KUBETEUS: An Intelligent Network Policy Generation Framework for Containers

Bom Kim, Hyeonjun Park, Seungsoo Lee
Incheon National University, Republic of Korea
{zxx0313, isc10093, seungsoo}@inu.ac.kr

***Abstract*—Containers have become the standard for delivering cloud-native services by taking advantage of their scalability, portability, and resource efficiency. However, particularly in network policies, they have also become major targets for various security attacks that exploit misconfigurations and vulnerabilities. Especially in complex cloud-native environments, manually managing network policies is prone to errors, and existing studies that automate policy generation often have limitations in accuracy. In this paper, we present KUBETEUS, a highly automated, intelligent network policy generation framework. Our system operates in an intent-driven manner, enhanced by natural language processing (NLP) and fine-tuned Large Language Models (LLMs), enabling the generation of network policies without needing to understand complex configurations. Furthermore, our system devises a multi-stage validation process to fundamentally prevent misconfigurations in network policy enforcement. The evaluation of KUBETEUS demonstrates its effectiveness, with the most improved fine-tuned LLM achieving a 360% increase in BLEU score and a 233% increase in ROUGE-2 score compared to the baseline model. We believe that the approach presented in this paper is applicable to the wide range of container-native policy platforms in used today, and that its broader adoption will help address more complex security policy generation concerns.**

***Index Terms*—Cloud-native Architecture, Container Network, Intent-based Network Policy, LLM-based Policy Generation**

I. INTRODUCTION

With the rapid advancement of cloud computing technology, the cloud market is experiencing unprecedented growth. According to reports released by Gartner and Flexera in 2023 and 2024, consumer spending on the public cloud is expected to reach $678.8 billion, with approximately 89% of enterprises adopting multi-cloud strategies and 51% having already fully migrated to cloud infrastructure [1]–[3]. This acceleration in cloud adoption is accompanied by an explosive increase in the use of cloud-native technologies. Cloud-native applications, which are based on microservice architectures, contrast with traditional monolithic architectures, leveraging the flexibility and efficiency of container technology. Notably, the emergence of container orchestration platforms, such as Kubernetes [4], has further facilitated the efficient management of these complex container-based microservices.

However, this evolution is accompanied by new security challenges, particularly cyberattacks targeting incorrect settings or security vulnerabilities. For example, the 2023 Toyota data breach revealed that human error led to cloud configuration errors that exposed the data of about 2.15 million customers over a period of 10 years [5]. According to Red Hat's 2024 State of Cloud Security Report [6], 42% of respondents identified security as the biggest concern in their container and Kubernetes, with misconfiguration being a major security risk factor. Managing these configurations, including network security policies, in a dynamically changing cloud-native environment is a significant challenge. In a microservice architecture environment, network configurations change frequently, and interactions between services are complex, making it impractical to manually configure and manage numerous network policies individually. Furthermore, manual management fails to ensure policy consistency and real-time responsiveness, and minor mistakes, such as typos or missing settings, can compromise the security of the entire system.

To address these challenges, various studies [7]–[12] have been proposed to automate network policy management in cloud-native environments. These studies primarily focus on methods for automatically generating network policies through log-based analysis and static analysis. However, these approaches have several significant limitations. First, log-based analysis methods generate policies based on past network activities, which may not be adequate for new types of traffic or unexpected situations. Second, static analysis-based approaches can help to understand the structure and intended behavior of applications but struggle to reflect dynamic changes, particularly due to the complexity and frequent updates of microservice architectures. Recently, intent-based access control (IBAC) approaches [13]–[18] which express users' intents in natural language and convert them into network policies, have been proposed. However, these methods are not well-suited for cloud-native environments characterized by frequent changes and require significant user intervention.

In this work, we propose an intelligent framework, KUBETEUS, for the automatic generation of network policies tailored specifically for cloud-native environments. KUBETEUS achieves intelligence by going beyond simple automation: it comprehends security requirements through Natural Language Processing (NLP) and Large Language Models (LLMs), while continuously adapting to and validating against real-time cluster states. Since fine-tuning is essential for the LLM to specialize in network policy generation, we provide an automatic guide function for LLM fine-tuning and hyperparameter optimization. Additionally, we devise an enhanced mechanism for the intent prompt, which is fed into the fine-tuned LLMs for the generation of network policies. Our system

* Corresponding author: Prof. Seungsoo Lee (seungsoo@inu.ac.kr)

is not tightly coupled with any specific container network interface (CNI) and supports the network policy grammar and functions of various CNIs. Finally, it includes multi-step pre-validation processes that detect and block incorrect policy configurations before enforcing the network policy.

In the evaluation, KUBETEUS demonstrated high performance with an F1 score of approximately 97% for entity classification in intent prompts. Moreover, the fine-tuned LLM showed significant improvements in generating network security policies across various container network interface (CNI) environments, with BLEU scores increasing up to 360% and ROUGE-2 scores up to 233% compared to the baseline model. These results clearly indicate the potential of KUBETEUS as a reliable tool for generating network policies in real-world cloud-native environments.

This paper makes the following contributions:

- We present the design and implementation of a new network policy generation framework, KUBETEUS, capable of intelligently generating network policies in cloud-native environments.
- We present an entity classification model for prompt enhancement and an automation for fine-tuning LLMs, thereby allowing users to generate policies without understanding the detailed policy architectures.
- We present a multi-step validation process for network policies that prevents incorrect policies from being enforced in the cluster.
- We evaluate KUBETEUS by analyzing various metrics against the proposed classification model and fine-tuned LLMs. Our evaluation demonstrates the effectiveness of KUBETEUS in generating network policies and detecting misconfigurations.

II. BACKGROUND AND PROBLEM STATEMENTS

A. Background

Containerization. The containers are a lightweight virtualization technology designed to overcome the limitations of virtual machines (VMs), by making them quick to start, resource-efficient, and portable. The containers offer consistent execution across different environments by packaging applications and dependencies into a single unit. By leveraging these benefits, container-based microservice architecture (MSA) [19] has become a core paradigm in modern cloud-native application development. MSA divides applications into independently deployable services, each handling specific business functions with enhanced scalability and maintainability. However, managing numerous microservices introduces operational complexity, which necessitates orchestration. Kubernetes, the de facto standard for container orchestration, uses a group of containers, known as pods, as the smallest resource unit and manages tasks such as scheduling, load balancing, service discovery, and rolling updates for these pods.

Intent-based Access Control using LLMs. The intent-based access control (IBAC) [20] is a promising mechanism that determines access rights by comprehensively analyzing

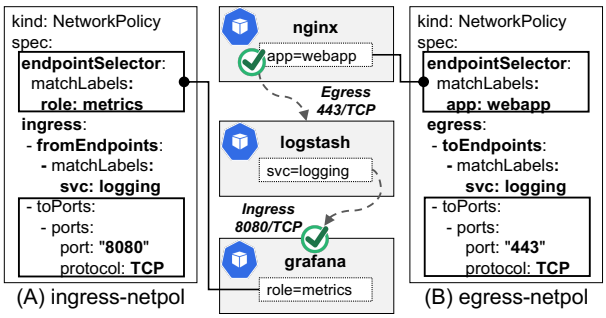

Fig. 1: The examples of two different types of network policy enforcement scenarios.

user behavior, the context of the request, and the impact on system resources. Notably, the adoption of natural language processing (NLP) and large language models (LLMs) in the implementation of IBAC holds significant potential. While LLMs can accurately interpret user intentions through advanced natural language processing derived from extensive training, their large number of parameters leads to high computational costs and processing delays [21]. To address these limitations, smaller-scale LLMs (sLLMs) have been proposed as a more resource-efficient alternative. Unlike traditional small language models (sLMs), which rely on simpler architectures optimized for specific tasks, sLLMs maintain the architectures and extensive pretraining benefits of full-scale LLMs while reducing parameter counts [22]. The performance of sLLMs can be further enhanced through fine-tuning, a process that involves additional training of a pre-trained model to specialize it for specific tasks or domains.

Network Policy in Cloud-native Environments. The network policies define communication rules between resources within the Kubernetes cluster, enhancing security through network isolation. Each container network interface (CNI) [23] implements these policies with unique structures and functions, supporting different levels of network access control [24]. Network policies use selectors to identify target resources, and rules to define traffic permissions through ingress and egress specifications, determining whether to allow or block network traffic. Figure 1 shows two enforcement scenarios. In (A), an ingress policy controls inbound traffic to pods labeled 'role: metrics' by allowing TCP traffic on port 8080 from endpoints labeled 'svc: logging', enabling secure metrics collection in monitoring systems where log aggregators need to forward metrics to visualization dashboards. In (B), an egress policy governs outbound traffic from pods labeled 'app: webapp' to port 443 over TCP, allowing secure log submission from web applications to logging services. Beyond basic IP and label-based traffic control, these policies support advanced features such as HTTP path control and DNS filtering, enabling comprehensive network security management in cloud-native environments.

B. Problem Statements

Generating network policies in cloud-native environments encounters multiple challenges, stemming from the intricate

and dynamic characteristics of microservice architectures. This section explores three critical challenges in ensuring effective network policy management.

C1: Insufficient Understanding of the Dynamic Nature of Container-Based Microservices. The current systems for generating network policies have limitations, particularly in reflecting the context of dynamic microservice environments. A representative method is the log-based approach [8], [11], which relies on historical network logs to generate policies. This approach struggles to continuously reflect the real-time relationships and communication patterns among frequently changing services. Similarly, a policy generation methodology based on static analysis of container source code [12] fails to account for the complexity and dynamic nature of microservice architectures. Consequently, these systems generally rely on predefined templates or rule-based approaches, lacking a comprehensive understanding of the broader container-based microservice ecosystem.

C2: Limitations in Supporting Heterogeneous Container Network Interfaces. Generating and managing consistent network policies in a container-based microservice environment with diverse Container Network Interfaces (CNIs) is a significant challenge. Each CNI has its own unique network policy structure and functions, requiring network administrators to understand and configure policies for each individually [24], [25]. This significantly increases the complexity of overall policy management and makes it difficult to maintain a consistent network security posture. Current automated policy generation systems are often closely integrated with a specific CNI. Consequently, significant administrator intervention is required to adapt these policies for compatibility with other CNIs. In complex applications comprising tens or hundreds of microservices, such continuous manual adjustments by the administrators hinder the swift and accurate application of policies and even increase the likelihood of human error.

C3: Risks of Misconfiguration in Network Policy Generation. In a microservice architecture comprising numerous containers, complex dependencies and frequent updates between services can lead to misconfigurations [26]. However, most automated network policy generation systems primarily focus on creating policies without verifying their correctness. Furthermore, the validity of existing network policies can be compromised instantly due to manual interventions by network administrators. Incorrectly configured network policies may either block necessary communication between services or permit unauthorized access, potentially causing serious security vulnerabilities or service disruptions. Unfortunately, due to the dynamic nature of microservices, detecting and correcting policy misconfigurations promptly is highly challenging.

III. KubeTeus Design

This section provides an overview of the design considerations that motivated the development of KubeTeus and describes its system architecture. Succinctly, our system leverages natural language processing (NLP) techniques to intelligently and automatically generate network policies tailored for cloud-native environments, which are characterized by their dynamic and complex nature.

A. Design Considerations

1) Advanced Intelligent Policy Generation. First, it should accurately recognize user intentions and simultaneously understand the cluster states in real time to intelligently generate network policies. Additionally, it should provide a highly automated interface that analyzes service relationships within the cluster using only the configuration files, minimizing user intervention. Thus, we adopt natural language processing technologies, such as large language models (LLMs) and prompt engineering, to interpret the security requirements of the users, and convert them into specific network policies.

2) Support for Diverse Container Network Interfaces. Second, it should support the various Container Network Interfaces (CNIs), including the default Kubernetes network policy, without being dependent on any specific CNI. To achieve this, the system should automatically recognize the unique characteristics and functions of each CNI and generate network policies optimized for the cluster environment. Additionally, we provide a methodology for maintaining consistency by automatically detecting CNI changes and distributing corresponding network policies, taking into account the dynamic nature of container and microservice environments. This approach enables users to create and manage network policies consistently without the need to manually configure complex settings for each CNI employed.

3) Multi-step Policy Validation. Third, it should include a mechanism to verify whether the generated network policy is suitable for the actual cluster environment and does not lead to critical misconfigurations. For this, the system employs the following multi-step validation process. First at all, it checks the syntactic correctness of the network policy itself, verifies the existence and status of the resources (e.g., containers) in the cluster to which the policy applies, and ensures that the policy details match the resources affected. And then, if a problem is identified during this validation process, the system provides a detailed error report and a correction plan to facilitate problem resolution. This ensures that only correctly generated network policies are enforced, thereby preventing the conflicts that could arise from incorrect policy configurations.

B. System Architecture and Workflow

This section presents the overall architecture and workflow of KubeTeus. As illustrated in Figure 2, our system comprises five key components: fine-tuning automator, prompt processor, policy processor, policy validator, and policy enforcer. The system operates in two phases to meet the above design considerations as follows.

First, the automatic fine-tuning phase is designed to automate the complex fine-tuning process for the LLMs and ensure compatibility with upcoming newer LLMs, with the *fine-tuning automator* playing a central role. The fine-tuning automator analyzes the configuration file containing fine-tuning parameters provided by the network administrator. If

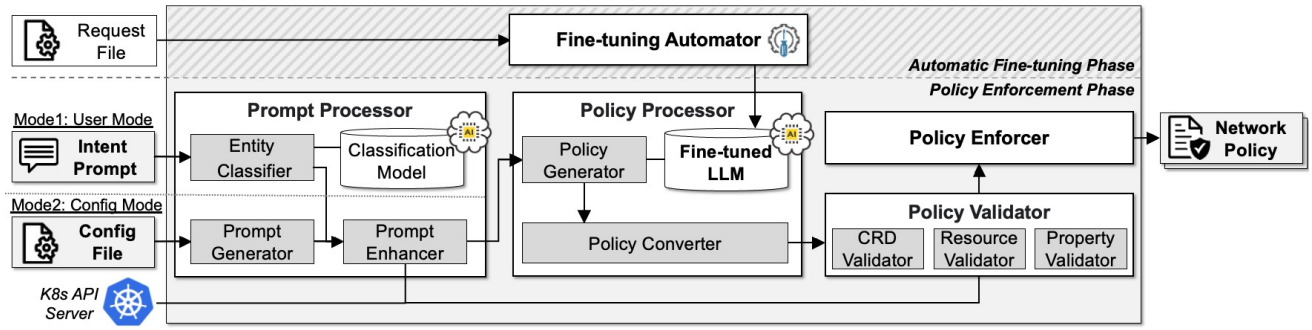

Fig. 2: Overall architecture and its workflow of KUBETEUS with five key components: (*i*) fine-tuning automator, (*ii*) prompt processor, (*iii*) policy processor, (*iv*) policy validator, and (*v*) policy enforcer. Additionally, our system includes two operational phases: automatic fine-tuning and policy enforcement.

specific parameters are not specified by the administrator, it automatically extracts optimized hyperparameters using Optuna [27], which efficiently identifies the optimal parameter combination through the advanced algorithms such as Bayesian optimization. The dataset required for the fine-tuning process can utilize the basic dataset provided by our system or the administrator's own dataset. Based on the analyzed parameters and the dataset, AutoTrain [28] is utilized to automatically fine-tune a specific LLM. The fine-tuned model is then used to generate network policies that reflect the administrator's intent and the real-time state of the cluster.

Second, the policy enforcement phase automates the entire process of network policy generation, with a primary focus on refining the prompt used in the generation process. This phase can be initiated either by direct user input (user mode) or based on resource configuration files (config mode) such as pod in the cluster. In the user mode, the user provides the policy intent in natural language without needing to understand complex policy syntax, and then the *prompt processor*, equipped with an entity classifier based on Bidirectional Encoder Representations from Transformers (BERT [29]), as shown in Figure 3, identifies the necessary entities from the input sentence. Conversely, in config mode, the prompt generator collects resource information, such as pods, deployments, and services, across all namespaces in the cluster. The detailed information, including metadata, specifications, and status of each resource, is extracted during this process. The initial prompt is then automatically generated using a predefined template that corresponds to each resource type. This initial prompt, generated in both modes, is enhanced by the prompt enhancer, which queries the Kubernetes API for real-time cluster information, infers relationships between the resources, and incorporates this data into the prompt, transforming it into a systematic, cluster-specific prompt. The enhanced prompt is then passed to the policy generator within the *policy processor*, which interprets the intent using a pre-fine-tuned LLM and generates the network policies optimized for the cluster's current state. The generated policies are then converted into the formats compatible with each CNI by the policy converter.

The generated policy is validated by the *policy validator* before being enforced in the cluster, following a three-step process. The custom resource definition (CRD [30]) validator serves as the first line of defense, checking the structural integrity of the policy itself by validating its structure against the schema defined in the policy CRD. Then, the resource validator verifies the existence and status of the resources in the cluster to which the policy will be applied. Finally, the property validator ensures the policy is accurately applied to the intended resources by verifying that the policy details align with the affected resources. If a validation failure is detected, an error is logged, and a correction plan is provided based on its cause. The network policies that pass validation are enforced in the cluster via the Kubernetes API by the *policy enforcer*. The enforcer monitors the results of the policy application and ensures accurate enforcement through a retry mechanism when necessary.

IV. KUBETEUS SYSTEM DETAILS

A. Generative AI-Based Model Learning

The key function of KUBETEUS is to accurately identify the user intention and generate an appropriate network policy based on it. For this, our system utilizes the advanced natural language processing techniques that combine the named entity recognition (NER) and the large language models (LLMs).

Dataset Construction and Preprocessing. Since existing general language model datasets do not adequately reflect the specialized structure and terminology of network policies for the cloud-native environments, we developed a custom dataset to train the learning model on various network policy structures and grammars from diverse CNIs. For this, we collected actual network policies from open sources, enabling the model to learn real-world network policy patterns. Additionally, we significantly expanded this initial dataset using data augmentation techniques to enhance the generalization ability of the model and adaptability to various scenarios.

For the specific dataset construction and preprocessing process, we first crawled the network policies from Kubernetes [4], Cilium [31], and Calico [32] CNIs from open-source platforms such as GitHub to construct the initial dataset. As shown in Table I, a total of 857 original network policies were collected and parsed based on their policy structures to separate them into selectors and rules. During this process,

TABLE I: The summary of the datasets for network policies (policy) and intent prompts (intent).

Type	Format	#Origin	#Train	#Test	#Total	Size(MB)
Policy	json	857	132,899	33,225	166,124	187.3
Intent	csv	857	40,048	10,012	50,060	93.2

TABLE II: The summary of LLMs fine-tuned for policy generation.

Name	#Size	#Params
Meta/Meta-Llama-3-8B-Instruct [34]	16GB	8.03B
DeepSeek/deepseek-coder-7b-instruct-v1.5 [35]	14GB	6.91B
MistralAI/Mistral-7B-Instruct-v0.2 [36]	15GB	7.24B
Google/codegemma-7b-it [37]	17GB	8.54B
Meta/codeLlama-7b-Instruct-hf [38]	14GB	6.74B

we removed the duplicate elements to ensure data uniqueness. Next, we applied shuffling and combination techniques to the refined data to increase the dataset size. By independently managing selectors and rules during this process, we ensured the semantic validity of combined policies while avoiding redundancy. This approach resulted in 166,124 network policy samples. These samples used a special token to distinguish input and output pairs (intent prompt, policy output), and we randomly selected 50,060 intent prompts to train the classifier model. Subsequently, we assigned labels corresponding to 13 entity types (Figure 3) to these intent prompt samples using the BIESO tagging system [33] in NER. This labeling task was performed through a combination of manual review and automated scripts.

Domain-Specific NER. To generate accurate network policies, it is crucial to identify policy-related entities in the intent prompts. Therefore, we devised a BERT-based named entity recognition (NER) model, as shown in Figure 3. This model is specialized for the network policy domain and can effectively identify the policy-related entities that general NER models could miss. The model architecture consists of a BERT embedding layer and a custom classifier layer. The BERT embedding layer utilizes a pre-trained BERT-base model (110M parameters), with the weights frozen to prevent overfitting and enhance learning efficiency while leveraging the robust language understanding capabilities of the BERT. The custom classifier layer is designed with a linear-tanh structure, comprising five layers (as shown in Figure 3), which can effectively model the complex relationships between various entities. The model is trained to identify 13 entity classes optimized for the network policies, such as POLICY, LABEL, POD_NAME, and NAMESPACE. These domain-specific entities, not covered in general NER systems, significantly improve the accuracy of prompt calibration and the network policy generation processes.

Automatic LLM Fine-tuning. The intent prompt is refined through the NER process described above and then transformed into a network policy using an LLM. The performance of the LLM can be improved by fine-tuning, and we propose an automated optimization process to enhance this

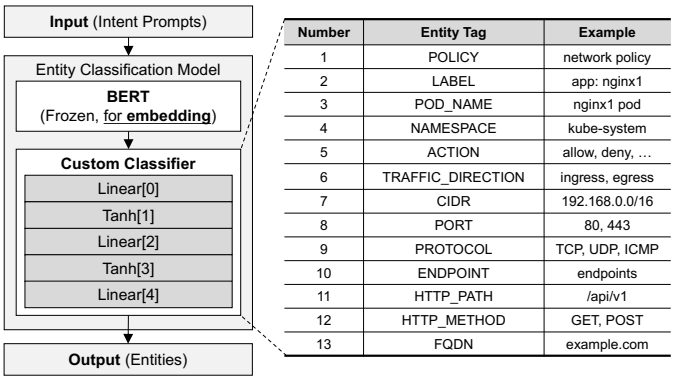

Fig. 3: The design of the entity classification model for network policy-specialized NER.

fine-tuning efficiently. For this, we utilize LoRA (Low-Rank Adaptation) [39]. The LoRA was selected because, rather than updating the entire model's weights, it learns only a small subset of parameters using low-rank decomposition, thereby only partially updating the parameters. Building on this efficient parameter update strategy, we use Optuna [27] for automatic hyperparameter tuning of learning rate, batch size, and LoRA-specific parameters. Our approach introduces rank-8 matrices into the core layers of the model (e.g., attention layers, feedforward layers), enabling effective fine-tuning while limiting the overall increase in model size to approximately 0.1-0.2% [40]. The learning process is further enhanced by the Adam optimizer with gradient clipping (maximum norm 1.0) and weight decay (0.01) for stability and efficiency.

This automated fine-tuning process was applied to five LLMs, as listed in Table II. These models are open text-to-text, decoder-specific generative models with 6-8 billion parameters, chosen to reduce computational cost and processing delays while maintaining the performance of large-scale LLMs. Despite using a smaller number of parameters, these models demonstrate high performance and generate high-quality responses to intent prompts related to the network policies. As a result of the automated fine-tuning, we effectively specialize the model for network policy generation, updating only about 0.5% of the overall model parameters. This approach enables rapid adaptation and expansion when adopting new LLMs.

B. Customized Prompt Engineering

KUBETEUS achieves accurate network policy generation by elaborately enhancing the intent prompt, which is then input into the fine-tuned LLM described above. This process involves accurately understanding the intent prompt and refining it by incorporating the real-time status of the cluster and the specifics of various CNIs.

Figure 4 illustrates how our system refines the initial intent prompt. First, the key entities are extracted from the initial intent prompt using the previously developed domain-specific NER model (A). In this example, the entities critical to the network policy, such as 'nginx1' (POD_NAME), 'incoming' (DIRECTION), and 'endpoint' (ENDPOINT), are accurately

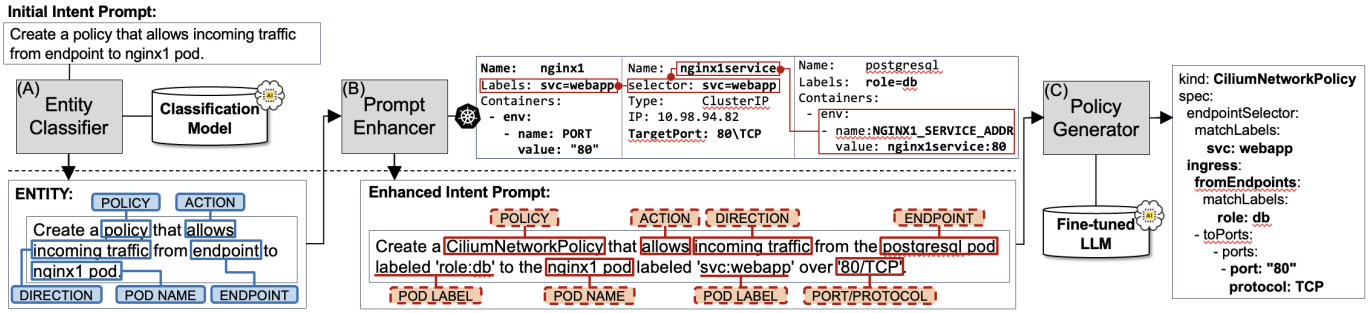

Fig. 4: The example procedure for generating a network policy, from prompt enhancement to the use of fine-tuned LLMs. The entities in blue solid lines are reconfigured to entities in red dotted lines.

identified from the initial prompt. Next, the real-time status of the cluster is considered based on the extracted entities. This is achieved by querying detailed information about the related resources through the Kubernetes API, such as the label of the 'nginx1' pod (svc=webapp) and associated service information. This step also includes inferring the relationships between these resources. By synthesizing the analyzed information, the initial intent prompt is refined into an enhanced intent prompt that is specific and clear (B). This enhanced intent prompt is then input into the fine-tuned LLM to generate the final policy (C). The fine-tuned LLM accurately configures each field of the policy based on the detailed intent prompt. For instance, DIRECTION is used to determine whether the rule is ingress or egress, and LABEL is mapped to the 'endpointSelector', which specifies the resource to which the policy applies.

Various CNIs employed in a cluster have their own network policy grammar and functions, complicating the generation of network policies. To address this issue, our system automatically detects the CNI currently installed in the cluster via the Kubernetes API first. This detection is achieved by querying the cluster configuration or checking for the existence of specific CNI-related custom resources. For instance, if the Calico CNI is installed, the presence of the 'caliconetworkpolicies' CRD can be verified. Next, during the intent prompt parsing step, the system identifies the CNI-specific keywords included in the prompt. For example, the term 'endpoint' is a Cilium-style network policy. However, if the currently installed CNI is Calico, the prompt enhancer automatically converts the term to the Calico equivalent, 'podSelector.' As a result, the fine-tuned LLM generates a network policy that adheres to the Calico grammar and structure based on this enhanced intent prompt. This approach allows users to generate correct network policies without needing to know the syntax of a specific CNI or when using terminology different from the CNI currently installed on the cluster. Furthermore, when the CNI used in the cluster transitions to a different CNI, the system automatically converts the existing network policies to align with the new CNI while preserving their original intent.

C. Multi-Step Policy Validation

KubeTeus implements a three-step validation procedure to ensure the safety of network policy enforcement: custom resource definition (CRD) validation, resource validation, and property validation. This procedure is applied not only to policies generated by our system but also to those written by administrators, ensuring thorough validation of all the network policies. In addition, any issues identified at each step are immediately logged, and administrators are provided with clear error messages and suggested corrections.

The first step, the CRD validation, checks the grammatical correctness and structural integrity of the policy. As shown in Figure 5, it verifies the existence and format of the required fields in the Network Policy, such as 'kind', 'metadata', and 'spec', and ensures that the structure of the 'podSelector' and 'egress' rules complies with the Kubernetes CRD schema. Upon passing the CRD validation, the resource validation then verifies in real-time whether the Kubernetes resources referenced by the policy actually exist via the Kubernetes API. For instance, as shown in Figure 5, it verifies whether a pod with the label 'app: httpd' is present in the cluster. Finally, the property validation thoroughly checks whether the detailed properties of the network policy align with the actual cluster environment. This includes verifying whether the specified port is correctly listening for the corresponding service and ensuring the protocol (TCP/UDP) is set correctly. It also checks the validity of IP ranges specified in the policy and ensures they accurately target the intended network segment. For example, as depicted in Figure 5, the port 4443 specified in 'Misconfiguration.yaml' does not match the actual port from the targeted pod, leading to an error being detected and the policy not being enforced. Subsequently, the correlation between the error log information and the properties of the actual resource is calculated to suggest the correct listening port information.

V. Evaluation

A. Implementation

We have implemented an instance of KubeTeus using a combination of Go and Python to verify its feasibility and effectiveness. Currently supported policies include Kubernetes Network Policy [41], Cilium Network Policy [42], and Calico Network Policy [43], covering both Layer 3-4 and Layer 7. To enhance the initial intent prompt, we developed an entity classification model based on BERT [29] and RoBERTa [44]. In addition, the five LLMs shown in Table II, which were

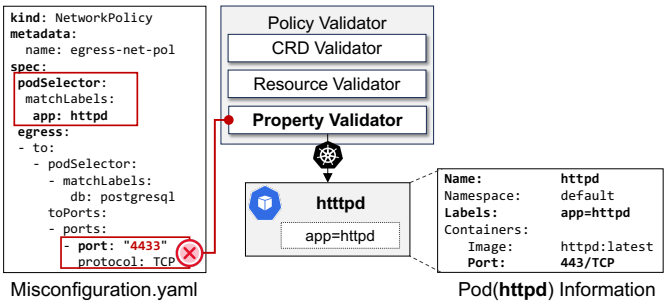

Fig. 5: The example of property validation for detecting the misconfigured policy.

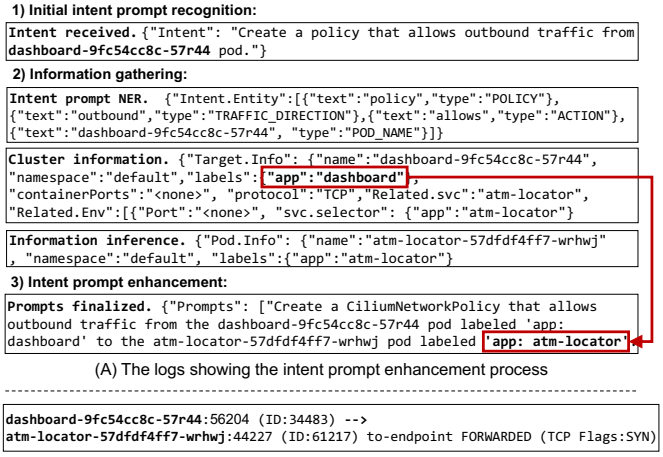

Fig. 6: The results of enhanced prompting (A) and network log from correct policy enforcement (B).

used to generate the network policies, were fine-tuned in advance. In summary, to support the design features described in Section III-A, we implemented a fine-tuning automator, a prompt processor, a policy processor, a policy validator, and a policy enforcer, in approximately 4,500 lines of code.

B. Evaluation Environments

We evaluated KUBETEUS using a high-performance server setup with an NVIDIA Hopper H100 80GB PCIe for fine-tuning LLMs, an AMD EPYC 9224 CPU, 256GB RAM, and a 2TB SSD for setting up the microservice environment. The Kubernetes cluster consisted of one master node and two worker nodes, each hosted on three Ubuntu 22.04 virtual machines. For the container-related configuration, we used containerd [45] as the container runtime and selected Cilium [31] and Calico [32] as the target CNIs to provide inter-container communication. We evaluated to generate three types of network policies: Kubernetes Network Policy, Cilium Network Policy, and Calico Network Policy. KUBETEUS was deployed on the master node, while the test applications were distributed across the two worker nodes.

We tested three cloud-native applications with different scales and functionalities: Online Boutique [46], consisting of 11 independently deployed and managed services, provides e-commerce demos such as product search, cart addition, and purchase completion. Bookinfo [47], a demo with 4 services, offers book information lookup functionality. Martian Bank [48], a financial services demo with 6 services, includes account management and transaction execution features. These applications were selected as they exemplify diverse architectural patterns in production environments, from simple service-to-service communications to complex architectures with various backend dependencies and frontend interactions, ensuring comprehensive coverage of real-world microservice deployment scenarios. This setup enabled a comprehensive evaluation of the network policy generation, validation, and enforcement processes under conditions similar to real-world microservice environments.

C. Functional Correctness

Prompt Enhancement Processing. To evaluate the prompt engineering capabilities of KUBETEUS, we analyzed the process of creating a network policy that permits traffic between specific pods. For well-defined microservices, such as Martian Bank [48], it is assumed that detailed metadata and labels are provided during service deployment. This process comprises three stages. First, in the initial intent prompt recognition, our system receives the policy requirements as the initial prompt from the user or through the automatic config mode. Second, during the information gathering stage, the system collects and analyzes detailed information about the relevant pods via the Kubernetes API. Third, in the intent prompt enhancement stage, the initial prompt is refined based on the collected information. Figure 6 illustrates the prompt engineering process and its results following the initial prompt generation. Starting with simple initial information, the system gathers and analyzes detailed data about relevant pods through the Kubernetes API. For example, as shown in Figure 6 (A)-2, our system identifies that the `app: dashboard` pod is related to the `app: atm-locator` pod. Consequently, the enhanced prompt includes important details not specified in the initial prompt, such as the specification of the `CiliumNetworkPolicy` type, accurate pod labels, and specific traffic flow directions. The accuracy of this enhanced prompt is verified through actual network logs shown in Figure 6 (B). The traffic patterns recorded in the logs precisely match the pod relationships specified in the enhanced prompt. This demonstrates that KUBETEUS's prompt engineering capability effectively generates accurate and detailed network policies in complex microservice environments.

CRD, Resource and Property Validation. This evaluation assesses the ability of our system to validate the CRD, resource, and property aspects of network policies. Figure 7 illustrates the test scenarios and results of the validation process. The top of Figure 7 presents information about the 'details' and 'productpage' pods in the 'bookinfo' namespace, both serving on port 9080/TCP. Based on this, Figure 7 (A) shows a valid network policy with correct labels, ports, and protocols, while (B) depicts an incorrectly configured policy with unsupported fields, non-existent pods, and incorrect attributes.

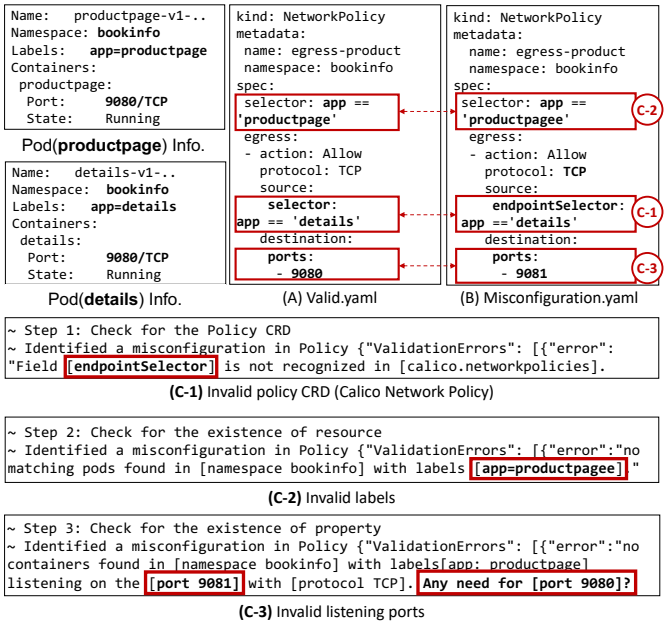

Fig. 7: The results of policy validation scenarios: valid (A), misconfigured (B), and the detection logs (C).

When attempting to enforce network policy (B), our system successfully detects a structural policy schema error during the CRD validation stage, specifically identifying the use of the unsupported field 'endpointSelector' (C-1). The system also identifies a misspelled resource label value intended for the policy ('productpagee' instead of 'productpage'), indicating that the resource does not exist within the cluster (C-2). Lastly, the system detects that port number 9081 specified in the port property of the policy does not correspond to the port serviced by the 'productpage' pod (C-3). Errors identified at each stage are immediately reported to the administrator via logs, enabling rapid identification and correction of these issues. These results demonstrate that our system effectively prevents potential errors before policy enforcement through multi-layered policy validation, ensuring accurate network security policy management in the microservice environments.

D. Performance

Classification Model Performance. In this evaluation, we tested the performance of the entity classifier by assessing its accuracy in correctly identifying key entities from intent prompts. As shown in Table I, 20% of the total intent samples were randomly selected and used as the test dataset. Each test dataset entry consisted of an intent prompt text and its corresponding list of entity tag pairs. Using this test set, we compared the performance of models based on BERT and RoBERTa. During the evaluation process, each model predicted entity tags for the given intent prompts, and these predictions were compared against the actual labels (ground truth) included in the test set. Through this comparison, we calculated each model's accuracy, precision, recall, and F1 score, as presented in Table III. The BERT-based classification model achieved 97.1% accuracy, 97.4% precision, 97.2% recall, and a 97.3% F1 score. The RoBERTa model showed similarly high performance, albeit slightly lower than BERT. Both models demonstrated F1 scores exceeding 97%, indicating a balance between precision and recall. The remaining 3% of cases typically involve ambiguous entities (e.g., service names mistaken for pod names) and non-standard format specifications (e.g., port ranges, CIDR blocks), which KubeTeus addresses through automated rule-based pattern analysis to complement the model's classification. These results suggest that KubeTeus can accurately identify crucial entities in network policies, such as pod names, IPs, and port numbers.

TABLE III: The summary of entity classifier performance with metrics: Accuracy, Precision, Recall, and F1 Score.

Model		Accuracy	Precision	Recall	F1-score
Entity Classifier	BERT	0.971	0.974	0.972	0.973
	RoBERTa	0.961	0.968	0.961	0.965

Fine-tuned LLM Performance. To evaluate the accuracy of network policies generated based on enhanced intent prompts, we compared the performance of base models and fine-tuned models using BLEU, METEOR, ROUGE-1, ROUGE-2, ROUGE-L, and chrF++ metrics. Given that even minor syntactic errors in network policies expressed in YAML can invalidate the entire configuration, accurate generation requires not only correct structure but also precise alignment with intended rules, comprehensive component coverage, and exact field naming. To assess these aspects, we used BLEU for syntactic conformity, METEOR for semantic alignment and intent preservation of policy rules, ROUGE metrics for policy component coverage, and chrF++ for character-level accuracy in field naming. The results are summarized in Table IV. This assessment aimed to evaluate the effectiveness of fine-tuning in enhancing the ability to generate network policies. All the tested models demonstrated significant performance improvements following the fine-tuning process provided by our system. Among the models, the deepseek coder 7b model exhibited the highest performance post-fine-tuning, with its BLEU score increasing by approximately 46% (from 0.52 to 0.76) and its ROUGE-1 score improving by 22% (from 0.71 to 0.87). The codegemma 7b model showed similar enhancements. These results indicate that fine-tuning has effectively specialized the models for network policy generation tasks. On average, we observed performance improvements of about 70% in BLEU scores, 25% in ROUGE-1 scores, and 15% in chrF++ scores. These substantial improvements in BLEU and ROUGE scores highlight the enhanced accuracy and completeness of the generated policies for diverse CNI environments. In conclusion, these evaluation results demonstrate the potential of our system as a reliable network security policy enforcement tool across various CNI environments.

VI. Related Work

Intent-Based Network Management. Due to the increasing complexity of network management, research on intent-based network management has been actively pursued [17],

TABLE IV: The summary of the fine-tuned LLMs performance with metrics: BLEU, METEOR, ROUGE-1, ROUGE-2, ROUGE-L, and chrF++.

Type	Model Name	BLEU	METEOR	ROUGE-1	ROUGE-2	ROUGE-L	chrF++
Baseline Model	Deepseek-coder-7b-instruct-v1.5	0.52	0.79	0.71	0.57	0.69	0.80
	Meta-Llama-3-8B-Instruct	0.19	0.36	0.53	0.27	0.44	0.44
	codegemma-7b-it	0.49	0.75	0.72	0.59	0.70	0.80
	Mistral-7B-Instruct-v0.2	0.10	0.37	0.34	0.15	0.29	0.47
	CodeLlama-7b-Instruct-hf	0.28	0.53	0.58	0.33	0.47	0.64
Fine-tuning Model	Deepseek-coder-7b-instruct-v1.5	**0.76**	0.81	**0.87**	0.78	0.83	**0.87**
	Meta-Llama-3-8B-Instruct	0.41	0.60	0.66	0.59	0.64	0.68
	codegemma-7b-it	0.72	**0.82**	0.85	**0.80**	**0.84**	**0.87**
	Mistral-7B-Instruct-v0.2	0.46	0.55	0.57	0.50	0.54	0.65
	CodeLlama-7b-Instruct-hf	0.31	0.54	0.59	0.34	0.48	0.65

[18]. Jacobs et al. [18] proposed a system that translates natural language intents into an intermediate language form, which is then converted into network configurations. The system uses machine learning and operator feedback to verify if the translated intent aligns with the operator's goals, and ensures accurate compilation and deployment in the network. Collet et al. [17] proposed an approach to automatically learn and deploy predictive models tailored to network management objectives. Research utilizing LLMs [14], [16] is also gaining attention. Dzeparoska et al. [16] employed few-shot learning with LLMs to incrementally decompose intents and automate application management through policy-based abstraction. Van Tu et al. [14] applied in-context learning with LLMs to intent-based configuration in NFV environments. This approach allows intent translation tasks to be performed without retraining the LLM, enabling learning using JSON templates. However, these studies are not well suited to the specific nature of container-based microservices (e.g., frequent updates) and require considerable effort in formulating the intent.

Advanced Network Policy Generation. Recently, several studies on automated policy generation in containerized environments have also been conducted [8], [11], [12]. Li et al. [12] developed a system that automatically generates access control policies by analyzing interactions between microservices using a static analysis-based request extraction mechanism. Xu et al. [8] proposed a method to automatically generate authorization policies based on access logs by using a log-based topology graph generation mechanism, a machine learning-based attribute mining method, and a traffic management-based policy upgrade mechanism. Lee et al. [11] introduced an automated network policy discovery framework that generates a minimal set of network security policies using network logs. Additionally, research focusing on policy verification has been proposed [49]–[51]. Li et al. [49], [50] presented a system that efficiently verifies cloud-native network policies at runtime through fast, complete, and incremental verification using a bit matrix model, including an intent-based verification language. Kang et al. [51] introduced a method for the automatic verification of container network policies using a novel graph structure representing policies. Moreover, open-source tools are being developed to manage policies [52]–[54]. Otterize [52] declaratively defines intended access policies and automatically converts them into network and Kafka policies. Open Policy Agent (OPA) [53] enables policy-based access control and verification across various environments, including cloud-native environments. GateKeeper [54] utilizes OPA to validate requests in Kubernetes clusters and block unauthorized access attempts. However, these works primarily focus on past data to generate network policies, making it challenging to reflect nuanced policy intentions.

Unlike previous works, KUBETEUS is an LLM-based network policy generation framework offering an end-to-end approach tailored for cloud-native environments. Notably, our system employs advanced prompt engineering with a specialized NER model, optimizing LLM efficiency for network policy domains. This automated approach minimizes user intervention and reduces human error while enhancing efficiency. By automating LLM improvements, KUBETEUS adapts to evolving cloud environments, enabling efficient network policy management, particularly in generating container-aware policies through real-time resource relationship inference.

VII. CONCLUSION

This paper presents an automated software framework, KUBETEUS, for generating network policies in real-world cloud-native environments. This study is the first to integrate sophisticated prompt engineering with fine-tuned LLMs specialized for complex cloud-native settings, distinguishing it from existing log-based or static analysis methods. Additionally, beyond policy generation, our system includes a multi-step validation process to prevent misconfigurations in advance. KUBETEUS has demonstrated impressive results in both entity classification and policy generation. The automated fine-tuning process for LLMs significantly improved model performance across all metrics in policy generation, demonstrating its ability to produce accurate and relevant network policies. This research provides a valuable reference implementation for intent-based security policy generation and suggests a new direction for security policy management in cloud-native environments.

The authors have made their code publicly accessible at [55] and their data at [56].

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) funded by the Korean Government (MSIT) under Grant 2022R1C1C1006093.

REFERENCES

[1] Gartner, "Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach $679 Billion in 2024," https://www.gartner.com/en/newsroom/press-releases/11-13-2023-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-679-billion-in-20240, 2023.

[2] Flexera, "2024 state of the cloud report," https://info.flexera.com/CM-REPORT-State-of-the-Cloud?lead_source=Organic%20Search, 2024.

[3] ——, "Cloud computing trends: Flexera 2024 state of the cloud report," https://www.flexera.com/blog/cloud/cloud-computing-trends-flexera-2024-state-of-the-cloud-report/, 2024.

[4] Kubernetes, https://kubernetes.io/, 2024.

[5] T. Asia, "Data breaches at Toyota: the company once again warns customers of a breach," https://techwireasia.com/2023/12/how-has-toyota-suffered-so-many-data-breaches/, 2023.

[6] R. Hat, "The state of Kubernetes security report: 2024 edition," https://www.redhat.com/en/engage/state-kubernetes-security-report-2024, 2024.

[7] H. Sun, Q. Huang, J. Sun, W. Wang, J. Li, F. Li, Y. Bao, X. Yao, and G. Zhang, "{AutoSketch}: Automatic {Sketch-Oriented} compiler for query-driven network telemetry," in 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), 2024, pp. 1551–1572.

[8] S. Xu, Q. Zhou, H. Huang, X. Jia, H. Du, Y. Chen, and Y. Xie, "Log2policy: An approach to generate fine-grained access control rules for microservices from scratch," in Proceedings of the 39th Annual Computer Security Applications Conference, 2023, pp. 229–240.

[9] A. Collet, A. Bazco-Nogueras, A. Banchs, and M. Fiore, "Automanager: a meta-learning model for network management from intertwined forecasts," in IEEE INFOCOM 2023-IEEE Conference on Computer Communications. IEEE, 2023, pp. 1–10.

[10] S. Sheng, K. Che, A. Mi, and X. Wan, "Network security mechanism optimization strategy in cloud native scenario," in 2023 6th International Conference on Electronics Technology (ICET). IEEE, 2023, pp. 614–618.

[11] S. Lee and J. Nam, "Kunerva: Automated network policy discovery framework for containers," IEEE Access, 2023.

[12] X. Li, Y. Chen, Z. Lin, X. Wang, and J. H. Chen, "Automatic policy generation for {Inter-Service} access control of microservices," in 30th USENIX Security Symposium (USENIX Security 21), 2021, pp. 3971–3988.

[13] A. Fuad, A. H. Ahmed, M. A. Riegler, and T. Čičić, "An intent-based networks framework based on large language models," in 2024 IEEE 10th International Conference on Network Softwarization (NetSoft). IEEE, 2024, pp. 7–12.

[14] N. Van Tu, J.-H. Yoo, and J. W.-K. 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. IEEE, 2024, pp. 1–8.

[15] J. Kim, B. E. Ujcich, and D. J. Tian, "Intender: Fuzzing {Intent-Based} networking with {Intent-State} transition guidance," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 4463–4480.

[16] K. Dzeparoska, J. Lin, A. Tizghadam, and A. Leon-Garcia, "Llm-based policy generation for intent-based management of applications," in 2023 19th International Conference on Network and Service Management (CNSM). IEEE, 2023, pp. 1–7.

[17] A. Collet, A. Banchs, and M. Fiore, "Lossleap: Learning to predict for intent-based networking," in IEEE INFOCOM 2022-IEEE Conference on Computer Communications. IEEE, 2022, pp. 2138–2147.

[18] A. S. Jacobs, R. J. Pfitscher, R. H. Ribeiro, R. A. Ferreira, L. Z. Granville, W. Willinger, and S. G. Rao, "Hey, lumi! using natural language for {intent-based} network management," in 2021 USENIX Annual Technical Conference (USENIX ATC 21), 2021, pp. 625–639.

[19] V. Singh and S. K. Peddoju, "Container-based microservice architecture for cloud applications," in 2017 International Conference on Computing, Communication and Automation (ICCCA). IEEE, 2017, pp. 847–852.

[20] A. Almehmadi and K. El-Khatib, "On the possibility of insider threat prevention using intent-based access control (ibac)," IEEE Systems Journal, vol. 11, no. 2, pp. 373–384, 2015.

[21] T. Ding, T. Chen, H. Zhu, J. Jiang, Y. Zhong, J. Zhou, G. Wang, Z. Zhu, I. Zharkov, and L. Liang, "The efficiency spectrum of large language models: An algorithmic survey," arXiv preprint arXiv:2312.00678, 2023.

[22] S. Ryu, H. Do, Y. Kim, G. G. Lee, and J. Ok, "Key-element-informed sllm tuning for document summarization," arXiv preprint arXiv:2406.04625, 2024.

[23] Container Network Interface, https://www.cni.dev/, 2024.

[24] S. Qi, S. G. Kulkarni, and K. Ramakrishnan, "Assessing container network interface plugins: Functionality, performance, and scalability," IEEE Transactions on Network and Service Management, vol. 18, no. 1, pp. 656–671, 2020.

[25] G. Budigiri, C. Baumann, J. T. Mühlberg, E. Truyen, and W. Joosen, "Network policies in kubernetes: Performance evaluation and security analysis," in 2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit). IEEE, 2021, pp. 407–412.

[26] R. K. Jayalath, H. Ahmad, D. Goel, M. S. Syed, and F. Ullah, "Microservice vulnerability analysis: A literature review with empirical insights," IEEE Access, 2024.

[27] Optuna - A hyperparameter optimization framework, https://optuna.org/, 2022.

[28] AutoTrain, https://huggingface.co/autotrain, 2024.

[29] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805, 2018.

[30] Extend the Kubernetes API with CustomResourceDefinitions, https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/, 2024.

[31] Cilium, https://cilium.io/, 2024.

[32] Project Calico, http, 2024.

[33] J. Li, A. Sun, J. Han, and C. Li, "A survey on deep learning for named entity recognition," IEEE transactions on knowledge and data engineering, vol. 34, no. 1, pp. 50–70, 2020.

[34] meta-llama/Meta-Llama-3-8B-Instruct, https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct, 2024.

[35] deepseek-ai/deepseek-coder-7b-instruct-v1.5, https://huggingface.co/deepseek-ai/deepseek-coder-7b-instruct-v1.5, 2024.

[36] mistralai/Mistral-7B-Instruct-v0.2, https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2, 2024.

[37] google/codegemma-7b-it, https://huggingface.co/google/codegemma-7b-it, 2024.

[38] codellama/CodeLlama-7b-Instruct-hf, https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf, 2024.

[39] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," arXiv preprint arXiv:2106.09685, 2021.

[40] Y. Wang, S. Agarwal, S. Mukherjee, X. Liu, J. Gao, A. H. Awadallah, and J. Gao, "Adamix: Mixture-of-adaptations for parameter-efficient model tuning," arXiv preprint arXiv:2205.12410, 2022.

[41] Network Policies, https://kubernetes.io/docs/concepts/services-networking/network-policies/, 2024.

[42] Cilium Network Policy, https://docs.cilium.io/en/latest/security/policy/, 2024.

[43] Calico Network Policy, https://docs.tigera.io/calico/latest/network-policy/get-started/calico-policy/calico-network-policy, 2024.

[44] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," arXiv preprint arXiv:1907.11692, 2019.

[45] containerd - An open and reliable container runtime, https://github.com/containerd/containerd, 2024.

[46] Online Boutique, https://github.com/GoogleCloudPlatform/microservices-demo, 2024.

[47] Bookinfo Application, https://istio.io/latest/docs/examples/bookinfo/, 2024.

[48] Martian Bank, https://github.com/cisco-open/martian-bank-demo, 2024.

[49] Y. Li, C. Jia, X. Hu, and J. Li, "Kano: Efficient container network policy verification," in 2020 IEEE Symposium on High-Performance Interconnects (HOTI). IEEE, 2020, pp. 63–70.

[50] Y. Li, X. Hu, C. Jia, K. Wang, and J. Li, "Kano: Efficient cloud native network policy verification," IEEE Transactions on Network and Service Management, 2022.

[51] H. Kang and S. Shin, "Verikube: Automatic and efficient verification for container network policies," IEICE TRANSACTIONS on Information and Systems, vol. 105, no. 12, pp. 2131–2134, 2022.

[52] Otterize, https://otterize.com/, 2024.

[53] Open Policy Agent, https://www.openpolicyagent.org/, 2024.

[54] GateKeeper, https://github.com/open-policy-agent/gatekeeper, 2024.

[55] KubeTeus, https://github.com/cclab-inu/KubeTeus, 2024.

[56] KubeTeus Dataset, https://huggingface.co/datasets/cclabinu/kubeteus, 2024.