vector space. Relevance is computed via similarity measures (e.g., dot product or cosine similarity), allowing the top-k documents to be retrieved efficiently. In addition to vector-based methods, recent advances incorporate graph-based retrieval techniques that model inter-document relationships to enhance context aggregation and semantic coherence of entities in documents. These hybrid retrieval schemes support richer knowledge integration, particu-

larly in complex question answering and dialogue generation tasks. Also, to mitigate the 'lost in the middle' [30] issue in LLMs, re-ranking mechanisms are often employed. Specifically, in the context of RAG, retrieved chunks are assigned importance scores based on their relevance to the task or query. These scores are then used to reorder the chunks such that more important content is placed at the beginning or end of the prompt, where LLMs are more likely to attend.

The principal advantage of RAG is its ability to generate outputs that are both contextually coherent and factually grounded, leveraging up-to-date information without the need for model retraining. This makes it particularly effective in dynamic or specialized domains, such as legal, biomedical, or technical knowledge-intensive tasks. Moreover, RAG models exhibit greater interpretability, as the retrieved documents can be inspected to trace the source of generated content. By decoupling knowledge storage from the generation model, RAG also facilitates more scalable and modular system design.

2.2 Network Function Virtualization (NFV)

Network Function Virtualization (NFV) represents a transformative shift in network architecture, wherein traditional hardware-based network functions—such as firewalls, routers, and intrusion detection systems—are decoupled from dedicated appliances and implemented as software-based virtual network functions (VNFs). This paradigm enables the deployment of network functions on general-purpose computing platforms, thereby enhancing flexibility, reducing capital and operational expenditures, and accelerating service innovation. The NFV architectural framework, as defined by the European Telecommunications Standards Institute (ETSI), comprises three principal components: the VNFs themselves, the underlying NFV Infrastructure (NFVI), and the Management and Orchestration (MANO) framework [33] that governs their lifecycle and resource allocation.

The NFVI encompasses the physical and virtualized compute, storage, and net-

working resources that support the execution of VNFs. In early implementations, hypervisor-based virtualization (e.g., using KVM or VMware) was predominant. However, with the increasing adoption of cloud-native design principles, NFVI has evolved to incorporate container-based architectures. In this context, Kubernetes [16] has emerged as a de facto standard platform for orchestrating cloud-native network functions (CNFs). Through mechanisms such as custom resource definitions (CRDs), Kubernetes Operators, and the Container Network Interface (CNI), Kubernetes provides a robust foundation for managing telecom-grade workloads in a scalable and declarative manner.

The adoption of NFVI yields numerous operational and strategic benefits. It enhances resource efficiency through infrastructure consolidation, reduces time-to-market for new services via agile instantiation, and lowers total cost of ownership by minimizing reliance on proprietary hardware. Furthermore, NFVI fosters vendor interoperability and multi-tenancy, enabling flexible deployment of VNFs/CNFs across heterogeneous environments. As networks continue to evolve toward distributed and programmable architectures, NFVI—particularly in its container-native form—is poised to serve as a foundational enabler of next-generation network services.

Although various NFVI technologies have been developed over the years, Kubernetes has emerged as the most widely adopted and dominant platform in recent network service deployments. Kubernetes uses technologies such as SR-IOV, DPDK, and Multus to facilitate high-throughput, low-latency packet processing, which is critical for carrier-grade performance. As Kubernetes has become a dominant technology in the cloud-native ecosystem, a wide range of third-party services have been developed to support its efficient operation. However, the increasing complexity and diversity of these surrounding tools have made it progressively more challenging to manage Kubernetes clusters with a comprehensive understanding of the entire stack. In response to this, various technologies like Nephio [11] and GitOps [34] aimed at achieving full automation of Kubernetes management are actively being developed. These capabilities render Kubernetes-based NFVI particularly well-suited for emerging 5G and edge

computing scenarios, where dynamic resource allocation and multi-domain orchestration are essential.

In general, an NFVI either provides a custom-developed GUI or exposes its management capabilities through APIs. Alternatively, open-source libraries such as Ansible [35], which allow users to issue commands to the NFVI via YAML file inputs, are also commonly used.

In this thesis, since our goal was to automate VNF deployment based on an LLM, we utilized either the Kubernetes API (accessible from Python) or Ansible to communicate with the NFVI.

2.3 Intent-Based Networking (IBN)

Intent-Based Networking (IBN) [6] is an advanced networking paradigm that automates the management of network resources by translating high-level business intentions into network configurations and policies. This approach allows network operators to express their goals and desired outcomes in high-level language, which IBN systems then interpret and implement through automated processes. By focusing on the intent rather than the specifics of the network configuration, IBN aims to enhance agility, reduce operational complexity, and improve overall network performance, enabling organizations to respond swiftly to changing business needs.

Previously, the implementation of IBNs faced significant challenges, particularly in accurately interpreting intents [6]. However, advancements in AI have renewed interest in this field. IBN-based management technologies are now recognized as a promising solution for addressing the escalating complexity of B5G/6G networks. Among the emerging technologies, the application of LLMs is particularly notable.

In the context of IBNs, an intent refers to an abstract request that specifies the desired network state. Intents may be articulated in natural language or represented numerically within a predefined format. Although IBN research remains in its nascent stages, with no commercially developed implementations to date, foundational studies

have focused on basic functionalities such as link disconnection and IP configuration. Over time, research has progressed towards encompassing more complex aspects of network management, including Service Level Agreement (SLA) management, VNF and CNF orchestration, and the automated configuration of network devices.

LLMs are advanced AI models trained on extensive text datasets, excelling in natural language processing (NLP) tasks such as summarization, translation, question answering, and text generation. Their capacity for understanding and generating human-like language makes LLMs highly versatile tools across various domains. Their potential application in network automation, particularly for implementing IBNs, has recently gained traction. A critical aspect of IBN realization is the analysis and automation of user intents—a capability significantly enhanced by LLM advancements. Consequently, recent research has increasingly explored LLM-driven approaches to IBN implementation [7–14].

General LLMs lack a deep understanding of network configuration knowledge, necessitating highly detailed explanations of the required configuration environment. Also, when necessary, supplementary reference documents relevant to the configuration should be provided. Furthermore, it is essential to verify whether the LLM has produced erroneous configuration outcomes. If inaccuracies are identified, a correction process must be implemented. Only by automating all these steps can the network configuration process be fully automated as intended. These aspects represent the current challenges in achieving complete automation of network configuration.

Current research in IBN is centered around improving the frameworks and technologies that facilitate intent interpretation and implementation. Recent studies explore the integration of AI, particularly machine learning and NLP, to enhance the accuracy and efficiency of intent translation [36]. Furthermore, there is a growing emphasis on developing standardized protocols and interfaces to ensure interoperability among diverse network devices and platforms. As organizations increasingly adopt SDN and NFV, IBN research aims to address challenges related to scalability, security, and real-time adaptability, thereby advancing the capabilities of next-generation

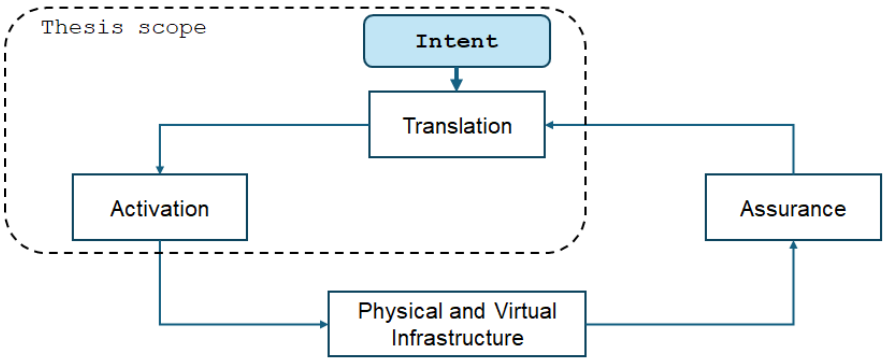

Figure 2.2: Thesis Scope in IBN Closed Loop

networks.

As illustrated in Fig. 2.2, the ultimate objective of IBN is to achieve closed-loop management by translating the given intent, activating it on physical or virtualized infrastructure, and continuously monitoring the system to ensure that the intent is fulfilled. However, as will be discussed in Section 2.4, IBN research remains in its early stages, and even successful demonstrations of the translation and activation phases are currently scarce. In this thesis, we specifically focus on addressing the translation and activation components of the IBN framework.

2.4 Related Work

Although there is currently no existing research on automating VNF deployment using LLMs, many studies are conducting research on several tasks in IBN based on LLMs.

D. Donadel *et al.* [7] conducted an evaluation to assess whether LLMs can comprehend network topology information and respond to system administrator-level queries. Their study focused on verifying the capability of LLMs to function as virtual system administrators by understanding topological structures, calculating paths, and identifying subnets and IP addresses. The evaluation involved presenting LLMs with pre-

defined network topologies—ranging from simple to complex—and prompting them with relevant questions in a zero-shot setting. A total of six models were compared, including proprietary models such as GPT-4 and open-source models like LLaMA2. Results showed that while some models demonstrated high comprehension accuracy (up to 79.3%) on simple topologies, their performance declined significantly on more complex configurations. These findings highlight the current limitations of LLMs in understanding intricate network structures and suggest the need for further research. The study also emphasized the critical role of prompt engineering in enhancing the performance of LLMs in network-related tasks.

A. Mekrache *et al.* [12] present experimental evidence supporting the use of LLMs as execution engines for IBN. They propose a unified approach to managing heterogeneous infrastructure—spanning cloud, edge, network, and RAN domains—through natural language. Their framework was evaluated using the EURECOM 5G testbed, where they employed Code Llama to perform intent decomposition and translation. In their proposed architecture, natural language intents are processed by the LLM and converted into subdomain-specific JSON/YAML configurations, which are then activated and validated in the target network environment. The study demonstrates a successful proof-of-concept in a 5G setting, confirming stable operation across service definition, decomposition, and execution phases. Despite its significance, the approach suffers from notable limitations. The LLM exhibited frequent errors in natural language understanding, and managing multiple domains simultaneously introduced challenges such as cross-domain conflicts, as well as security and trust concerns. Moreover, since the system relies on direct translation of intents into static JSON/YAML formats, its scalability for more dynamic or evolving service requirements is limited.

J. Lin *et al.* [8] conducted experiments using LLMs to convert high-level, multi-domain network requirements specified in natural language into detailed network configurations and settings. In this process, few-shot learning was employed with data on specific network configurations and settings, and it was observed that providing

configuration examples as input led to more accurate network configuration requests.

E. Jeong *et al.* [9] introduced "S-Witch," an LLM-powered assistant designed to aid in switch configuration by using prompt engineering techniques. S-Witch enables users to generate precise switch configurations by interpreting natural language inputs. The study proposes a hierarchical LLM architecture by separating the LLMs responsible for managing the design and configuration of network switches. To evaluate this approach, a simplified network digital twin was designed and used for experimentation. Through specialized prompt engineering, the system optimizes the interaction with LLMs, achieving high accuracy in interpreting user intents and converting them into specific switch commands.

NETBUDDY [13] is a study aimed at leveraging LLMs to make network configuration more human-friendly. The authors explore the feasibility of automatically translating high-level natural language policies and requirements into low-level configurations, including P4, BGP, and other network APIs. Their approach involves converting abstract prompts into executable code using an LLM, incorporating mechanisms for step-wise error detection and self-healing throughout the configuration process. The system was evaluated in a network emulation environment, demonstrating the potential of LLMs to generate runnable network configurations. However, the study also identified significant limitations in the accuracy of the generated configurations. As a result, the authors emphasize the importance of prompt engineering and incremental error correction to improve reliability.

There are also several studies focused on automating VNF/SFC (Service Function Chain) management through the use of LLMs. K. Dzeparoska *et al.* [10] focused on automating the generation of management policies for applications using LLMs. Specifically, the framework utilizes LLMs to interpret high-level user intents and automatically create corresponding policies that guide the application configuration, deployment, and operation of SFC.

In our previous research [14], we proposed a framework that leverages LLMs to facilitate intent-based configuration of NFV. We defined six actions related to VNF

and SFC and conducted experiments to transform natural language containing these actions into a predefined NFV-Intent template. Using a large dataset generated with GPT, we compared various LLM models and found that the GPT model performed best in executing these transformations. This experiment demonstrated not only the feasibility of simple intent extraction from natural language but also the potential of LLMs in network management automation, which we explored in this thesis.

Nephio [11] is an open-source initiative launched by Google and the Linux Foundation in 2022, aiming to simplify the automation and orchestration of CNFs across distributed, multi-vendor telco networks. The core goal of Nephio is to bring Kubernetes-native IBN and GitOps principles to network management by abstracting infrastructure complexities and enabling declarative configuration of network functions and services.

Nephio leverages Kubernetes as a control plane to manage not only compute resources but also the lifecycle and configuration of CNFs, enabling zero-touch provisioning, automated reconciliation, and hierarchical resource management. It introduces custom Kubernetes resources and controllers for telecom-specific workflows, emphasizing modularity, vendor neutrality, and cloud-native scalability.

As of 2024, Nephio has released several functional components, including the Nephio Reference Implementation (NRI), Topology Controller, and Package Reconciler, with integration tests conducted across major cloud-native CNF providers. Academic and industrial stakeholders continue to explore its feasibility for 5G RAN orchestration, edge network automation, and multi-domain service provisioning. Nephio is being developed to achieve fully automated, intent-driven Kubernetes networking, with a particular focus on enabling zero-touch networking. Its ongoing development emphasizes capabilities such as continuous post-deployment monitoring, self-healing, and automated rollback mechanisms. However, as of 2025, natural language-based approaches to network configuration remain in the early stages of development. Specifically, VNF installation continues to rely on YAML/CRD-based methods, and natural language-driven VNF installation—the central topic of this thesis—remains largely experimental.

As observed, numerous recent studies are actively exploring the application of intent-based approaches to various aspects of traditional network management. However, to the best of our knowledge, IBN remains at an early stage of research and development, no prior work has achieved full automation of VNF installation—the primary objective of this thesis. Table 2.1 highlights the key differences between existing approaches and our proposed method.

Method	Target	Output/Goal	Year
D. Donadel *et al.* [7]	Network topology QA System	Topology information	2024
Mekrache et al. [12]	E2E network configuration	Domain-specific requirement formats	2024
AppleSeed [8]	Network services across multiple domains	Python code framework	2023
S-Witch [9]	Network switch configuration	Switch configuration commands	2024
NETBUDDY [13]	Network policies translation	Low-level network APIs	2023
K. Dzeparoska et al. [10]	SFC management policy	Actionable policies	2023
N. Tu et al. [14]	NFV configuration	Configuration management templates	2024
Nephio [11]	Intent-based fully automated networking system	K8S manifets (YAML/Helm, network config)	In progress
Ours	Natural language based VNF deployment	NFVI runnable workflow	-

Table 2.1: Related Work Comparison

III. System Design

In this chapter, we present the design of our proposed LLM-based VNF deployment AI Agent.

3.1 Overview of LLM-based VNF Deployment AI Agent

Given the absence of large-scale input–output datasets necessary for fully automating VNF deployment, we did not apply fine-tuning. Instead, using in-context learning, we applied additional techniques to suit our task with open LLM. Rather than employing a newly developed AI model, the proposed agent utilizes an existing, publicly available LLM. As its primary role is to enhance the usability and effectiveness of the underlying LLM, we refer to our proposed system as an LLM-based 'AI Agent'.

As outlined in the introduction, the problem addressed in this work presents significant technical challenges. Although many recent LLMs have been developed specifically for code generation, deploying a VNF requires additional knowledge of Kubernetes configurations. It also demands an understanding of MOPs and the ability to generate code that aligns with the specific task being performed. To enhance the accuracy of code generation, the agent employs a range of supporting techniques. Furthermore, to ensure the reliability of the generated outputs, a verification phase using a Network Digital Twin (NDT) [37] is performed prior to actual deployment.

Fig. 3.1 illustrates the architecture proposed and employed in our experiments. We opted to utilize an MOP that contains detailed instructions for the process, rather than requiring the user to provide a comprehensive explanation of the VNF. The AI Agent accepts an MOP related to VNF deployment as input and generates a corresponding workflow (Python code or Ansible YAML code) to automate the deployment

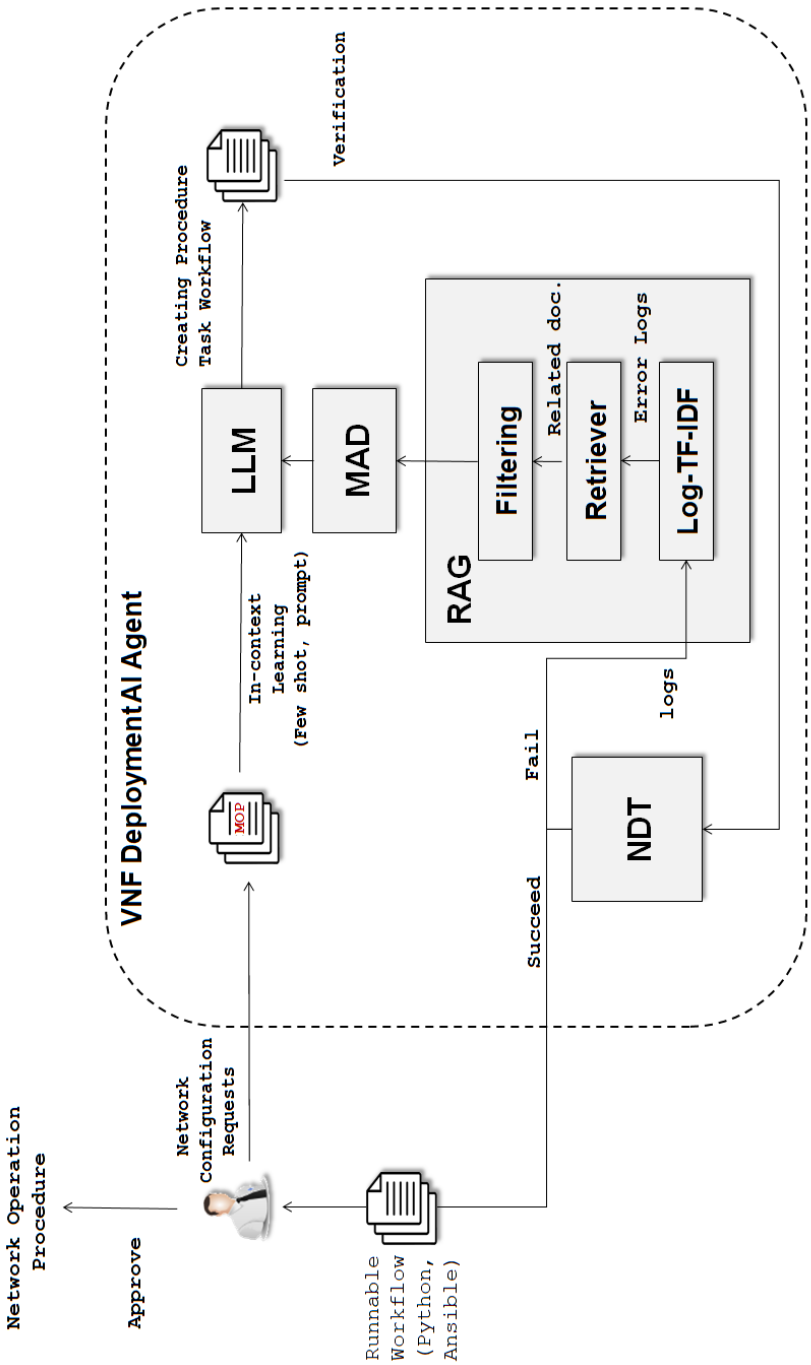

Figure 3.1: LLM-based VNF Deployment AI Agent Structure

process, and through the verification process at NDT, a stable workflow is created before application to the actual network. Through the verification process at NDT, a stable workflow is created before application to the actual network. NDT and verification modules will be described in more detail in the Section 4.3.

We propose a framework in which, when an error occurs during the execution of LLM-generated code in the NDT environment, the corresponding logs are provided to the LLM to prompt code revision based on the error context. Through preliminary experiments, we observed that the LLM often failed to correctly fix the code even when error logs were given as input. This limitation is attributed to the fact that the LLM employed in our thesis is not specifically fine-tuned for networking-related coding tasks and, even if partially specialized, lacks domain-specific expertise in Kubernetes-related operations.

To address this issue, we adopted RAG and Multi-Agent Devate (MAD) techniques to enhance the LLM's ability to interpret and resolve such errors. These methods aim to augment the LLM's code generation capabilities by providing relevant external knowledge and leveraging multi-agent reasoning.

Fig. 3.2 illustrates the complete workflow triggered when input is provided to the AI Agent. Also, Algorithm 1 outlines the underlying mechanisms of the proposed algorithm in greater detail. When a user issues a VNF deployment request, the system converts it into a predefined prompt format and forwards it to the LLM along with the corresponding MOP. The LLM generates a workflow based on the input prompt, which is then passed to the testing module for verification within the NDT environment. In addition, to support future MAD-based reasoning, responses from other LLM instances are collected and sent to the MAD module.

If the testing result indicates success, the workflow is immediately returned to the user as an executable output. However, in practice, this is rarely the case. Most workflows require refinement. In such cases, the testing module collects execution logs from the NDT and sends them to the RAG module. The RAG component filters the logs to extract relevant error messages (input filtering) and uses them to retrieve

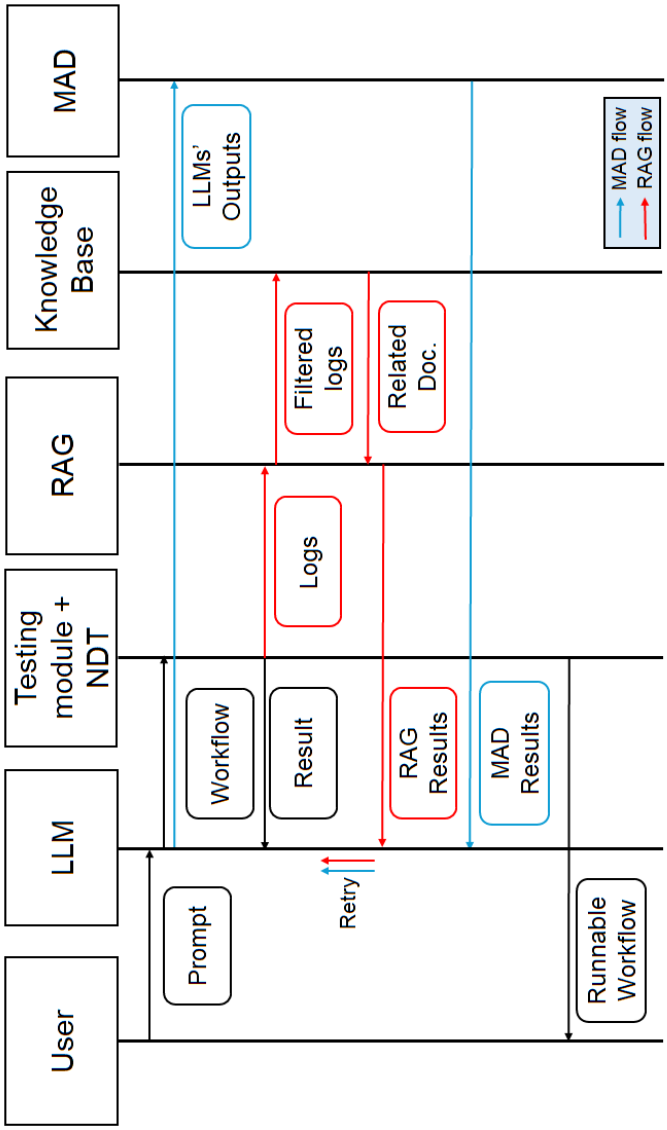

Figure 3.2: LLM-based VNF Deployment AI Agent Workflow

Algorithm 1 LLM-based VNF Deployment AI Agent

Require: $MOP, VNFConfigureRequests$
Ensure: $ExecutableWorkflow$

$LLMInput \leftarrow PromptFormat + MOP + VNFConfigureRequests$
for $0 \leq i < MaxTryNum$ **do**
    $Workflow \leftarrow MainLLM(LLMInput)$
    $VerificationResult, ResultLogs \leftarrow TestingModule(Workflow)$
    if $VerificationResult == Succeed$ **then**
        return $Workflow$
    else
        $MADResults \leftarrow SubLLMs(LLMInput)$
        $LLMInput \leftarrow PredefinedModifyingFormat + MadResults$
        $RelatedDoc \leftarrow RAG(ResultLogs)$
        $Related, FilteredDoc \leftarrow JudgeLLM(RelatedDoc)$
        if $Related == True$ **then**
            $LLMInput \leftarrow LLMInput + FilteredDoc$
        end if
    end if
end for
return False

related documents from the knowledge base. A judge LLM evaluates the retrieved content, filtering and selecting only the most useful information (output filtering).

Simultaneously, the MAD module delivers the responses of other LLMs to the current LLM instance to serve as a reference. The LLM then receives inputs consisting of the error logs, RAG output, and MAD suggestions, and proceeds to revise the code accordingly. The modified workflow is passed back into the pipeline, and the process is repeated iteratively until a runnable workflow is successfully generated. To avoid infinite loops in cases where repeated revisions fail, a maximum number of attempts is predefined, after which the loop terminates.

This constitutes the complete operational flow of the AI Agent. The detailed design and functionality of each module are described in the following sections.

3.2 In-Context Learning

One of the most fundamental techniques for utilizing LLMs in specific tasks is in-context learning. Naturally, we adopted this method as a baseline to improve the performance of the LLM. In in-context learning, the only controllable aspect is the input prompt; thus, the design of the prompt becomes critical. We explored a variety of prompt structures to investigate under which conditions the LLM generates correct code that aligns with the intended task.

Fig. 3.3 illustrates the final input structure of the prompt used in our experiments. As mentioned earlier, the MOP related to the target VNF is provided as input. We conducted a research collaboration with Samsung, and they provided a manual of developed network function by the form of MOPs. According to Samsung, errors frequently occur when following these MOPs during the installation process in real network environments, highlighting the need for automation. Therefore, we decided to investigate whether an LLM could understand and automate the installation process by using MOPs as input. Also, the MOP is not presented in isolation; rather, the prompt includes contextual information describing the current task, constraints to follow, and

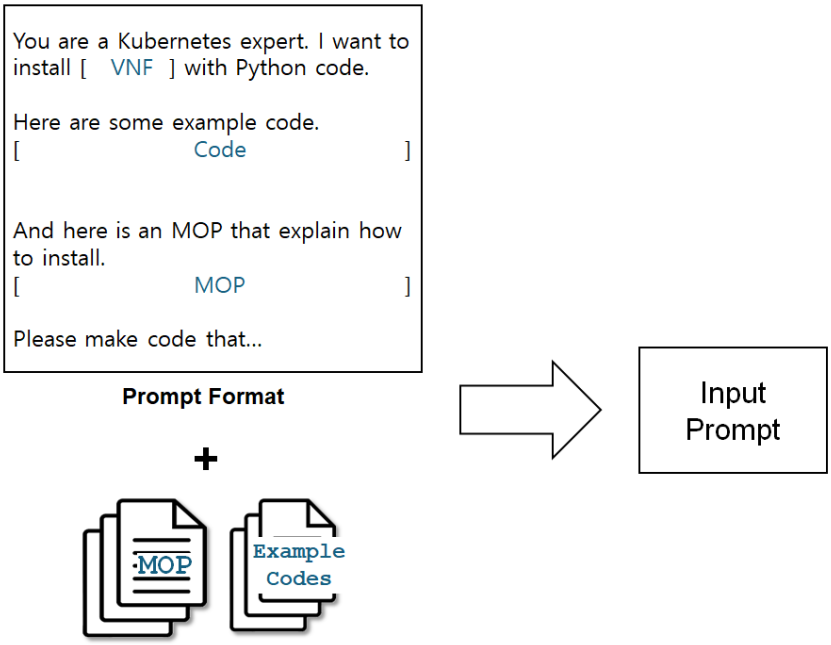

Figure 3.3: Prompt Formatting

the desired output. We formulated a predefined prompt format incorporating these elements. When the user specifies the target VNF, provides the corresponding MOP, and optionally includes additional configuration instructions, the system automatically inserts this information into the predefined format to generate the final prompt.

To further improve performance in complex tasks, it is known that providing examples—i.e., few-shot learning—is beneficial for LLMs. Accordingly, we included reference examples in the prompt. These example codes do not correspond to the target VNF but rather to other VNFs, allowing the LLM to observe how to configure a Pod and install a VNF, and to extrapolate this process to a new VNF.

Throughout our experiments, we observed that the LLM occasionally introduced common errors into the generated code. While adding constraints for every possible mistake into the prompt would make it excessively long, we identified specific critical issues (e.g., including infinite loops in Python, or generating incorrect error-handling

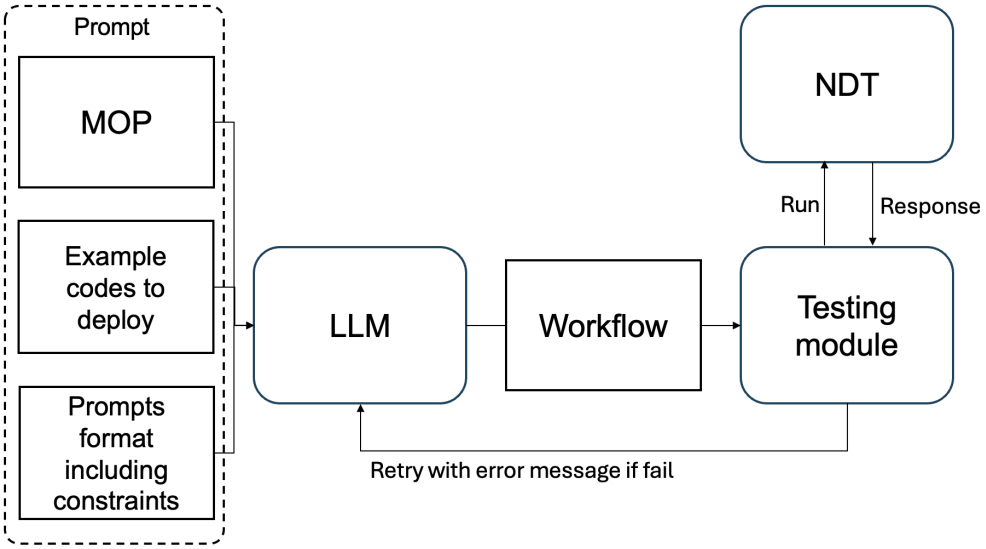

Figure 3.4: In-Context Learning Structure

logic) that could negatively affect our AI Agent's performance. For such cases, we explicitly added guidance in the prompt to instruct the LLM to avoid these errors. These instructions are incorporated into the predefined prompt format to reduce the likelihood of such failures.

Once the complete prompt is assembled and fed into the LLM, the model generates a workflow in the form of Python or Ansible YAML code. This output is then passed to other modules within the AI Agent for further processing. Detailed examples of the prompt format used in our system are presented in the Chapter IV.

Fig. 3.4 shows the AI Agent structure from now on. With only this step, the user can configure the VNF using an LLM. However, as previously discussed, this baseline structure often results in errors and failures in VNF deployment due to the inherent limitations of the LLM. To address these shortcomings, we integrated additional techniques on top of this base architecture to enhance the reliability and accuracy of the overall system.

3.3 RAG (Retrieval-Augmented Generation)

Throughout our experiments, we observed that the LLM frequently failed to correct erroneous code, even when provided with explicit error messages. This result will be covered in Chapter V. This limitation is primarily attributed to the general-purpose nature of LLM pretraining, which lacks deep domain-specific knowledge, particularly in networking and infrastructure automation. To address this shortcoming, we leveraged RAG to inject relevant, high-quality network-related information into the model's input context so that the AI Agent can perform self-healing.

The core idea of RAG involves constructing an external knowledge base composed of task-relevant information, retrieving documents related to the input query, and supplying either the entire document or contextually matched segments to the LLM as additional input.

In this work, we adopt a RAG-based approach to assist in correcting errors that arise during VNF deployment. Specifically, when deployment failures occur, relevant external documents are retrieved and provided to the LLM to support error resolution. LLMs are commonly employed in question-answering (QA) systems, and RAG has likewise been primarily designed to support such applications by retrieving specific pieces of "knowledge". As a result, conventional RAG frameworks are typically optimized to identify and extract relevant entities and their relationships from a knowledge base. Retrieved content often includes selected sentences that are closely linked to the target entity, and in some cases, the sentence order may be restructured to serve the QA context better.

In contrast, our objective differs significantly: rather than locating factual knowledge, our system requires contextually appropriate information to help interpret and resolve log-based errors. In this setting, the notion of discrete "entities" is largely irrelevant. Furthermore, technical documents such as manuals and troubleshooting guides generally present information in a well-structured, sequential manner, which makes sentence reordering not only unnecessary but potentially harmful to comprehension.

Therefore, standard RAG techniques are not suited to our use case. To address this gap, we designed a custom RAG framework specifically tailored to the needs of our system, focusing on preserving document structure and extracting semantically relevant context for resolving infrastructure-related errors.

To build the external knowledge base, we crawled two primary sources from the web: (1) official documentation and (2) Stack Overflow posts [38]. For official documentation, we collected structured content from the official Kubernetes and Ansible documentation sites. Stack Overflow posts were gathered using keyword-based queries targeting Kubernetes, Ansible, and VNF-related discussions. The Stack Exchange API [39] was used to retrieve Stack Overflow data, while other web documents were obtained using custom HTML parsing scripts.

As previously mentioned, the concept of entities is ambiguous in the context of the knowledge we aim to utilize, and identifying relationships between entities is unnecessary for our task. Therefore, instead of relying on conventional RAG methods that depend on entity-centric retrieval, we adopted the most fundamental and widely used approach: vector-based retrieval.

Using a pre-trained sentence embedding model, RAG vectorizes the titles of all documents in the knowledge base. In the case of Stack Overflow documents, RAG additionally includes the question body in the embedding process. Input error logs are similarly transformed into vector representations, and retrieval is performed by identifying the document (or document title and question pair) with the highest vector similarity to the input log.

However, during this process, we encountered two major challenges: (1) Input logs often contained noise or unrelated information, which led to the retrieval of irrelevant documents. (2) In some cases, no relevant documents existed in the knowledge base, resulting in the retrieval of documents unrelated to the actual error.

In both scenarios, the LLM may receive context that is not only unhelpful but also misleading, ultimately degrading its ability to generate appropriate fixes for the error.

To address these issues, we introduce two filtering mechanisms into our pipeline:

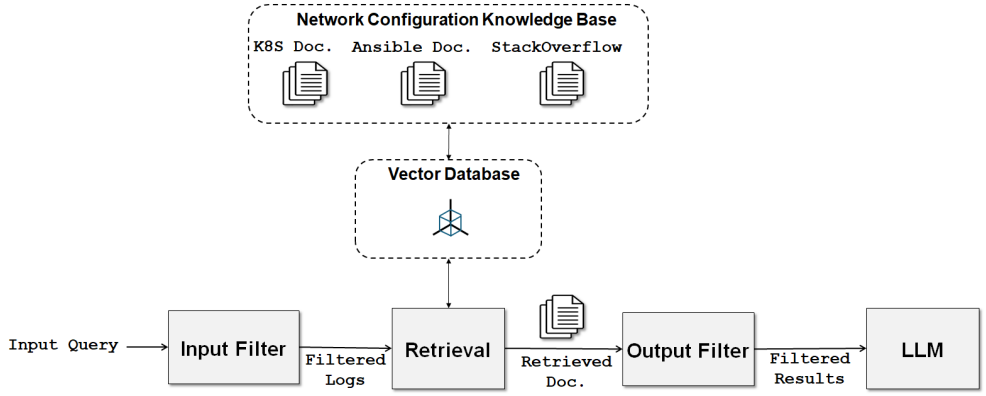

Figure 3.5: RAG Structure

- *Input-side filtering*: Attempts to isolate and extract the actual error log from noisy input.
- *Output-side filtering*: Discards retrieved documents that are unlikely to be relevant to the current error context.

These filtering steps are essential for ensuring that only semantically meaningful and contextually aligned documents are provided to the LLM during the RAG process. Fig. 3.5 illustrates the structure of our RAG module.

We employed Log-TF-IDF for input filtering and a Judge LLM for output filtering. The detailed mechanisms and implementation strategies for each of these techniques are described in the following subsections.

3.3.1 Log-TF-IDF

In our prior work [40,41], we investigated anomaly detection in network switches by analyzing the distribution of log patterns in normal and abnormal states. Motivated by the concept of Term Frequency-Inverse Document Frequency (TF-IDF, [42]), a well-established statistical measure in NLP, we proposed Log-TF-IDF, a method tailored to quantify the rarity of log patterns. While traditional TF-IDF is used to evaluate

the importance of words within a document relative to a corpus, Log-TF-IDF adapts this principle to the domain of system log analysis. The TF-IDF technique can be broken down into two main components, each serving a specific role (we will refer to 'document' as 'doc.' and 'number' as 'num.' in equations for brevity):

- **Term Frequency (TF)**:

$$TF(w, d) = \frac{\text{Num. of word } w \text{ appears in doc. } d}{\text{Total num. of words in doc. } d}$$

  Quantifies how frequently a term appears in a document.

- **Inverse Document Frequency (IDF)**:

$$IDF(w) = \log\left(\frac{\text{Total num. of doc.}}{\text{Num. of doc. containing word } w \ + 1}\right)$$

  Penalizes terms that appear across many documents, highlighting terms that are more unique.

$$TF\_IDF(w, d) = TF(w, d) \times IDF(w)$$

The standard formulation is particularly effective for tasks such as information retrieval, text classification, clustering, and keyword extraction. In NLP applications, both TF and IDF are typically log-scaled to accommodate the large number of documents and avoid division-by-zero errors, e.g., by adding 1 to the denominator.

We observed that abnormal system behavior is often accompanied by an increased occurrence of log patterns that are rare in normal conditions. Therefore, we focused on measuring the rarity of log events (similar log patterns)—rather than their absolute frequency—within a corpus of normal system logs.

To this end, we treat each log event extracted via a log parser as the equivalent of a "term", and use log entries grouped by date as the analog of "documents". Instead of evaluating the importance of an event in a specific document, we assess its rarity

in the entire normal dataset. This approach mirrors the penalty mechanism employed by the IDF component in traditional TF-IDF, which assigns lower importance to terms that appear frequently across the entire corpus, regardless of their frequency within individual documents. If a log event exhibits a relatively low overall occurrence count, it is penalized during the IDF computation if it appears consistently across all days. For example, certain log messages may be generated at a fixed time each day; while these logs may not occur frequently overall, their persistent appearance indicates that they are unrelated to abnormal system behavior. Thus, the IDF component effectively penalizes such events to reduce false associations with anomalies.

In line with this concept, our thesis analyzes the rarity of each log event to identify potential error logs within a given input. We continuously collected log data by running LLM's output workflow in our experimental environment and manually annotated the logs to distinguish normal logs from error logs. This annotated dataset is used to construct a baseline of normal log behavior. This allowed us to perform log analysis in a per-dataset manner, analogous to the per-date analysis in the original Log-TF-IDF methodology.

The resulting formulation of Log-TF-IDF mirrors traditional TF-IDF in structure but adapts both components to the log analysis context.

Let ND denote the normal dataset. The Log-TF-IDF of a log event t is defined by two components (in equations, $\log$ denotes logarithm and *Log* denotes log message):

- **Log-TF**:

$$Log\_TF(t) = \log\left(\frac{\text{Total num. of } \textit{Log}\text{s in ND}}{\text{Num. of event } t \text{ appears in ND}}\right)$$

  This inverse frequency formulation ensures that events with lower occurrence counts are assigned higher Log-TF scores. The logarithm mitigates the effect of large values and aligns with the scaling used in TF-IDF for large corpora. The addition of 1 avoids division by zero, allowing the method to handle previously unseen event.

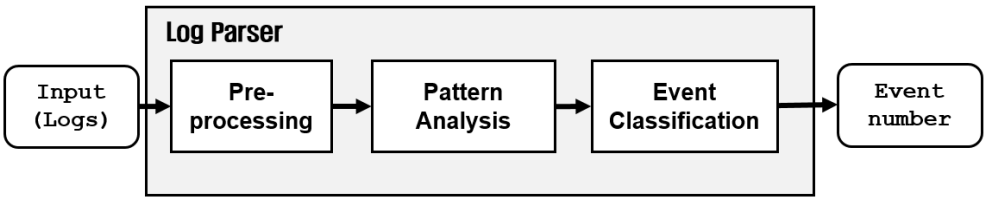

Figure 3.6: Log Parser Structure

- **Log-IDF**:

$$Log\_IDF(t) = \frac{\text{Total num. of distinct datas in ND} + 2}{\text{Num. of datas that event } t \text{ appears} + 1}$$

Unlike in NLP, where the corpus often contains thousands of documents, system log datasets usually span a smaller amount of data. For this reason, we omit log-scaling for the Log-IDF component. To ensure Log-IDF values remain greater than one—preventing the entire Log-TF-IDF score from collapsing after applying logarithms—we add 2 to the numerator.

The final Log-TF-IDF score for event *t* is then computed as:

$$Log\_TF\_IDF(t) = \log\left(Log\_TF(t) \times Log\_IDF(t)\right)$$

This formulation allows us to detect events that are rare either due to infrequent total appearances or sporadic date-wise occurrences.

In our previous work, experimental results demonstrated that utilizing both Log-TF and Log-IDF jointly provides a more effective means of quantifying the rarity of log events across the entire dataset. Moreover, for the extraction of log events necessary for calculating Log-TF-IDF, we employed the log parser and event classification model developed in our earlier study [43].

Fig. 3.6 presents an overview of the log parser architecture. The log parser is designed to abstract and normalize input logs by removing non-essential information, thereby enabling the grouping of semantically similar log entries into representative

patterns. Initially, the parser replaces tokens not found in a predefined word dictionary—such as numerical values or file paths—with generalized placeholders (e.g., `[num]`, `[loc]`). The word dictionary used by the log parser is constructed during its initial development by extracting all distinct words from the collected log dataset. To eliminate ephemeral or context-specific tokens—such as process names—only words that appear at least three times across the dataset are included in the dictionary. This frequency threshold is intentionally chosen to filter out transient terms that are unlikely to contribute to meaningful pattern generalization. This abstraction process facilitates the extraction of canonical log patterns, a task made feasible by the structured nature of log generation systems, which typically rely on developer-defined templates.

Subsequently, the parser applies a similarity-based classification algorithm to group log patterns into higher-level events. Algorithm 2 presents a pseudocode implementation of our pattern classification method, based on an improved Minimum Edit Distance algorithm [44]. This modified algorithm serves as a similarity measure, calculating the minimum number of edits (deletions, insertions, and substitutions) required for two sentences to match. Specifically, it employs an inter-pattern similarity metric based on Minimum Edit Distance to assign structurally similar patterns to the same event category. Importantly, the parser also incorporates semantic filtering to distinguish between patterns with opposite meanings but similar syntactic structure. By leveraging a publicly available antonym dictionary [45], the parser can identify and segregate such cases—for example, distinguishing between "`container [name] down`" and "`container [name] up to state`"—and classify them into separate event groups.

Through these processes, the log parser maps each semantically similar log pattern to a corresponding event based on whether the patterns already classified into the same cluster can, on average, be transformed into the current pattern by modifying less than half of their elements. In operational terms, when a new log entry is processed by the log parser, it returns the event number associated with its matched pattern. For example, the log message "`Peer 5.5.5.1(3/1) from Down to Up`" is abstracted

Algorithm 2 Log Event Classification

Require: $LogPatternList$
Ensure: $EventList$
$EventList \leftarrow [\,]$
for $SingleLogPattern \in LogPatternList$ **do**
    $matched \leftarrow$ False
    for $SingleEvent \in EventList$ **do**
        $EditDistanceSum \leftarrow 0$
        for $Pattern \in SingleEvent$ **do**
            $EditDistanceSum \leftarrow Edit(SingleLogPattern, Pattern) + EditDistanceSum$
        end for
        if $\frac{EditDistanceSum}{|SingleEvent| \cdot |SingleLogPattern|} < 0.5$ **then**
            Append $SingleLogPattern$ to $SingleEvent$
            $matched \leftarrow$ True
            break
        end if
    end for
    if not $matched$ **then**
        Append $[SingleLogPattern]$ to $EventList$
    end if
end for
return $EventList$

to the pattern "`peer [num] from down to up`" and subsequently mapped to a predefined event number. When a log pattern that does not exist in the dataset used during the parser's construction is encountered, the system measures its similarity to existing patterns. If the pattern is found to be insufficiently similar to all existing event clusters, a new event identifier is created and assigned accordingly.

Using the classified log events, we computed the Log-TF-IDF values based on the normal log dataset. A threshold is then established according to the maximum Log-TF-IDF observed within the normal logs. In the actual RAG system, incoming logs are similarly processed through the log parser to extract structured log events. If the Log-TF-IDF score of a given event exceeded the predefined threshold, the log is identified as an error message. Only these filtered error messages are subsequently used as input queries to the RAG module.

3.3.2 Judge LLM

In addition, we applied filtering at the output stage of the RAG process. To evaluate the relevance of the retrieved documents, we introduce a Judge LLM, a language model distinct from the Main LLM, whose sole responsibility is to assess the validity and relevance of the RAG outputs. The Judge LLM is fine-tuned using our pre-collected log data alongside corresponding documentation that explains the associated error messages. This fine-tuning process enables the Judge LLM to learn the contextual relationships between logs and their explanatory resources within Kubernetes-based environments.

Consequently, the Judge LLM is capable of verifying whether the documents returned by the RAG module were genuinely relevant to the input error logs. In cases where the documents are deemed relevant—especially when the retrieved content is excessively long—the Judge LLM is further instructed to extract only the core information necessary to resolve the error. Conversely, if the Judge LLM determines that the retrieved document is unrelated to the input error, the RAG result is excluded from

the prompt to the Main LLM. This selective filtering mechanism is employed to reduce the likelihood of misleading or confusing the Main LLM with irrelevant information.

Algorithm 3 describes the complete RAG algorithm, which incorporates both input and output filtering mechanisms.

3.4 MAD (Multi-Agent Debate)

Multi-Agent Debate (MAD, [18]) is a technique in which multiple language model instances generate individual responses and reasoning processes, and subsequently engage in a discussion. This approach has been shown to enhance mathematical and strategic reasoning capabilities, improve the factual consistency of generated content, and mitigate common issues observed in modern LLMs, such as errors and hallucinations. In this thesis, we applied the MAD technique to improve the performance of our AI Agent further.

As with our adaptation of RAG, although most LLM-based techniques are designed for QA systems, our task deviates significantly in structure and purpose. Therefore, while our use of MAD shares foundational similarities with conventional approaches, it also exhibits important distinctions tailored to our setting.

We observed that even the same LLM exhibited inconsistent performance in VNF deployment tasks—sometimes generating correct code, and other times failing. Notably, once an LLM produced an initial attempt, it often became confined to that first solution space, making it difficult to generate substantially different alternatives in subsequent iterations. To overcome this limitation and broaden the scope of reference, we incorporated MAD into our system.

The central idea behind our MAD implementation is to inform the main LLM of alternative responses generated by other LLMs for the same prompt. By exposing the main LLM to diverse solutions, we encourage it to escape from its initial trajectory and leverage these references to refine its output. In our design, MAD is activated only when the main LLM fails to generate a runnable code. However, in preliminary exper-

Algorithm 3 RAG Algorithm

Require: $InputLogs, TF\_IDF\_Dict, EmbeddingModel, VectorKB, JudgeLLM$
Ensure: $RelatedDoc$

$ErrorList \leftarrow [\,]$
for $Log \in InputLogs$ **do**
    $LogEvent, IsNew \leftarrow LogParser(Log)$
    if $IsNew$ **then**
        $Log\_TF\_IDF \leftarrow CalculateTF\_IDF(LogEvent)$
    else
        $Log\_TF\_IDF \leftarrow TF\_IDF\_Dict[LogEvent]$
    end if
    if $Log\_TF\_IDF \geq TF\_IDF\_Threshold$ **then**
        Append $[Log]$ to $ErrorList$
    end if
end for
if $ErrorList$ is Empty **then**
    $ErrorList \leftarrow InputLogs$
end if
$InputVector \leftarrow EmbeddingModel(ErrorList)$
$MaxSim \leftarrow 0$
for $(Vector, Doc) \in VectorKB$ **do**
    $sim \leftarrow VectorSim(Vector, InputVector)$
    if $sim > MaxSim$ **then**
        $MaxSim \leftarrow sim$
        $QueriedDoc \leftarrow Doc$
    end if
end for
$Related, FilteredOutput \leftarrow JudgeLLM(QueriedDoc)$
if $Related$ and $MaxSim > Sim\_Threshold$ **then**
    return $FilteredOutput$
end if
return False

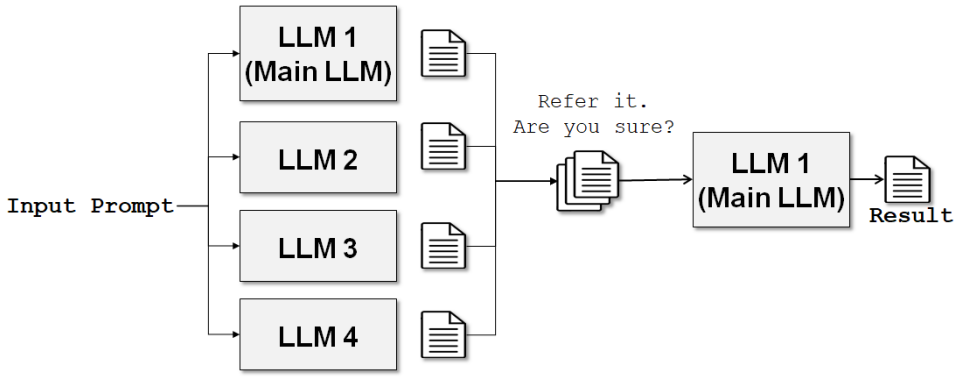

Figure 3.7: MAD Module

iments, we also explored the effectiveness of applying MAD from the initial attempt and conducted comparative performance analyses, which are presented in Section 5.2.

Fig. 3.7 illustrates the detailed workflow of this MAD process. Through extensive experimentation, we identified the four LLMs that consistently produced the most accurate code and designated them as peer agents for MAD. The MAD module stores their initial responses at the first attempt. When the main LLM fails to generate a runnable deployment, these pre-recorded responses are injected into the prompt as reference materials. This approach guides the main LLM to revise its output by consulting the outputs of other models, thereby enhancing the likelihood of recovering from failure and ultimately producing a valid solution.

IV. Implementation

In this chapter, we presents the implementation details of each module introduced in the Chapter III.

4.1 MOPs dataset generation

To construct an appropriate input dataset for the LLM, an MOP dataset was required. Initially, a sample MOP file was obtained from Samsung Electronics. Based on this sample, we decided to generate a synthetic dataset consisting of similar MOPs. Consequently, the structure and formatting of the generated MOPs closely follow the style of the original document provided by Samsung.

This thesis focuses on a defined scope involving the provisioning of Pods and the deployment of selected VNFs within a Kubernetes-based environment. In particular, we targeted the deployment of widely used VNFs, including a firewall (iptables), a load balancer (HAProxy [46]), a deep packet inspection engine (nDPI [47]), an intrusion detection system (Suricata [48]), and a network monitoring tool (ntopng [49]). Corresponding MOPs were generated to support the installation and configuration processes for each of these VNFs.

Given the impracticality of manually producing a large volume of procedural documents, we employed the GPT-4o-mini model [19] to automate content generation. The sample MOP served as the input prompt, and the LLM was instructed to generate detailed documentation describing the steps required to provision Pods in a Kubernetes environment, install the specified VNFs, and configure them to perform predefined tasks. To enhance the model's ability to generate relevant Kubernetes-specific procedures, we incorporated a directive prompt—"You are an expert in Kubernetes management"—to effectively guide the model toward utilizing domain-relevant

knowledge.

- Firewall: Allow only specific subnets, allow only specific ports
- Load Balancer: Load balance between the servers, redirect to a specific server
- DPI: Inspect specific subnet, block specific traffic
- Monitoring: Monitoring specific protocol
- IDS: Running basic detection rule

The MOPs were generated by iteratively modifying the actions for each VNF. Although the objective remained the same, we aimed to generate diverse MOPs by adjusting the prompts accordingly. LLMs tend to overlook some intentions when a large number of prompts are entered; therefore, during our experimentation, it was occasionally necessary to emphasize specific instructions. For instance, when configuring the installation of nDPI and blocking a specific port, some generated MOPs employed the '*ufw*' command for port blocking, which is not our intention. Each time an MOP was generated that did not align with our intended objectives, we adjusted the input prompt to prevent the generation of such MOPs. We generated both Korean and English MOPs to facilitate a performance comparison between using Korean and English MOPs. These prompt details can be found in the '*data_generating*' section of our published code [50]. The generated responses were saved in a Word document format using Python's '*docx*' library.

We manually verified all MOPs generated by GPT to ensure their accuracy, discarding any that were incorrectly generated. As a result, we retained a total of 60 MOPs for the experiment, comprising 30 Korean MOPs and 30 English MOPs. All input prompts except Kor MOPs used in the experiments were in English.

Example 4.1 is an example excerpt of a generated MOP.

MOP: Configuring nDPI
Document Control
Date: 2024-08-29
Version: 1.0
Purpose
This MOP outlines the steps to create a Pod in Kubernetes, install nDPI, and configure it to inspect packets from a specific subnet.
...
Prerequisites
- Access to a Kubernetes environment with appropriate permissions to create containers.
...
System Requirements for Container
- Operating System: Ubuntu 22.04 LTS
...
Step 1: Create a Pod in Kubernetes
1. Set Environment Variables
Ensure you have sourced your Kubernetes credentials:
$bash$
$kubectlrun < pod - name > –image = ubuntu : 22.04–restart = Never–requests =$ '$cpu = 2, memory = 2G$'
...

Example 4.1: Generated MOP Example

4.2 Prompt Input

Once the MOP dataset was prepared, we proceeded with experiments using the provided MOPs. As a starting point, we employed a basic prompt of the following form: "Based on the MOP below, implement a Python script using the Kubernetes library to automate the installation of the VNF described in the MOP." In addition, we appended example code after the prompt to support few-shot learning.

Although this represented the most fundamental prompt structure, several issues emerged during experimentation. For instance, the generated code contains an infinite loop when the VNF installation fails, continuously retrying the operation without termination. In other cases, despite clearly stating that credentials such as the Kubernetes access token were stored locally, the LLM would generate code that invoked stdin to request a password at runtime. We also observed code that attempted to split the configuration process across multiple containers, included only usage comments without any functional code, or defined placeholder function names with annotations such as "Insert VNF configuration code here."

```
from kubernetes import client, config
import time

def create_pod(pod_name, namespace):
    # Load kubeconfig file
    config.load_kube_config()
    v1 = client.CoreV1Api()
    image_name = input('Input the Image Name: ')
    # image name input
    container = client.V1Container(
        name="vnf-container",
        image=image_name,
        command=["/bin/bash"],
        args=[
            # VNF Configuration Code here
            # Build and install Firewall
```

```
            # Configure Firewall to block specific traffic
        ],
        resources=client.V1ResourceRequirements(
            requests={"cpu": "1", "memory": "256Mi"}
        ),
        security_context=client.V1SecurityContext(
           privileged=True)
    )
    metadata = client.V1ObjectMeta(name=pod_name, namespace
       =namespace)
    pod = client.V1Pod(metadata=metadata, spec=pod_spec)

    while True:
        # try untill succeed
        try:
            response = v1.create_namespaced_pod(namespace=
               namespace, body=pod)
            if response:
                return True
            else:
                return False
        except client.ApiException as e:
            print(e)
            return False
```

Listing 4.1: Bad Code Example (Putting configuration as comments, using infinite loop)

Such behaviors caused problems in our experimental environment, including indefinitely running code and unexpected resource usage. To mitigate these issues, we explicitly added constraints to the prompt that prohibited the generation of such patterns. These refinements were incorporated iteratively as new error types were identified throughout the experiments.

Furthermore, as will be discussed in detail in the following chapter, we designed

the experimental framework to ensure stability by requiring the Python code to put the whole code inside the 'create_pod' function. These functions are sequentially imported and executed during the experiment. Each function is designed to return True upon successful execution and is defined to receive relevant parameters as input.

We explicitly specified these structural requirements within our predefined prompt format. During prompt generation, the prompt format is combined with the corresponding MOP to form the complete input to the LLM (Fig. 3.3). The resulting prompt guides the model to produce code adhering to this modular structure. The predefined prompt format for Python coding used in our thesis is presented below.

```
You are a Kubernetes cloud expert.

Please write Python code that creates a Kubernetes Pod
   and installs a VNF inside it, taking the following
   points into consideration.
You don't have to explain your code. Keep your answer
   code-focused and as simple as possible.
I will provide you with a Method of Procedure (MOP)
   that describes the process of installing a Pod in
   Kubernetes and configuring a VNF on that Pod. Based
   on this MOP, you must write a single, self-contained
    Python function with the following strict
   requirements:

1.  The entire logic for creating the Pod and
   configuring the VNF must be contained within one
   function only; do not split the logic into multiple
   functions or files.

2.  The function must be named 'create_pod'.

3.  The 'create_pod' function must accept exactly three
    input parameters: 'pod_name', 'namespace', and '
   image_name'.

4.  The function must return True if the Pod is created
    and the VNF is configured successfully; otherwise,
   it must return False.

5.  Do not include any additional parameters or helper
   functions; everything must be implemented inside the
    create_pod function.

6.  Don't put any kind of infinity loop in the code.
```

```
7.  The Kubernetes configuration file is in its default
     path. So, to load the configuration, use '
    load_kube_config' instead of 'load_incluster_config
    '.

8.  Don't put usage or example in the code block.

9.  Instead of using the cluster's default DNS settings
    , manually set the DNS to 'DNS_IP'.

10. Put 'sleep infinity' command, so that the container
     doesn't get killed.

12. Since systemctl cannot be used in containers, even
    if the MOP instructs to install the VNF using
    systemctl, an alternative method like running with
    the daemon option must be found.

12. Don't use stdin to get any kind of password.

13. Don't make 'host_name' option True in Pod creation
    step.

14. Configure the VNF in one container.

Please ensure that you follow these instructions
    exactly and do not deviate from the specified
    function name, parameter list, or return value.

Here is the MOP:
```

Listing 4.2: Prompt Format for Python

In addition, we employed a different prompt format for retry attempts. When a retry is triggered, the system supplies not only the original prompt but also relevant contextual information, such as execution logs in the case of general failures, RAG outputs when RAG is used, or MAD results when MAD is applied. To accommodate these scenarios, we defined an alternative prompt format specifically for retry cases.

This alternative format is not overly complex, but structured to convey error feedback and supporting context to the LLM. A typical template follows the structure: "When executing your code, the following error occurred: [*error message*]. The following document(s) may be relevant to this error: [*RAG output*]. Please revise your code by referring to this information." This retry-specific prompt enables the LLM to

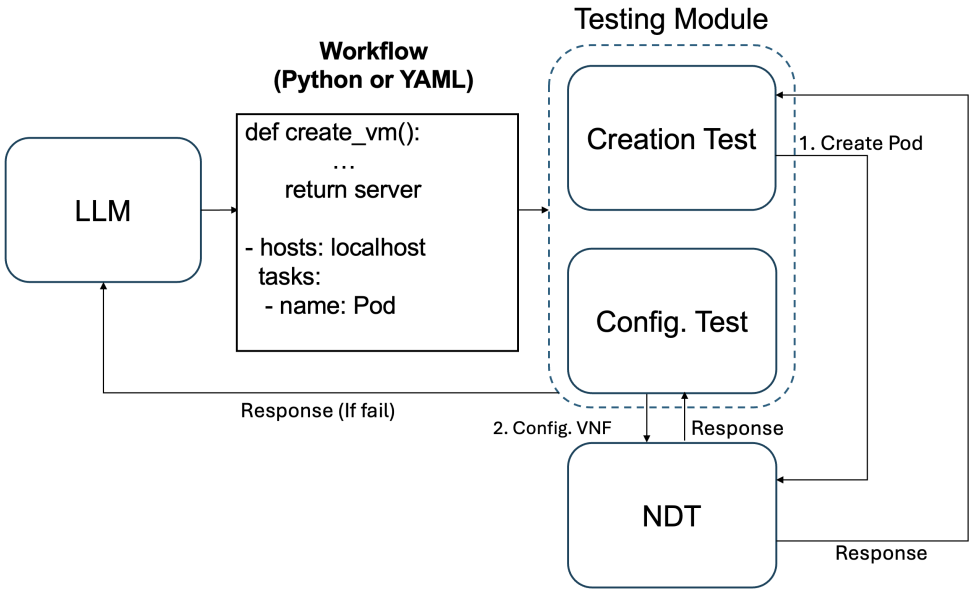

Figure 4.2: Detailed Architecture of Testing LLM-generated Workflow

revise its previous response with improved contextual awareness.

4.3 Testing Module and NDT

Fig. 4.2 illustrates a detailed structure of the testing module and NDT. The testing module is designed to validate the workflow generated by the LLM prior to its deployment in the actual network environment. Specifically, it verifies whether the generated code is syntactically and semantically correct by executing it within the NDT. This module encompasses three key stages: executing the code in the NDT environment, validating its behavior, and preventing the execution of erroneous configurations that could cause potential failures during deployment.

4.3.1 Network Digital Twin (NDT)

The NDT was implemented on a Kubernetes-based infrastructure. We deployed Kubernetes version 1.29.15, configuring a cluster composed of one Master node and

four Worker nodes. Each node operated on Ubuntu 22.04 LTS.

The testing module and the LLM server were deployed separately from the NDT to ensure modularity. The server running the testing module was granted access to the Kubernetes cluster via *kubectl*, thereby enabling direct interaction with the cluster for both Ansible- and Python-based testing procedures. This setup allowed the execution of Pods and the configuration of VNFs directly within the Kubernetes environment.

To prevent interference with the default CNFs in the NDT, a dedicated namespace was created. The testing module was explicitly configured to operate only within this namespace. During experimentation, all Pods within the namespace were periodically removed to maintain a clean testing environment.

4.3.2 Testing Module

Although the Python- and Ansible-based modules were implemented separately, their overall structure is similar. Therefore, we first describe the Python-based implementation in detail, and subsequently explain only the components that differ in the Ansible-based approach. In the case of a Python-based testing module, it saves the parsed Python code as a '.py' file and executes it to verify whether it successfully communicated with the NDT environment and automated the creation of containers and VNFs.

To facilitate testing, we instructed LLM to put the entire logic as 'create_pod' function. The 'create_pod' function takes the Pod name, namespace, and image name as input parameters. It returns True upon successful Pod creation and False if the creation fails for any reason. During the Python code testing process, although we specified in the input prompts that only function definitions should be included without examples in the Python code, there were instances where the LLM would still generate top-level code with example usage. Additionally, when importing the generated code, if the LLM did not include a check for '*__main__*', the top-level code would execute automatically. On rare occasions, we observed that this resulted in an infinite loop,

particularly when the LLM inadvertently included logic to continuously create Pods. To prevent such occurrences, we enforced a practice of wrapping top-level code within a *__main__* block, ensuring it would only run when explicitly invoked.

In both the Python- and Ansible-based workflows, the code generated by the LLM is executed with explicit input parameters, including the Pod name, namespace, and image name. This allows us to maintain control over the Pod creation process and ensures consistency across experiments.

For the container image, we employed a vanilla Ubuntu Docker image (version 20.04) pre-equipped with basic Ubuntu libraries. This choice enables most VNFs to be installed directly via apt without requiring additional dependencies or custom image preparation.

The overall process in the Ansible-based module follows the same pipeline as in the Python-based implementation. In the Ansible setup, the test module parses the YAML code from the LLM's response and saves it. This YAML code is then executed using the Ansible-runner Python library [51], which allows Ansible playbooks to be invoked programmatically.

In the case of Ansible, it is necessary to explicitly specify the target server or node on which each command should be executed. However, the LLM occasionally generates workflows in which this target is mistakenly set to localhost. In such cases, the VNF configuration is executed on the server running the testing module rather than on the intended Pod. This misconfiguration can lead to critical issues, such as inadvertently modifying the host environment, for example, by installing a firewall on the testing server itself, thereby compromising the AI Agent's experimental setup.

To address this issue, we implemented a rule-based detection mechanism that identifies and blocks workflows containing such erroneous target specifications before they are executed.

Kubernetes does not provide an API to verify whether all commands passed during Pod creation have completed execution. Since the code is executed exactly as generated by the LLM, it is not possible to predetermine which commands will be

executed within the Pod, making it difficult to inspect command completion through direct Kubernetes access.

Through empirical observations across multiple test cases, we found that correctly configured VNFs typically complete Pod creation and internal setup within 150 seconds. Based on this finding, we adopted a simple timeout-based strategy: after a Pod is successfully created, the system waits for 150 seconds before proceeding to the next step. During this period, we continuously monitor the Pod's status to ensure that no errors occur. If the Pod remains in a healthy state throughout the waiting period, we assume that all VNF-related commands have been executed successfully. As will be detailed in a Chapter V, this approach led to a consistent Pod creation time of 150 seconds, impacting the overall processing time of the AI Agent.

And now, the test module takes a verification step to ensure that the deployed VNF operates as intended. For instance, for the firewall, the MOP specified configuring '*iptables*' to only allow traffic through certain subnets or ports. We verified this by running the command '*iptables -L -v -n*' and checking for the presence of 'DROP' in the output. If an incorrect subnet or port was blocked, the process would fail, signaling an issue. As will be discussed in the Results section, many LLMs were unable to resolve this issue, demonstrating low performance effectively.

For other VNFs (Suricata, nDPI, ntopng, and HAProxy), we used the '*systemctl*' command to verify that the service was running. Although the MOP details how to configure each VNF fully, the LLM-generated code's performance declined significantly when tasked with verifying every detail of the configuration. Consequently, in this thesis, we set the baseline for passing as the successful operation of the VNFs, without requiring the configuration to be flawless. Additionally, for HAProxy, we conducted a validation step by executing a command to verify the validity of the HAProxy rule file and inspecting the results. Developing LLM models that can generate fully accurate configurations remains part of our future research objectives.

The test module was designed to inform the LLM of any issues that arise during workflow execution. This mechanism is implemented consistently across both the

Table 4.1: Test Module Status Codes and Description

Status Code	**Description**
0	Run well
1	Can't find code
2	VNF does not work as intended. Related message will be out
10	Parsing false, check the format
11	Code should be updated to run only inside Kubernetes
12	Infinite loop detected
13	Error while import create_pod
14	create_pod does not exist, Error message may out
20	Error occurs in Pod. Error message is out
21	Pod fall into error state. State information is out
22	Exception occurs in my code
23	Ansible failed. (status, output) will be out
30	Error occurs while configuring VNF. Error logs will be out
31	Pod name doesn't exist in Namespace
32	Container exit, need to add 'sleep infinity'
33	Container failed with error
41	Skipped task by incorrect host name
43	Error while searching Pod
51	Function didn't return True
90	Running Timeout. stdout may out
91	Container ready timeout

Python- and Ansible-based modules. Depending on the point of failure during execution, the testing module returns a predefined status code that reflects the nature of the encountered issue. This allows the AI Agent to infer what went wrong during internal execution.

Table 4.1 summarizes the status codes we defined and their corresponding meanings. These codes provide insights into the various failure scenarios that may occur during LLM-driven VNF deployment. In certain cases, additional error messages were retrieved from the NDT, internal libraries such as ansible-runner, or the Pod itself. These messages were returned alongside the status codes, enabling the AI Agent to perform more detailed analysis based on the reported context.

4.4 Retrieval Augment Generation (RAG)

To implement RAG, we first needed to construct a comprehensive knowledge base. As described in the design phase, we collected two primary sources of information: official documentation and Stack Overflow question threads. Specifically, we crawled a total of 538 documents from Kubernetes official documentation, 2,676 documents from Ansible documentation, and 1,552 question threads from Stack Overflow, resulting in a combined corpus of 4,766 documents. Notably, the Ansible database was not utilized when the workflow did not involve Ansible. In the case of official documentation, we separated explanations of individual functions—when provided in a single document—into distinct entries to facilitate finer-grained retrieval. For Stack Overflow, we used keywords such as “Kubernetes”, “Ansible”, “nDPI”, and “Firewall” to identify relevant questions. Additionally, during experimentation, if an encountered error message yielded useful documents not already included in our corpus, we manually incorporated them into the knowledge base.

Following the construction of the knowledge base, we developed a vector-based retrieval system. Each document was stored in an SQLite database with associated fields including title, URL, content, and, when available, the question text. To en-

able vector-based retrieval, we employed a widely-used sentence embedding model, *all-MiniLM-L6-v2* [52], which is commonly utilized in RAG implementations. In addition, we used the same model by fine-tuning with our log data and conducted a comparison experiment. For each document, a vector representation was generated using its title and question text, if available, as input, and the resulting vectors were stored in a Chroma vector database [53].

To evaluate the performance of our RAG system, we executed AI Agents in real scenarios and collected logs from 129 instances where errors occurred. We manually classified the error messages from each log entry, thereby also constructing a dataset of non-error (normal) log traces for comparison. Based on this dataset, we computed Log-TF-IDF values to quantify the significance of terms within log sequences. Using these logs, we built both a pattern dictionary and an event dictionary and calculated the Log-TF-IDF score for each event. The analysis showed an average Log-TF-IDF score of 3.54, with a maximum of 5.46. Leveraging this observation, we defined a threshold of 5.46 for log-based input filtering in our RAG pipeline—logs with a score exceeding this threshold were classified as error-indicative inputs.

Furthermore, we utilized the generated log dataset to train a Judge LLM. For each error scenario, we manually tagged the corresponding documentation related to the log traces. These document titles were then provided to Judge LLM along with the associated log sequences to enable supervised fine-tuning. Specifically, we fine-tuned a Phi-4 model using the PEFT library [54] with LoRA technique, allowing the model to learn the mapping between error patterns and relevant documentation.

With these components in place, we established a fully operational RAG pipeline. To evaluate the performance of the system, we conducted comparison experiments for each module of RAG using the same set of log-document pairs that had been utilized for fine-tuning. The experimental results and detailed evaluation will be presented in the following chapter.

All source code used for implementing the AI Agent in this thesis is publicly available at the citation provided in [50].

V. Experiment and Evaluation

5.1 Experiment

We evaluated our proposed LLM-based AI Agent framework by experimenting with various combinations of LLMs and MOPs. Through these experiments, we assessed each LLM's capability to generate the correct workflow. To compare the performance of various LLM models (ChatGPT-4-o, o3-mini [19], Llama3.3:70b [21], Qwen2.5-coder:32b [28], DeepSeek-Dist:70b [23], Gemma3:27b [22], QwQ:32b [24], Phi4:14B [25] and Mistral:7b [26]) we conducted a series of experiments. Except for the two GPT models, all models were executed locally on our machine using *Ollama* [55]. Since GPT models are not open to the public, the two GPT models were accessed via API requests. The offline models ran on a machine equipped with dual Intel Xeon Silver 4216 CPUs, 128GB of RAM, and an Nvidia Quadro RTX A6000 GPU with 48GB of VRAM.

QwQ and o3-mini are models specialized for reasoning tasks, while Qwen2.5-coder is tailored for code generation. Following the recent trend of developing models optimized for specific capabilities such as reasoning or programming, we adopted these specialized models in our study. Through preliminary evaluations, we excluded some models with relatively low performance from the final experiments.

In the case of the DeepSeek model, although the original model is publicly available, its size (671B parameters) was impractically large for our experimental setup. Therefore, we employed a distilled version based on the LLaMA 3 model. Distillation refers to the process in which a smaller model is trained to mimic a larger model by learning from its input-output behavior through supervised learning.

In each experiment, if the LLM-generated code produced errors or failed, the LLM was given two additional opportunities. We calculated the success rate when

Table 5.1: List of LLMs Used in Evaluation

Model	**Description**
ChatGPT-4-o [19]	OpenAI GPT-4o model
o3-mini [19]	OpenAI reasoning model of GPT series
Llama3.3 [21]	Meta AI Llama 3.3 model, 70B parameters
Qwen2.5-coder [28]	Alibaba Qwen-2.5 based coding specific model, 32B parameters
DeepSeek-Dist [23]	DeepSeek DeepSeek-R1 based distilled model in Llama model, 70B
Gemma3 [22]	Google Gemma 3 model, 27B parameters
QwQ [24]	Alibaba reasoning model of Qwen series, 32B parameters
Phi4 [25]	Microsoft Phi-4 model, 14B parameters
Mistral [26]	Mistral AI Mistral model, 7B parameters

successful in one try and after successful retries to see if the performance increases through retries.

Each experiment was conducted once per setting, where the AI Agent generated both Python- and Ansible-based workflows for 60 MOPs per LLM. Success rates were measured for each deployment. Even in cases where VNF installation failed, Pod creation success was recorded separately if the Pod was successfully instantiated.

Success rates were also evaluated separately for MOPs written in Korean and English. In addition, the execution time of each module within the AI Agent was individually measured to analyze the overall processing time—from the initial prompt input to the final confirmation of VNF installation—and to identify which stages contributed most to total latency.

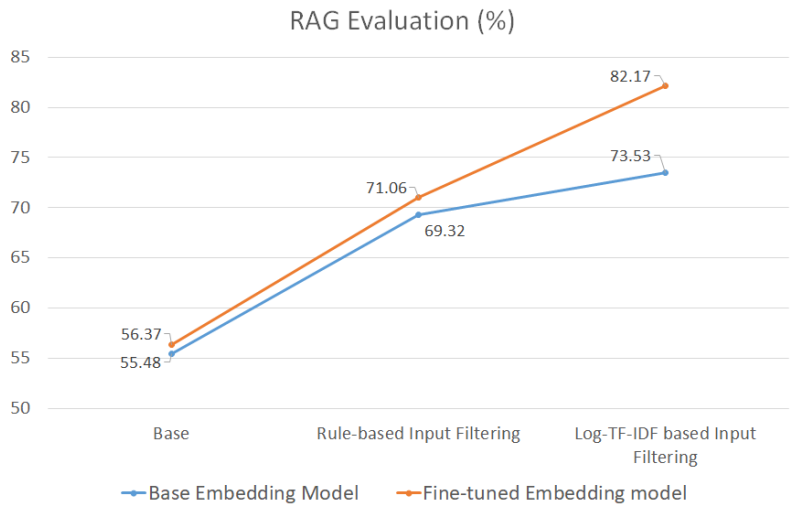

Figure 5.1: RAG Evaluation Test Results

5.2 Evaluation

5.2.1 Retrieval-Augmented Generation (RAG)

We first conducted experiments to evaluate the performance of the RAG module in isolation. Specifically, we constructed a dataset consisting of 129 log–document pairs and assessed whether the RAG system could retrieve the correct document corresponding to each log. A retrieval was considered successful if the cosine similarity between the title of the retrieved document and the title of the ground-truth document exceeded 0.8, acknowledging multiple answers if there is another similar document

To compare performance, we tested both a baseline embedding model and a fine-tuned version trained on our custom log dataset. Additionally, we evaluated the effectiveness of our proposed Log-TF-IDF-based input filtering mechanism by comparing it against a rule-based filtering approach. In the rule-based method, logs containing the keywords ‘error’ or ‘Err’ were used directly as queries for the RAG system.

Fig. 5.1 summarizes the results of the experiment. Experimental results confirmed that, as intended, the fine-tuned embedding model combined with input filter-

ing yielded higher accuracy in retrieving ground-truth documents. In particular, the Log-TF-IDF-based input filtering method significantly outperformed the rule-based approach in effectively narrowing down relevant inputs.

However, even in the best-performing setup, the retrieval accuracy did not exceed 90%. Upon further analysis, we identified that excessively long input logs were a primary contributing factor. This issue was especially prevalent in logs generated by Ansible, which often include verbose metadata such as executed commands, node identifiers, container information, and image references. In such cases, even with our log parser applied, the input length remained substantially longer than average, typically exceeding 70 words, compared to fewer than 20 words for most logs. This excessive verbosity negatively affected the output pattern and degraded retrieval performance based on vector similarity.

These observations suggest the need for specialized log parsing techniques tailored specifically for Ansible logs to isolate semantically meaningful content. Developing a dedicated parser for Ansible is thus a promising direction for future work.

Building upon these findings, we integrated both the fine-tuned embedding model and the Log-TF-IDF-based input filtering mechanism into our AI Agent. To assess their impact, we measured VNF deployment success rates with and without input filtering, enabling a comparative evaluation of system performance.

5.2.2 Multi-Agent Debate (MAD)

Before integrating MAD into the AI Agent, we conducted a preliminary experiment to assess its potential effectiveness. Unlike the standard MAD approach, which executes multiple LLMs in parallel and directly evaluates their responses in the NDT, we explored a variation in which the initial responses from multiple LLMs were passed to the main LLM, allowing it to review and synthesize a final answer.

We employed Llama 3.3 as the main LLM, configured with 3-shot learning. As in previous experiments, failure cases were followed by a RAG-based review of the

associated error messages. The helper LLMs used for peer response generation were Qwen2.5-coder:32b, Gemma3:27b, QwQ:32b, and Phi4:14b, selected for their balance of response speed and generation quality.

Experimental results showed that, for Python-based workflows, the baseline success rate of 68.3% for Llama 3.3 increased to 72.4% when applying this form of MAD. For Ansible-based workflows, the success rate improved from 66.7% to 70.3%. While the improvements were modest, they were consistent across both workflow types.

However, the MAD approach incurred a substantial latency cost. The average response time increased from 160 seconds to approximately 939 seconds—nearly a fivefold increase—due to the concurrent invocation of four LLMs. Given this significant time-performance trade-off, we decided to reduce the number of helper models in the final experimental setting to three: Gemma3:27b, QwQ:32b, and Phi4:14b.

Furthermore, as described in the design chapter, the MAD module in our final implementation is only invoked when the initial response from the main LLM fails. In such cases, the agent consults responses from the helper LLMs to revise and improve the original output.

5.2.3 AI Agent

After evaluating the performance of each module, we proceeded to assess the overall AI Agent by integrating all components. We conducted incremental experiments in which each module was added sequentially, while also varying the number of few-shot examples. For each deployment, we measured both the creation and configuration success rates, as well as the total processing time.

Processing time was measured only for cases in which the configuration succeeded; thus, instances where the configuration entirely failed were excluded from the time analysis. Table 5.2 and Table 5.3 present the success rates and processing times for both Python- and Ansible-based workflows. As expected, the addition of more modules generally led to higher success rates but also resulted in increased processing

times. It is important to note that, unlike the other models, GPT's responses were obtained via a paid API, making a direct comparison of processing time infeasible. This limitation should be considered in the interpretation of the subsequent processing time analysis.

Starting from 3-shot learning, most models—excluding Mistral—achieved between 95% and 100% success rates in the creation phase; therefore, we excluded creation performance for those models in later stages. Due to space limitations, processing time is reported for a subset of representative setups only. Success rates are expressed in percentages, while processing times are reported in seconds.

Overall, Ansible-based workflows exhibited lower performance compared to their Python counterparts. We attribute this to the higher syntactic complexity of Ansible and the relative scarcity of Ansible-related data in the training corpus of most LLMs, which are generally more exposed to Python during pretraining.

The highest success rates for both environments were achieved when all enhancement techniques—TF-IDF-based RAG, Judge LLM-based RAG output filtering, and MAD—were applied. With this setup, Python and Ansible workflows reached success rates of 86.7% and 73.3%, respectively.

We also observed noticeable performance differences among LLMs. In general, GPT models, Qwen2.5-coder, and Gemma3 demonstrated consistently strong performance. Although processing time comparisons with GPT models are not directly meaningful due to the use of API-based access rather than local execution, a general trend was observed where models with larger parameter counts required longer processing times.

Interestingly, even among models with similar parameter sizes, those with higher success rates occasionally exhibited increased processing times. We hypothesize that this is due to more sophisticated reasoning processes triggered during error detection and correction. High-performing models may spend additional time generating revised outputs or making multiple attempts, thereby increasing the average execution time.

In some cases, the condition with fewer modules exhibited slightly higher perfor-

Table 5.2: Success Rate (%) and Processing Time (s) Results for Python

Model		GPT-4o	o3-mini	Llama3.3	Qwen2.5-coder	DeepSeek-Dist	Gemma3	QwQ	Phi4	Mistral
Parameter (B)		-	-	70	32	70	27	32	14	7
No RAG, 0-shot	**Cr. Suc.**	66.7	75	75	85	46.7	70	85	30	18.3
	Conf. Suc.	**10**	**15**	**0**	**0**	**8.3**	**0**	**18.3**	**13.3**	**0**
	Proc. Time	195	193	-	-	377	-	386	234	-
No RAG, 3-shot	**Cr. Suc.**	100	100	100	100	96.7	100	100	93.3	48.3
	Conf. Suc.	**43.3**	**55**	**61.7**	**46.6**	**51.7**	**50**	**45**	**56.6**	**15**
RAG, 0-shot	**Cr. Suc.**	81.6	88.3	86.7	91.7	88.3	86.7	86.7	70	28.3
	Conf. Suc.	**46.6**	**51.7**	**43.3**	**48.3**	**61.7**	**53.3**	**40**	**45**	**6.7**
	Proc. Time	183	188	348	295	398	253	367	280	213
RAG, 1-shot	**Cr. Suc.**	95	100	86.7	95	91.7	100	100	96.6	45
	Conf. Suc.	**63.3**	**61.7**	**70**	**55**	**60**	**33.33**	**70**	**50**	**20**
RAG, 3-shot	**Conf. Suc.**	**58.3**	**66.6**	**71.6**	**60**	**50**	**65**	**58.3**	**41.7**	**28.3**
	Proc. Time	178	187	343	298	401	243	453	278	221
TF-IDF RAG, 3-shot	**Conf. Suc.**	**73.3**	**71.7**	**68.3**	**65**	**68.3**	**70**	**66.7**	**48.3**	**31.7**
	Proc. Time	181	183	351	252	403	243	480	224	234
TF-IDF RAG, Judge, 3-shot	**Conf. Suc.**	**83.3**	**76.7**	**71.7**	**78.3**	**68.3**	**76.7**	**68.3**	**53.3**	**35**
	Proc. Time	223	203	367	268	410	265	470	272	238
TF-IDF RAG, Judge, 3-shot, MAD	**Conf. Suc.**	**85**	**78.3**	**73.3**	**86.7**	**75**	**83.3**	**71.7**	**56.7**	**38.3**
	Proc. Time	523	484	523	559	694	526	686	465	477

Table 5.3: Success Rate (%) and Processing Time (s) Results for Ansible

Model		GPT-4o	o3-mini	Llama3.3	Qwen2.5-coder	DeepSeek-Dist	Gemma3	QwQ	Phi4	Mistral
Parameter (B)		-	-	70	32	70	27	32	14	7
No RAG, 0-shot	**Cr. Suc.**	68.3	73.3	33.33	66.6	55	70	48.3	26.6	15
	Conf. Suc.	**8.3**	**13.3**	**0**	**6.7**	**0**	**13.3**	**3.3**	**0**	**0**
	Proc. Time	167	174	-	296	-	223	330	-	-
No RAG, 3-shot	**Cr. Suc.**	93.3	100	91.6	96.6	98.3	100	100	78.3	40
	Conf. Suc.	**51.7**	**55**	**48.3**	**53.3**	**53.3**	**56.7**	**48.3**	**23.3**	**0**
RAG, 0-shot	**Cr. Suc.**	83.3	85	78.3	83.3	81.6	88.3	78.3	70	41.6
	Conf. Suc.	**28.3**	**35**	**43.3**	**45**	**38.3**	**48.3**	**50**	**25**	**8.3**
	Proc. Time	171	176	336	438	421	210	240	216	198
RAG, 1-shot	**Cr. Suc.**	96.6	100	98.3	100	100	100	93.3	91.6	48.3
	Conf. Suc.	**61.7**	**63.3**	**63.3**	**53.3**	**56.7**	**55**	**50**	**38.3**	**28.3**
RAG, 3-shot	**Conf. Suc.**	**66.7**	**65**	**66.7**	**58.3**	**60**	**65**	**63.3**	**50**	**28.3**
	Proc. Time	180	204	235	339	442	209	375	240	192
TF-IDF RAG, 3-shot	**Conf. Suc.**	**68.3**	**68.3**	**68.3**	**55**	**61.7**	**65**	**63.3**	**53.3**	**38.3**
	Proc. Time	183	187	243	258	398	242	243	252	197
TF-IDF RAG, Judge, 3-shot	**Conf. Suc.**	**63.3**	**71.7**	**68.3**	**66.7**	**58.3**	**68.3**	**68.3**	**41.7**	**43.3**
	Proc. Time	186	227	250	369	478	223	390	263	210
TF-IDF RAG, Judge, 3-shot, MAD	**Conf. Suc.**	**71.7**	**68.3**	**65**	**68.3**	**68.3**	**73.3**	**66.6**	**51.6**	**43.3**
	Proc. Time	457	509	507	635	735	478	656	523	460

mance than those with more modules. We attribute this to the fact that each experiment was conducted only once. For more rigorous validation, repeated trials over the same dataset with statistical averaging would be necessary. However, due to time constraints, all experiments in this thesis were performed only once per setting. Despite these limitations, the overall trend aligns with our expectations: the addition of more modules consistently led to improved performance.

Fig. 5.2 presents a detailed comparison of model-wise performance under the best-performing setup of the AI Agent, where all enhancement techniques—3-shot learning, RAG with TF-IDF, Judge LLM-based RAG output filtering, and MAD—were applied. For each model, we distinguish between cases where creation or configuration succeeded on the initial attempt and those that succeeded after retrying. This separation allows us to observe the impact of our proposed RAG and MAD modules, which contributed significantly to improving success rates during the retry phase.

In contrast, in our previous study [56], we conducted a performance comparison of VNF configuration in an OpenStack environment without applying RAG or MAD. The results indicated that there was no significant difference between the initial success rate and the success rate after retries. suggesting that our proposed mechanisms meaningfully enhance the LLM's self-healing capabilities.

We also investigated whether performance would vary depending on the language of the MOP, specifically comparing English and Korean inputs. Contrary to our expectations, no substantial difference was observed. Although previous reports indicated that some models developed in China, such as Qwen, perform well on Korean inputs, our experiments showed that most models achieved comparable success rates for both Korean and English MOPs.

We conducted an additional performance analysis under the best-performing setup: Python-based workflows with 3-shot learning, Log-TF-IDF input filtering, Judge LLM-based RAG output filtering, and MAD. Specifically, we compared success rates across different VNFs by aggregating the results over all models and evaluating which VNFs were successfully configured in the full MOP dataset.

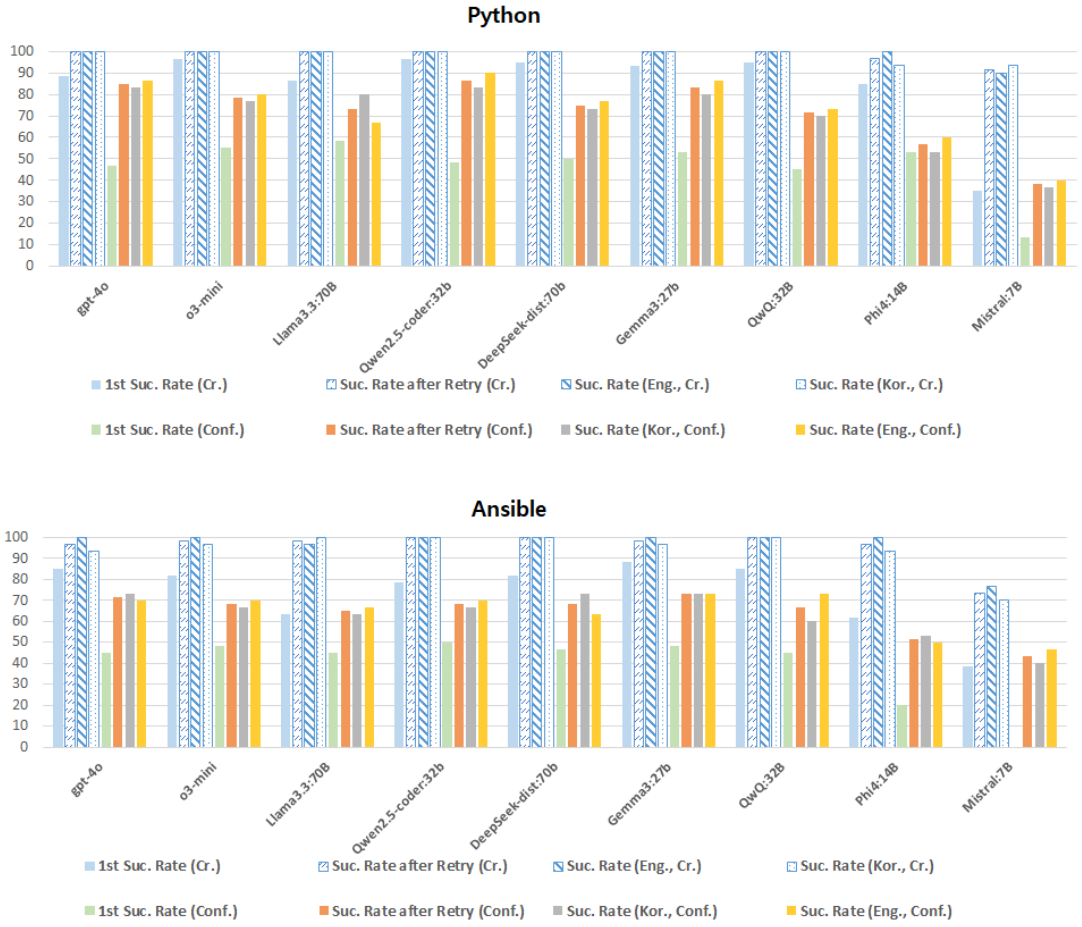

Figure 5.2: 3-Shot, RAG-TF-IDF, Judge, MAD Results

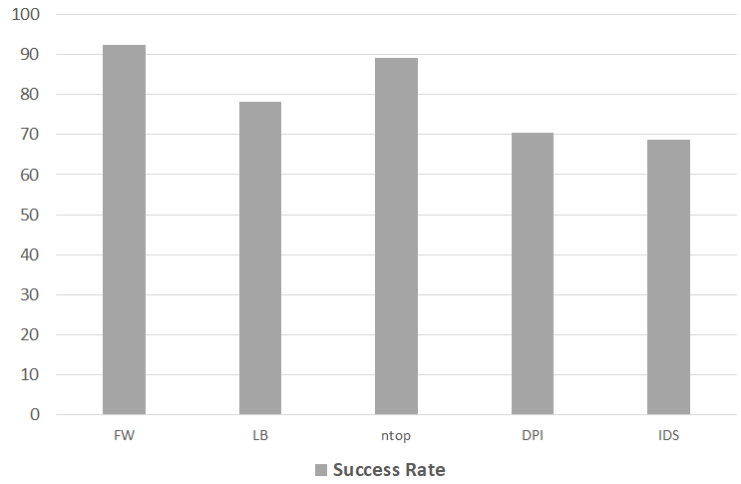

Figure 5.3: Success Rates (%) by VNFs

The results showed that Firewall achieved the highest success rate at 92.5%, whereas IDS exhibited the lowest at 68.75%. This disparity can be attributed to the relative simplicity of installing VNFs such as Firewall and ntop, in contrast to the more complex installation processes required for Load Balancer and IDS. In particular, the installation of these VNFs often relies on apt, which can introduce intricate dependency issues that are challenging for LLMs to resolve.

To mitigate this, we enriched the RAG knowledge base with documentation related to *'apt'* installation procedures, which led to incremental performance improvements. Nonetheless, the consistently lower success rates for complex VNFs underscore the need for future research focused on enhancing LLM capabilities for handling more intricate installation workflows.

We further conducted a comparative analysis of the execution time required by each module (Fig. 5.4). Fig. 5.5 illustrates the per-module execution time across different LLMs, focusing on three representative models. The x-axis represents the cumulative time incurred as modules are incrementally added, while the y-axis indicates the corresponding gain in success rate.

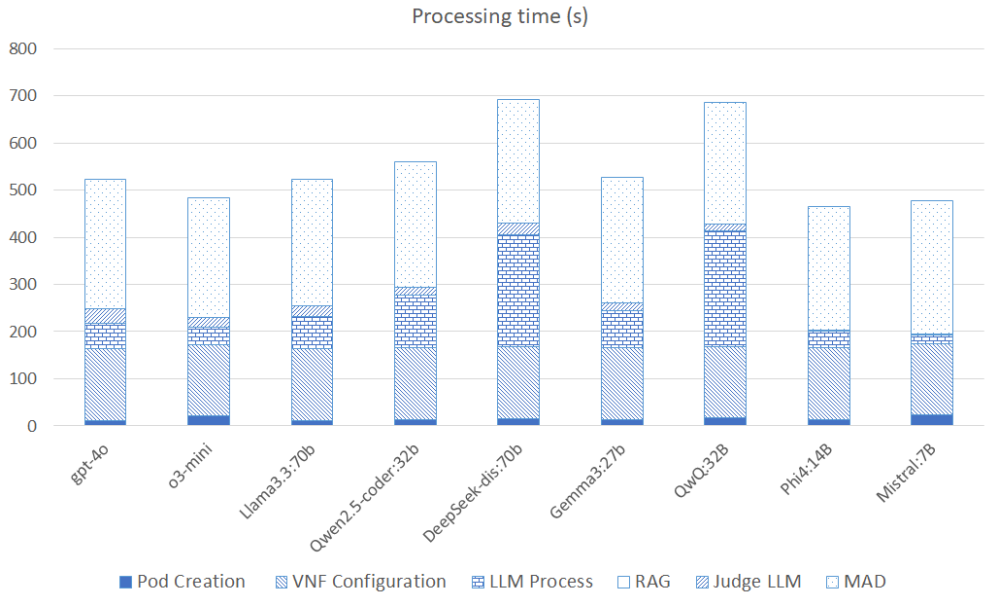

Figure 5.4: Comparison of Execution Times (s)

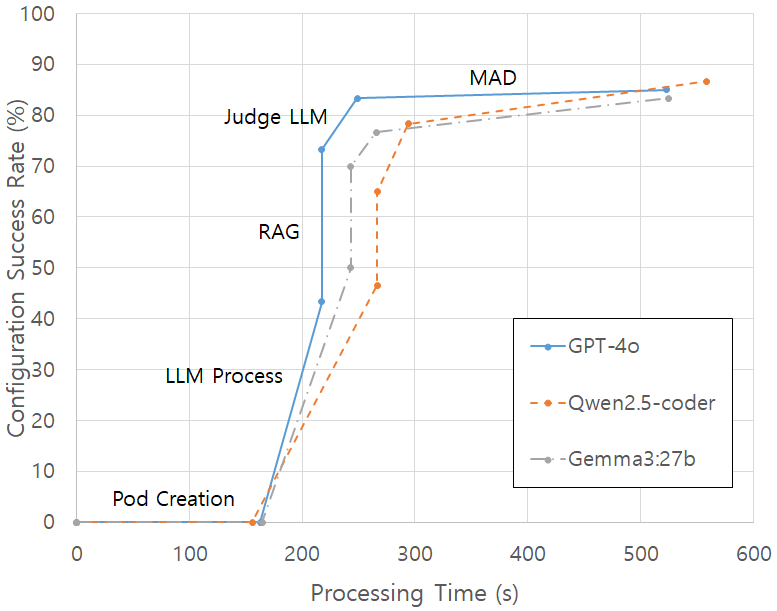

Figure 5.5: Configuration Success Rate (%) Line Chart by Processing Time (s)

As noted in the design section, Kubernetes does not offer a direct method to verify whether VNF configuration commands have completed. Therefore, we applied a uniform 150-second waiting period following Pod creation in all cases. As a result, the time spent on Pod creation and configuration remains roughly consistent across all settings, and we instead focus on the time consumed by LLM processing and other supporting modules.

Excluding ultra-lightweight models such as Mistral, LLM processing times generally ranged between 30 and 240 seconds. Notably, lightweight models exhibited significantly faster response times than server-based models like GPT. This is likely due to GPT models spending more time on complex reasoning. Nevertheless, their absolute response times remained relatively low, averaging around 30–50 seconds. The slowest model was DeepSeek-Dist (70B), with an average of 237 seconds, while Llama 3.3 (also 70B) achieved a much faster average of 69 seconds. Although we did not design the Llama 3.3 model ourselves, this discrepancy may stem from reduced time spent on reasoning, which may also explain its comparatively lower performance.

Models like Gemma3 demonstrated a favorable trade-off, achieving high performance with fewer parameters and an average response time of 78 seconds, highlighting its efficiency advantage.

The time required for RAG was relatively consistent across models. Querying alone took approximately 0.05 seconds, while applying output filtering increased the total RAG time to around 20–30 seconds. Importantly, the performance-time curve (Fig. 5.5) shows that even this small increase in processing time (via RAG) results in a substantial improvement in success rate, underscoring the significant utility of RAG in the system. In contrast, output filtering is more time-intensive due to its reliance on a separate LLM. This suggests a direction for future work: exploring alternative, non-LLM-based filtering methods to reduce latency.

MAD incurred the highest time cost, with an average of approximately 260 seconds. This was consistent across all models, as the MAD module operates independently of the specific LLM used in earlier stages. In many cases, the time consumed by

MAD alone is comparable to the combined time of all preceding modules. As such, the decision to employ MAD should be left to the user, depending on the available resources and required deployment reliability. As also shown in Fig. 5.5, the performance gain per additional unit of time for MAD is relatively low, in contrast to the steep gain observed with RAG.

In conclusion, LLM models without our AI agent (i.e., zero-shot without RAG) achieved an average accuracy of around 10%, whereas the application of our AI agent resulted in performance levels reaching approximately 70–80%. This demonstrates the significant effectiveness of our AI agent in automating the deployment of VNF using LLM.

Given the high human error rate typically observed in this domain, our results suggest that incorporating our AI agent can enable highly accurate automation of the VNF deployment process, with a substantially reduced error rate. While it would be ideal to compare our results with human error rates or processing times from manual configurations, such data is currently unavailable. Nevertheless, as discussed in Chapter I, the current state of network management often involves manual configurations that are prone to high error rates. Therefore, achieving this level of accuracy through automation has strong potential to reduce human errors in real-world deployments.

VI. Conclusion

6.1 Summary

In this thesis, we investigated an LLM-based approach for automating VNF installation in Kubernetes environments. The proposed method leverages publicly available LLMs, supported by an AI Agent designed to guide the LLM in producing accurate responses. While in-context learning was employed to enable the generation of valid VNF installation workflows by default, we further integrated RAG and MAD to enhance the LLM's self-healing capability in cases of failure.

To ensure safety and correctness before deployment in a real network, we designed a test module based on a Kubernetes-based network digital twin, which was incorporated into the AI Agent framework. For evaluation, we created MOPs corresponding to widely used VNFs and used them as input prompts to various LLMs. Each LLM was tasked with generating automation workflows in either Python or Ansible.

Using the test module, we measured the success rate of VNF deployments and found that the proposed AI Agent significantly improved the reliability of the generated workflows. Notably, models that initially produced 0% success were able to achieve up to 86.7% success when all supporting techniques were applied. This indicates that, although LLMs alone are currently insufficient for fully automating VNF deployment, they can become effective when combined with additional techniques. Consequently, it becomes possible to eliminate human errors that arise during manual, MOP-based configurations in real network environments, as discussed in the introduction. In our experiments, each individual LLM successfully handled approximately 80% of the MOPs. However, when considering the collective results across all LLMs, at least one LLM succeeded for every MOP. This suggests that, given sufficient resources and time, complete automation of VNF deployment is achievable. Further-

more, we conducted module-wise validation to demonstrate the effectiveness of each component, thereby verifying the overall utility of the proposed AI Agent framework. Also, all developed code is publicly available on GitHub [50].

6.2 Future Work

As natural language-based IBN remains in its early stages, the proposed study opens up several promising avenues for future research. We outline key directions below:

6.2.1 Research Scope Expansion

While our current work has focused on automating the installation of VNFs in a containerized environment, future efforts should shift toward deploying pre-installed CNFs on containers to enhance practical applicability. It is essential to explore methods for simplifying and automating this process. Projects such as GitOps [34] are actively investigating such approaches, and further research is needed to determine how natural language can be used by end-users to trigger automation, and how LLMs can be effectively integrated into this workflow.

6.2.2 Methodological Enhancements

Although we enhanced LLM performance through external techniques without modifying the base models, many promising methods remain unexplored. In particular, fine-tuning LLMs using datasets of correct and incorrect workflows, or applying reinforcement learning-based fine-tuning that rewards correct behavior, could further improve model accuracy and robustness for task-specific automation. In particular, since our environment allows for the evaluation of whether the test module has succeeded, it is possible to generate rewards based on this outcome. Therefore, further research is needed to explore how this feedback mechanism can be effectively utilized.

6.2.3 AI Agent Functionality Extension

In this thesis, the AI Agent considered VNF installation and operational success as the outcome. However, to achieve deeper automation, future work should address dynamic VNF lifecycle management, including SFC (Service Function Chaining), scaling, and migration. These tasks will require combining our proposed AI Agent framework with existing VNF orchestration techniques. Although such integration falls beyond the scope of this thesis, it represents a critical direction toward the complete automation of VNF/CNF management.

요 약 문

우리는 Intent based networking (IBN)의 첫걸음으로서, 자연어 기반의 Virtualed Network Function (VNF) 설치 자동화를 위한 연구를 진행하였다. 특히, 우리는 최근 자연어 처리 분야에서 자연어 입력 자동 분석 뿐 아니라 reasoning, 코딩에까지 높은 성능을 보이고 있는 Large Lanugage Model (LLM)을 이용하여 VNF 구성에 관한 자연어를 입력으로 받아 이를 처리하는 시스템을 개발하고자 하였으며, LLM을 이용하여 높은 성공률로 VNF 구성 자동 워크플로를 생성할 수 있는 AI Agent를 제안, 개발 및 검증하였다. 개발한 AI Agent는 few-shot learning을 통해 사용자가 설치하고자 하는 VNF 설치 워크플로를 생성하며, 이를 실제 네트워크에 적용하기 전에 쿠버네티스로 구축한 Network Digital Twin (NDT)에서 정상적으로 동작하는 워크플로인지 검증하는 단계를 거친다. 만약 문제가 발생할 경우, NDT에서 로그를 추출하여 LLM에게 전달하며 워크플로의 문제점을 고치도록 하며, 특히, 이 때 문제점을 고치기 위해 외부 지식을 활용할 수 있도록 Retrieval Augmented Generation (RAG), Multi-Agent Devate (MAD) 등의 기법을 적용하였다. 실험 결과 제안하는 AI Agent를 통해 프롬프트 만으로는 VNF 구성 완전자동화를 할 수 없던 LLM들 또한 높은 성능으로 NDT에서 동작하는 코드를 생성할 수 있음을 확인하였으며, 이를 통해 IBN의 첫 단계인 translation 및 activation에 해당하는 성과를 내었다.

References

[1] Open Networking Foundation. Software-defined networking: The new norm for networks. Technical report, Open Networking Foundation, April 2012.

[2] ETSI Industry Specification Group (ISG) NFV. Network functions virtualisation – introductory white paper. Technical report, ETSI, October 2012.

[3] Dimitrios Michael Manias, Ali Chouman, and Abdallah Shami. Towards intent-based network management: Large language models for intent extraction in 5g core networks. In *2024 20th International Conference on the Design of Reliable Communication Networks (DRCN)*, pages 1–6. IEEE, 2024.

[4] BlueCat Networks. Network management megatrends 2024. `https://bluecatnetworks.com/wp-content/uploads/2024/08/network-management-megatrends-2024.pdf`, 2024. Accessed: May 30, 2025.

[5] Xu Liu, Peng Zhang, Anubhavnidhi Abhashkumar, Jiawei Chen, and Weirong Jiang. Automatic configuration repair. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, HotNets '24, page 213–220, New York, NY, USA, 2024. Association for Computing Machinery.

[6] Alexander Clemm, Laurent Ciavaglia, Lisandro Zambenedetti Granville, and Jeff Tantsura. Intent-Based Networking - Concepts and Definitions. RFC 9315, October 2022.

[7] Denis Donadel, Francesco Marchiori, Luca Pajola, and Mauro Conti. Can llms understand computer networks? towards a virtual system administrator. In *2024*

IEEE 49th Conference on Local Computer Networks (LCN), pages 1–10. IEEE, 2024.

[8] Jieyu Lin, Kristina Dzeparoska, Ali Tizghadam, and Alberto Leon-Garcia. Appleseed: Intent-based multi-domain infrastructure management via few-shot learning. In *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, pages 539–544. IEEE, 2023.

[9] Eui-Dong Jeong, Hee-Gon Kim, Sukhyun Nam, Jae-Hyoung Yoo, and James Won-Ki Hong. S-witch: Switch configuration assistant with llm and prompt engineering. In *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, pages 1–7. IEEE, 2024.

[10] Kristina Dzeparoska, Jieyu Lin, Ali Tizghadam, and Alberto Leon-Garcia. Llm-based policy generation for intent-based management of applications. In *2023 19th International Conference on Network and Service Management (CNSM)*, pages 1–7. IEEE, 2023.

[11] Linux Foundation and Google. Nephio: Automating cloud-native network functions with kubernetes. `https://nephio.org/`, 2022. White Paper.

[12] Abdelkader Mekrache, Adlen Ksentini, and Christos Verikoukis. Intent-based management of next-generation networks: an llm-centric approach. *IEEE Network*, 38(5):29–36, 2024.

[13] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Dejan Kostic, and Marco Chiesa. Making network configuration human friendly. *arXiv preprint arXiv:2309.06342*, 2023.

[14] Nguyen Van Tu, Jae-Hyoung Yoo, and James Won-Ki Hong. Towards intent-based configuration for network function virtualization using in-context learning in large language models. In *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, pages 1–8. IEEE, 2024.

[15] OpenStack Foundation. OpenStack: Open Source Software for Creating Private and Public Cloud. https://www.openstack.org. [Online; accessed 2025-06-08].

[16] The Kubernetes Authors. Kubernetes. https://github.com/kubernetes/kubernetes, 2014. Accessed: 2025-06-08.

[17] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020.

[18] Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. *arXiv preprint arXiv:2305.14325*, 2023.

[19] OpenAI. OpenAI GPT models. https://platform.openai.com/docs/guides/gpt. [Online; accessed 2025-06-08].

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[21] Meta. Llama 3: Open and efficient foundation models. https://ai.meta.com/research/llama3, 2024. Online; accessed 2025-06-08.

[22] Google. Gemma: Open models based on gemini research and technology, 2024.

[23] DeepSeek. Deepseek llm and deepseek coder. https://github.com/deepseek-ai, 2024. Accessed: 2025-06-08.

[24] Alibaba Group. Qwen-2: A scalable foundation model from alibaba. `https://modelscope.cn/models/qwen/Qwen-2-7B`, 2024. Online; accessed 2025-06-08.

[25] Mojan Javaheripi, Sébastien Bubeck, Marah Abdin, Jyoti Aneja, Caio César T. Mendes, Weizhu Chen, Allie Del Giorno, Ronen Eldan, Sivakanth Gopi, Suriya Gunasekar, Yin Tat Lee, Yuanzhi Li, Anh Nguyen, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Adam Taumann Kalai, and many others. Phi-2: The surprising power of small language models. *Microsoft Research Blog*, 2023. Accessed: 2025-06-08.

[26] Mistral AI. mistralai: Python client library for mistral ai platform. `https://github.com/mistralai/client-python`, 2025. Installable via `pip install mistralai`.

[27] Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, others, and the CodeGemma Team. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*, 2024.

[28] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, et al. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[29] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, et al. Language models are few-shot learners. *Advances in neural information processing systems (NeurIPS)*, 33:1877–1901, 2020.

[30] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use

long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024.

[31] Abhishek Mukherjee, Nazneen Rajani, and Thomas Wolf. Peft: State-of-the-art parameter-efficient fine-tuning methods. *arXiv preprint arXiv:2303.10135*, 2023.

[32] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *Proceedings of the Tenth International Conference on Learning Representations (ICLR)*, 2022.

[33] ETSI. Network Functions Virtualisation (NFV); Management and Orchestration. Technical Report GS NFV-MAN 001 V1.1.1, European Telecommunications Standards Institute (ETSI), 2014. Accessed: 2025-07-03.

[34] Open GitOps Community. Open GitOps specification and guidance. `https://github.com/open-gitops/documents`, 2025. Accessed: 2025-06-20.

[35] Red Hat, Inc. Ansible Documentation, 2025. Accessed: 2025-06-20.

[36] Adnan Yazici et al. Machine learning applications for intent-based networking. *IEEE Communications Magazine*, 58(10):72–77, 2020.

[37] Davide Giustozzi, Fabio Paganelli, Giuseppe Tanganelli, and Carlo Vallati. Toward a network digital twin. *IEEE Communications Magazine*, 60(10):72–77, 2022.

[38] Stack Exchange, Inc. Stack Overflow. `https://stackoverflow.com`. Accessed: 2025-06-08.

[39] Stack exchange api. `https://api.stackexchange.com/`. Accessed: 2025-06-08.

[40] Sukhyun Nam, Jae-Hyoung Yoo, and James Won-Ki Hong. Log-tf-idf for anomaly detection in network switches. In *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, pages 1–9. IEEE, 2024.

[41] Sukhyun Nam, Eui-Dong Jeong, and James Won-Ki Hong. Log-tf-idf and netconf-based network switch anomaly detection. *International Journal of Network Management*, 35(1):e2322, 2025.

[42] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 28(1):11–21, 1972.

[43] Sukhyun Nam, Euidong Jeong, Jibum Hong, Jae-Hyoung Yoo, and James Won-Ki Hong. Log analysis and prediction for anomaly detection in network switches. In *2023 19th International Conference on Network and Service Management (CNSM)*. IEEE, 2023.

[44] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet physics doklady*, 10(8):707–710, 1966.

[45] Princeton University. WordNet. `https://wordnet.princeton.edu/`, 2005. [Online; accessed 2025-06-08].

[46] HAProxy Technologies. Haproxy: The reliable, high-performance tcp/http load balancer, n.d. Online; accessed 2025-06-08.

[47] ndpi. ndpi: Open source deep packet inspection toolkit. `https://www.ntop.org/products/deep-packet-inspection/ndpi/`. Online; accessed 2025-06-08.

[48] OISF (Open Information Security Foundation). Suricata: Open source threat detection engine. `https://suricata.io/`. Online; accessed 2025-06-08.

[49] ntop. ntopng: High-speed web-based traffic analysis and flow collection, n.d. Online; accessed 2025-06-08.

[50] Sukhyun Nam. Llmbasedvnf github. https://github.com/obiwan96/LLMbasedVNF, 2024. Online; accessed 2025-06-08.

[51] Inc. Red Hat. ansible-runner: A tool and python library that helps interface with ansible directly or embed it into other systems. https://github.com/ansible/ansible-runner, 2024. Accessed: 2025-06-08.

[52] Nils Reimers and Iryna Gurevych. sentence-transformers/all-minilm-l6-v2. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2, 2021. Accessed: 2025-06-08.

[53] Chroma Team. Chroma: Ai-native open-source vector database. https://github.com/chroma-core/chroma, 2025. Latest version v1.0.12 (May 30, 2025), Apache2.0 License, accessed: 2025-06-08.

[54] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45. Association for Computational Linguistics, 2020.

[55] Ollama. Ollama: Run and deploy large language models locally, 2023. Accessed: 2025-06-08.

[56] Sukhyun Nam, Nguyen Van Tu, and James Won-Ki Hong. Llm-based vnf deployment automation in openstack environment. In *NOMS 2025-2025 IEEE Network Operations and Management Symposium (accepted to appear)*. IEEE, 2025.

Acknowledgements

우선 6년동안 아낌없는 지도 해주신 홍원기 교수님, 유재형 교수님께 감사드립니다. 두 분께서 학위과정 동안 많은 지도와 가르침을 주셨기에 제가 많이 성장한 것을 느낍니다. 학부 졸업식날 박사 졸업생들을 보며 저 분들은 어떤 기분일까 궁금했었는데, 제게도 이런 글을 쓰는 날이 오는 것을 보니 무엇이든 결국 담담히 이겨내는 것이 중요한 것 같습니다. 매일같이 지켜보며 서로 응원하고 이끌어줬던 선, 후배, 동기들 모두 고맙습니다. 연구실 구성원들이 정말 좋은 사람이어서 6년간 좋은 기억을 많이 가지게 되었습니다. 항상 아껴주고 사랑해주며 응원해준 가족들, 친구들에게 고맙습니다. 덕분에 큰 스트레스 없이 학위과정을 밟을 수 있었습니다. 먼 포항까지 와주고, 좋은 기억 만들어주고, 적응할 수 있도록 도와줘서 고맙습니다. 포항에서 새로 알게된 고마운 사람도 많습니다. 힘들 때 서로 이야기하며 술 한잔에 털어버린 많은 기억들이 소중합니다.

저를 믿고 곁에서 묵묵히 응원해주고 제게 사랑을 준 고마운 인연들에게 감사합니다. *무엇보다도 서로 깊이 사랑하십시오. 사랑은 허다한 죄를 덮습니다.* - 베드로전서. 앞으로의 시련들 또한 사랑의 힘으로 이겨낼 수 있는 사람이 되고 싶습니다.

Curriculum Vitae

Personal Information

Name	:	Sukhyun Nam
Position	:	Ph.D.
Laboratory	:	Distributed Processing & Network Management (DPNM) Lab.
Department	:	Computer Science and Engineering
E-mail	:	obiwan96@postech.ac.kr

Research Areas of Interest

Software-Defined Networking (SDN); Network Function Virtualization (NFV); Network Management; Machine Learning (ML); Large Language Model (LLM); Intent-Based Networking (IBN)

Education

2014 – 2019	School of Computing, Korea Advanced Institute of Science and Technology (B.S.)
2019 – 2021	Department of Computer Science and Engineering, Pohang University of Science and Technology (M.S.)
2021 – 2025	Department of Computer Science and Engineering, Pohang University of Science and Technology (Ph.D.)

Research/Project Experience

2024. 6. – 2025. 12. LLM 기반 네트워크 구성 관리 자동화 기술 개발 (Funded by Samsung)

2024. 4. – 2025. 12. 6G 네트워크 통합 지능평면 기술 개발 (Funded by IITP)

2022. 6. – 2024. 4. 로그 분석 기반 네트워크 비정상 상태 탐지 기술 개발 (Funded by Samsung)

2020. 4. – 2022. 4. 인공지능 기반 트래픽 엔지니어링 기술 개발 (Funded by Samsung)

2019. 7. – 2020. 12. 멀티 서비스를 지원하는 프로그래머블 스위치 제어 기술 개발 (Funded by IITP)

2020. 4. – 2024. 12. 초저지연 네트워크 서비스를 위한 SDN 기반 인공지능 관제 시스템 개발 (Funded by KIAT)

2021. 1. – 2023. 12. 인공지능 기반 가상 네트워크 관리 기술 개발 (Funded by IITP)

Publications: International Journal

1. **Sukhuyn Nam**, Eui-Dong Jeong, James Won-Ki Hong, "Log-TF-IDF and NETCONF based Switch Anomaly Detection", International Journal of Network Management (IJNM) (SCIE) Jan 2025.
2. Nguyen Van Tu, **Sukhuyn Nam**, James Won-Ki Hong, "Intent-Based Network Configuration Using Large Language Models", International Journal of Network Management (IJNM) (SCIE), Nov 2024.

3. Daniela N. Rim, DongNyeong Heo, Chungjun Lee, **Sukhuyn Nam**, Jae-Hyoung Yoo, James Won-Ki Hong, Heeyoul Choi, "Anomaly detection based on system text logs of virtual network functions", Big Data Research (SCIE), Volume 38, 2024.

Publications: International Conference

1. **Sukhuyn Nam**, Nguyen Van Tu, James Won-Ki Hong, "LLM-based VNF Deployment Automation in OpenStack Environment", 2025 IEEE/IFIP Network Operations and Management Symposium (NOMS 2025), Honolulu, USA, 12-16 May, 2025.
2. **Sukhuyn Nam**, Jae-Hyoung Yoo, James Won-Ki Hong, "Log-TF-IDF for Anomaly Detection in Network Switches", 2024 IEEE/IFIP Network Operations and Management Symposium (NOMS 2024), Seoul, South Korea, 6-10 May, 2024.
3. Eui-Dong Jeong, Hee-Gon Kim, **Sukhuyn Nam**, Jae-Hyoung Yoo, James Won-Ki Hong, "S-Witch: Switch Configuration Assistant with LLM and Prompt Engineering", 1st IEEE/IFIP Workshop on Generative AI for Network Management 2024 (GAIN 2024), Seoul, South Korea, 10 May, 2024.
4. **Sukhuyn Nam**, Euidong Jeong, Jibum Hong, Jae-Hyoung Yoo, and James Won-Ki Hong, "Log Analysis and Prediction for Anomaly Detection in Network Switches", 19th International Conference on Network and Service Management (CNSM 2023), Niagara Falls, Canada, Oct. 30 – Nov. 2, 2023.
5. **Sukhuyn Nam**, Jae-Hyoung Yoo, and James Won-Ki Hong, "VM Failure Prediction with Log Analysis using BERT-CNN Model", 18th International Conference on Network and Service Management (CNSM 2022), Thessaloniki, Greece, Oct. 31 - Nov. 4, 2022.
6. **Sukhuyn Nam**, Jibum Hong, Jae-Hyoung Yoo, James Won-Ki Hong, "Virtual Machine Failure Prediction using Log Analysis", The 22nd Asia-Pacific Net-

work Operations and Management Symposium (APNOMS 2021), Tainan, Taiwan, Sep. 8-10, 2021.

7. Jiyoon Lim, **Sukhuyn Nam**, Jae-Hyoung Yoo, James Won-Ki Hong, "Best next hop Load Balancing Algorithm with Inband network telemetry", 16th International Conference on Network and Service Management (CNSM 2020), Virtual Conference, Nov. 2-6, 2020.

8. **Sukhuyn Nam**, Jiyoon Lim, Jae-Hyoung Yoo, James Won-Ki Hong, "Network Anomaly Detection Based on In-band Network Telemetry with RNN", The Fifth International Conference On Consumer Electronics (ICCE-Asia 2020), Seoul, Korea, Nov. 1-3, 2020.

9. Jiyoon Lim, **Sukhuyn Nam**, Jae-Hyoung Yoo, James Won-Ki Hong, "Load Balancing Algorithm with Programmable Switch", The 21st Asia-Pacific Network Operations and Management Symposium (APNOMS 2020), Daegu, Korea, Sep. 23-25, 2020

Publications: Domestic Journals

1. 임지윤, **남석현**, 유재형, 홍원기, "INT 기반 네트워크 이상 상태 탐지 기술 연구", KNOM Review, Vol. 22, No. 3, December 2019.

Publications: Domestic Conference

1. **남석현**, 정의동, 홍지범, 유재형, 홍원기, "패턴 분석 기반 스위치 로그 예측 기법 연구", KNOM Conference 2023, Jeju, Korea, May 18-19, 2023.

2. 홍지범, 허동녕, **남석현**, 유재형, 홍원기, "가상 네트워크 관리를 위한 기계학습 기반 네트워크 공격 및 침입 탐지 시스템 설계", KNOM Conference 2022, Chuncheon, Korea, May 12-13, 2022.

3. **남석현**, 유재형, 홍원기, “BERT 기반 VM 고장 예측 기법 연구”, KNOM Conference 2022, Chuncheon, Korea, May 12-13, 2022.

4. 홍지범, 정세연, **남석현**, 유재형, 홍원기, “머신러닝을 이용한 VNF 이상 탐지 및 고장 예측 기반 NFV 관리 시스템 설계”, 2022 한국통신학회 동계종합학술발표회, Pyeongchang, Korea, Feb. 9-11, 2022.

5. **남석현**, 홍지범, 유재형, 홍원기, “로그 및 자원 분석을 통한 VNF 고장 예측에 관한 연구”, KNOM Conference 2021, On-line KNOM Conference Venue, Korea, April 30, 2021, pp. 17-20.

6. 홍지범, **남석현**, 유재형, 홍원기, “가상 네트워크 관리를 위한 기계학습 기반 이상 탐지 시스템 설계”, KNOM Conference 2021, On-line KNOM Conference Venue, Korea, April 30, 2021, pp. 120-122.

7. 임지윤, **남석현**, 유재형, 홍원기, “강화학습 기반 링크가중치 조정 로드밸런싱 알고리즘 연구”, KNOM Conference 2020, On-line KNOM Conference Venue, Korea, May 15, 2020, pp. 28-31.

8. **남석현**, 임지윤, 유재형, 홍원기, “네트워크 텔레메트리 기반 통합 네트워크 관리 시스템 연구”, KNOM Conference 2020, On-line KNOM Conference Venue, Korea, May 15, 2020, pp. 130-132.

9. **남석현**, 현종환, 유재형, 홍원기, “네트워크 텔레메트리를 활용한 머신 러닝 기반 네트워크 이상 탐지 기법 연구”, KNOM Conference 2019, Daegu, Korea, May. 30, 2019, pp. 75-77.

International Patents

1. Seungho Ryu, **Sukhuyn Nam**, Taewoo Kim, Jae-Hyoung Yoo, Jaegon Kim, James Won-Ki Hong, “Electronic Device and Method for Identifying Anomaly”, Patent No.: PCT/KR2024/013947, Sep. 12, 2024 (Applicant: POSTECH, Samsung Electronics Co., Ltd).

2. James Won-Ki Hong, Jae-Hyoung Yoo, **Sukhuyn Nam**, “Method and Apparatus of Virtual Machine Failure Prediction using Machine Learning”, Patent No.: PCT/KR2023/008760, Jun. 23, 2023 (Applicant: POSTECH).

Domestic Patents

1. Jaegon Kim, **Sukhuyn Nam**, Seungho Ryu, James Won-Ki Hong, Taewoo Kim, “Method and Apparatus to Detect Status of Network Entity by Analyzing Log Data”, Patent No.: 10-2025-0073887, Jun. 05, 2025 (Applicant: POSTECH, Samsung Electronics Co., Ltd).

2. James Won-Ki Hong, Eui-Dong Jeong, **Sukhuyn Nam**, “Method and Device for Setting Network Automatically”, Patent No.: 10-2024-0183556, Dec. 11, 2024 (Applicant: POSTECH).

3. James Won-Ki Hong, Hee-Gon Kim, **Sukhuyn Nam**, “System and Method for Virtual Network Function Management using Artificial Intelligence and Network Digital Twin”, Patent No.: 10-2024-0183716, Dec. 11, 2024 (Applicant: POSTECH).

4. Seungho Ryu, **Sukhuyn Nam**, Taewoo Kim, Jae-Hyoung Yoo, Jaegon Kim, James Won-Ki Hong, “Electronic Device and Method for Identifying Anomaly”, Patent No.: 10-2023-0152228, Nov. 06, 2023 (Applicant: POSTECH, Samsung Electronics Co., Ltd).

5. James Won-Ki Hong, Jae-Hyoung Yoo, **Sukhuyn Nam**, “Virtual Machine Failure Prediction Automation Techniques based on Logs and BERT-CNN”, Patent No.: 10-2023-0059859, May. 09, 2023 (Applicant: POSTECH).

6. James Won-Ki Hong, Jae-Hyoung Yoo, Jibum Hong, **Sukhuyn Nam**, “Method for Automating Failure Prediction of Virtual Machines and Servers through

Log Message Analysis, Apparatus and System thereof ", Patent No.: 10-2021-0143954, Oct. 26, 2021 (Applicant: POSTECH).

7. James Won-Ki Hong, Jae-Hyoung Yoo, Jiyoon Lim, **Sukhuyn Nam**, "Network Anomaly Detection Method and Device", Patent No.: 10-2019-0147915, Nov. 18, 2019 (Applicant: POSTECH).

8. James Won-Ki Hong, Jae-Hyoung Yoo, Jiyoon Lim, **Sukhuyn Nam**, "Network Control Method and Device", Patent No.: 10-2019-0147898, Nov. 18, 2019 (Applicant: POSTECH).

Awards

Title	Organizations	Date
Student Traver Grant	NOMS 2025	2025.05

Personal Experience

Title	Organizations	Date
Web Master & Cameraman	NOMS 2024	2024.05

Teaching Assistance

Title	Course	Date
Dept. of Humanities & Social Science, KAIST	HSS062: Humanity/Leadership III, EDM and DJing	2015 Spring-2016 Spring
Dept. of CSE, POSTECH	CSED232: Object Oriented Programming	2020 Spring, 2022 Spring

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