Journal of Network and Computer Applications 256 (2026) 104567

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Research paper

A unified framework for cognition-driven security policy generation in cloud-native environments

Bom Kim [a], Seungwon Shin [a], Seungsoo Lee [b,*]

[a] KAIST, Daejeon, 34141, Republic of Korea
[b] Incheon National University, Incheon, 22012, Republic of Korea

ARTICLE INFO

Dataset link: https://github.com/cclab-inu/KubeTeus, https://huggingface.co/datasets/cclabinu/kubeteus

Keywords:
Cloud-native security
Container security policy
Intent-based security policy
LLM-based policy generation

ABSTRACT

The rise of containerized microservices has introduced challenges in managing security policies within cloud-native environments. Their dynamic nature complicates enforcement, while manual policy management risks misconfigurations that increase the attack surface. Existing automated policy generation methods, often reliant on historical logs and narrow scopes, struggle to meet the accuracy and heterogeneity demanded by modern security needs. In this paper, we introduce KubeTeus+, a unified framework for intent-driven security policy generation across diverse tools. KubeTeus+ employs a cognition-driven approach: it understands security requirements through domain-specific entity recognition and prompt engineering, perceives real-time cluster states, and generates accurate policies through fine-tuned large language models. This enables administrators to specify requirements in natural language for both network- and system-level policy synthesis without low-level configuration expertise. The framework further includes multi-phase validation and conflict detection to prevent misconfigurations. Our evaluations across complementary metrics show consistent improvements across both policy domains, with BLEU increasing by up to 1440% (from 0.05 to 0.77), and ablation studies confirm each component's independent contribution. This work lays the foundation for unified, intent-driven security policy automation that supports diverse policy platforms, helping address the growing complexity of policy generation.

1. Introduction

The rise of generative AI is reshaping demand for cloud computing infrastructure, marking a significant shift in the industry (Google Cloud, 2025a,b; Mckinsey & Company, 2024). As scalable infrastructure becomes essential for AI workloads, cloud infrastructure spending is rising rapidly (Intel, 2025). According to IDC's 2025 Cloud Infrastructure Spending Forecast Report, cloud infrastructure investments are expected to increase by 33.3% in 2025, reaching $271.5 billion (IDC, 2025). This growth is driven by enterprises adopting various cloud deployment models to meet complex needs, including high-performance GPU resources, data sovereignty requirements, and cost optimization. The Flexential's State of AI Infrastructure Report in 2025 shows that 60% of organizations use private cloud, 48% operate in hybrid environments, and 47% use public cloud for AI data storage (Flexential, 2025). Enabling these multi-cloud strategies are cloud-native technologies, particularly containerization, which serves as the foundation by providing application portability across diverse cloud environments. Cloud-native applications contrast with traditional monolithic architectures by leveraging the flexibility and efficiency of container technology. Notably, the emergence of container orchestration platforms, such as Kubernetes (The Kubernetes Authors, 2025), has further facilitated the management of these complex container-based microservices.

However, these advancements introduce critical security challenges, as misconfigured settings and vulnerabilities become prime targets for cyberattacks (Sun, 2020; Singh and Chatterjee, 2017; Khan, 2016; Singh et al., 2016). The Red Hat's State of Cloud Security Report (2024) (Hat, 2024) indicates that 42% of respondents identified security as the primary concern in their container and Kubernetes environments, with misconfiguration cited as a major risk factor. In fact, in 2023, Toyota experienced a data breach caused by human error in cloud configuration, which led to the unauthorized exposure of approximately 2.15 million customer records over a decade (Asia, 2023). Similarly, in 2024, the Snowflake case illustrated that a single-sign-on (SSO) misconfiguration, set to '*optional*' instead of '*mandatory*', allowed direct login using local credentials and resulted in customer data exposure (Alliance, 2025).

In microservice-based architectures, network configurations change frequently, and service interactions are inherently complex, rendering

* Corresponding author.
E-mail address: seungsoo@inu.ac.kr (S. Lee).

https://doi.org/10.1016/j.jnca.2026.104567
Received 13 November 2025; Received in revised form 8 April 2026; Accepted 6 August 2026
Available online 31 August 2026

manual configuration and management of security policies impractical. In addition, system policies that restrict file access and process execution within containers demand in-depth knowledge of application internals and container images, further complicating policy management. Thus, the inability of manual approaches to ensure policy consistency and real-time responsiveness across security domains makes systems vulnerable, as even minor errors, such as typographical mistakes or missing configurations, can compromise overall security.

To address those challenges, various studies have been proposed to automate security policy management in cloud-native environments (Xu et al., 2025; Sun et al., 2024; Xu et al., 2023; Collet et al., 2023; Sheng et al., 2023; Lee and Nam, 2023; Li et al., 2021; Ma et al., 2020; Cinque et al., 2022). These studies primarily focus on methods for automatically generating 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 activities, which may not be adequate for new types of traffic or unexpected container behaviors. 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. Recent intent-based access control (IBAC) approaches (Siino et al., 2025; Santos et al., 2025; Fuad et al., 2024; Van Tu et al., 2024; Kim et al., 2023; Dzeparoska et al., 2023; Collet et al., 2022; Jacobs et al., 2021; Yang et al., 2021) have been proposed, which express users' intents in natural language and convert them into policies. While these approaches reduce manual effort through intent-driven automation, they typically remain confined to a single security domain, such as network configuration (Han et al., 2025) or network policy generation (Kim et al., 2025b), and do not address comprehensive security policy management across heterogeneous domains. Furthermore, most existing methods validate policies individually and lack integrated mechanisms for conflict detection prior to deployment, which may result in unpredictable system behavior as policies accumulate.

In this work, we propose KubeTeus+, a unified framework for cognition-driven security policy generation tailored specifically for cloud-native environments. To realize this intent-driven approach, KubeTeus+ employs intelligent cognition-driven processing 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. Moreover, since fine-tuning is essential for the LLM to specialize in security policy generation, we provide an automatic guide function for LLM fine-tuning and hyperparameter optimization. Additionally, we develop an enhanced mechanism for the intent prompt, which is fed into the fine-tuned LLMs for generating security policies. Our system is not tightly coupled with any specific security policy provider and provides unified support for various providers, including container network interfaces (CNIs) for network policies and runtime security tools for system policies. Finally, it includes multi-step pre-validation processes and conflict detection mechanisms that detect and block incorrect policy configurations before enforcement.

In the evaluation, KubeTeus+ demonstrated comprehensive performance across six microservice applications of varying scales. The system achieved high performance with an F1 score of approximately 97% for intent and entity classification. The fine-tuned LLMs showed consistent improvements across multiple complementary metrics in generating security policies across heterogeneous provider environments, with BLEU scores improving by up to 1,440% (from 0.05 to 0.77) for system policies and 360% for network policies compared to baseline models. An ablation study on representative models further validates that both fine-tuning and prompt enhancement independently contribute to generation quality, with prompt enhancement showing the largest impact on both metric scores and structural validity. We provide a detailed breakdown of processing times to identify performance bottlenecks and their influencing factors.

This paper makes the following contributions:

- We present the design and implementation of a new security policy generation framework, KubeTeus+, capable of intelligently generating both network and system 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 across heterogeneous security policy providers.
- We present a multi-step validation process and set-theoretic conflict detection mechanism that prevents incorrect policies and inter-policy conflicts 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 security policies and detecting misconfigurations across both network and system policy domains.

The remainder of this paper is organized as follows: Section 2 provides background on cloud-native environments and the challenges of security policy management. Section 3 analyzes the structural patterns of heterogeneous security policies. Section 4 outlines the design and key requirements of KubeTeus+. Section 5 delves into KubeTeus+'s core features and operational principles. Section 6 presents the implementation and performance evaluation of KubeTeus+. Section 7 reviews related work and its limitations. Section 8 discusses system limitations and future work. Finally, Section 9 concludes the paper with a summary of key findings and implications.

2. Background and problem statements

In this section, we present the fundamental underlying cloud-native environments, including containerization and security policy mechanisms. We then discuss the key challenges in configuring security policies within these dynamic environments.

2.1. Cloud-native and containerization

Cloud-native refers to an architectural approach that leverages the elasticity and distributed processing capabilities of cloud infrastructure to build and operate applications (Zeng et al., 2023). At the core of this paradigm lies containerization, 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 the advantage of consistent execution across different environments by packaging applications and their dependencies into a single unit. Building on containerization, cloud-native applications commonly adopt microservice architecture (MSA) (Singh and Peddoju, 2017), which divides applications into independently deployable services. Each service handles specific business functions and can be developed, deployed, and scaled independently in its own container (Srirama et al., 2020). This architecture enhances scalability and maintainability but causes operational complexity in managing numerous microservices, 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.

2.2. Security policy

Security policies in cloud-native environments are rule-based mechanisms that protect resources through access control, define explicit communication flows between services, and proactively block security threats (Kim and Lee, 2024). These policies are broadly categorized into network policies and system policies, which complement each other in

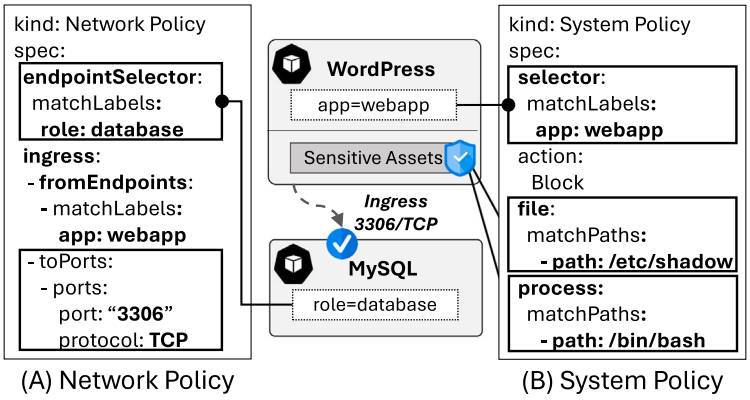


Fig. 1. The examples of two different types of security policy.

addressing different security concerns at distinct architectural layers. Fig. 1 illustrates enforcement examples of network and system policies applied to two deployed pods. Network policies control inter-service communication by explicitly defining which sources can communicate with which destinations, allowing only authorized traffic to reach specific services (Qi et al., 2020; Kim et al., 2025a). For example, Fig. 1(A) shows a network policy that permits ingress traffic from the pod labeled 'app: webapp' to the 'database' pod on port 3306/TCP while denying all other access. This prevents unauthorized pods or external entities from accessing sensitive database services, thereby protecting against data breaches and lateral movement within the cluster.

System policies control container behavior by restricting which system resources containers can access and which operations they can perform (Her et al., 2025). As shown in Fig. 1(B), the system policy for the 'webapp' pod blocks write access to the /etc/shadow file and prevents execution of /bin/bash. Since containers share the host operating system kernel, limiting access to critical system files is essential. For instance, the /etc/shadow file contains encrypted user passwords—allowing read access while blocking write access prevents malicious processes from modifying authentication credentials. Similarly, blocking shell execution prevents attackers from establishing interactive sessions even if they compromise a container. Through such restrictions, system policies mitigate risks such as container escapes and privilege escalation.

2.3. Intent-based access control using LLMs

The intent-based access control (IBAC) (Almehmadi and El-Khatib, 2015) is a promising mechanism that determines access rights by comprehensively analyzing 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 the IBAC holds significant potential. The LLMs can accurately interpret user intentions due to their advanced natural language processing capabilities, which are derived from extensive language learning models. However, the large number of parameters in most LLMs can lead to high computational costs and processing delays (Ding et al., 2023). 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 (Ryu et al., 2024). 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.

2.4. Problem statements

Generating security policies in cloud-native environments presents multiple challenges stemming from the complex and dynamic characteristics of microservice architectures, as well as the coexistence of heterogeneous security policies. This section examines four key challenges in implementing effective security policy management.

C1: Insufficient Understanding of the Dynamic Nature of Container-based Microservices. The current systems for generating security policies have limitations, particularly in reflecting the dynamic characteristics of microservice environments. A representative method is the log-based approach (Xu et al., 2023; Lee and Nam, 2023), which relies on historical network logs or system audit trails to generate policies. This approach struggles to continuously reflect the real-time relationships and communication patterns among services that frequently change. Similarly, policy generation methodologies based on static analysis of container configurations or application source code (Li et al., 2021) fail 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 evolving topology and dependency dynamics of container-based microservice ecosystems where services are continuously deployed and updated.

C2: Limitations in Supporting Heterogeneous Security Policies. Generating and managing consistent security policies in a container-based microservice environment where diverse security tools are employed is a significant challenge. Each Container Network Interface (CNI) has its own unique policy structure and functions focused on communication control, and system security tools similarly possess distinct policy schemas and enforcement models centered on resource access management, requiring administrators to understand and configure policies for each tool individually (Budigiri et al., 2021; Qi et al., 2020). This significantly increases the complexity of overall policy management, making it difficult to maintain a consistent security posture across both domains. Current automated policy generation systems are often closely integrated with specific security tools or designed to support only a single security domain. Even systems that attempt to provide unified policy management introduce their own policy schemas that administrators must learn, adding another layer of complexity rather than eliminating the heterogeneity problem. Consequently, significant administrator intervention is required to adapt these systems for compatibility with different security tools or to manage both network and system policies together. In complex applications comprising tens or hundreds of microservices, such continuous manual adjustments by administrators hinder the swift and accurate application of policies and even increase the likelihood of human error.

C3: Risk of Misconfiguration in Security Policy Generation. In a microservice architecture comprising numerous containers, complex dependencies, and frequent updates between services can lead to

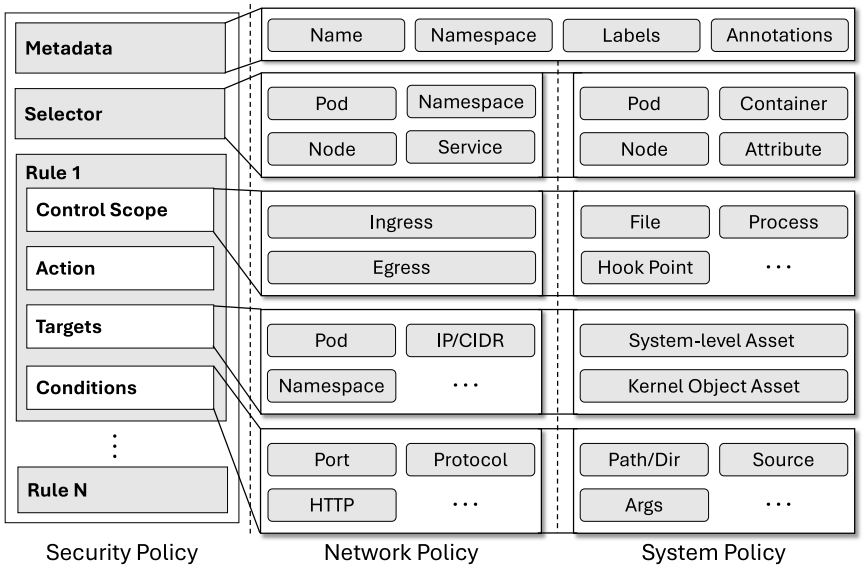


Fig. 2. The overview of the security policy structure.

misconfigurations (Jayalath et al., 2024). 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 cloud administrators. Incorrectly configured security policies may block necessary communication between services, permit unauthorized access, or enable container escape and privilege escalation by exposing shared host resources, 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.

C4: Conflicts Arising from Compositely Enforced Policies In cloud-native environments, it is common for multiple security policies to be applied simultaneously to a single resource. These policies may be created at different times by various administrators or automation tools, and represent a different dimension of problem from syntactic errors or resource mismatches in individual policies, as each policy may be independently valid but become contradictory or nullified when applied together. Several conflict patterns can emerge in such scenarios (Zahoor et al., 2023; Togay et al., 2021). Redundancy occurs when a policy's rules are entirely subsumed by another more comprehensive policy, rendering it operationally meaningless and creating unnecessary enforcement overhead. Explicit conflicts arise when policies specify opposing actions for identical resource selectors. Correlation conflicts emerge when policies partially overlap in their target specifications or enforcement scopes, creating ambiguous cases where the actual enforcement behavior becomes unpredictable. These conflicts cannot be detected by validating individual policies alone and can only be identified by examining related policies together, but as policies accumulate over time, manually identifying such conflicts among the expanding policy set becomes exceedingly difficult. Current automated policy generation systems lack integrated mechanisms that both generate policies and detect conflicts with existing ones before deployment. This absence of proactive conflict detection leaves the security posture of clusters dependent on reactive incident response, rather than preventive policy management, undermining the reliability of the overall security infrastructure.

3. Security policy structure analysis

Intent-driven policy automation systems should transform user intentions expressed in natural language into executable policy specifications. To design such systems, we must first understand the structure of the security policies: (i) what architectural patterns these policies follow, (ii) how their components are organized, and (iii) which fields are syntactically required versus semantically meaningful. This section examines network and system policies to identify the structural patterns and component relationships that aim to identify the essential elements whose omission introduces ambiguity in intent specification.

3.1. Common architectural patterns

Although policy implementations vary across security tools (van Vugt and Malik, 2023), a consistent three-tier architectural pattern is evident. Fig. 2 shows this structure, which includes Metadata, Selector, and Rule components. Rules are further divided into Control Scope, Action, Targets, and Conditions.

The metadata contains administrative information that establishes policy identity and organizational context, including the policy name, namespace specification, labels for categorization, and annotations for auxiliary information such as creation timestamps or ownership details. While metadata does not directly influence enforcement logic, it enables policy lifecycle management, version tracking, and discovery. The selector determines the policy subject, the entity to which the policy applies. In network policies (Kim et al., 2025a), selectors identify communication endpoints to be governed, while in system policies (Her et al., 2025), selectors target runtime entities whose behavior will be controlled. The rule structures control objectives through four components. The control scope specifies the operational boundary where the rule applies, defining the dimension of system behavior that the policy controls. The action defines the enforcement decision when rule conditions are satisfied, determining how the system responds to matching events or requests. The targets identify the specific entities or resources that the policy subject interacts with or accesses, establishing the object of the control. The conditions establish the matching criteria that trigger rule evaluation, specifying the attributes or parameters that must be present for the rule to apply.

We examine network policies (NP) including Kubernetes NP (The Kubernetes Authors, 2025b), Cilium NP (Cilium Authors, 2025b), and Calico NP (Tigera, 2025a) that focus on communication control, as well as system policies that secure container runtimes through various methods: KubeArmor Policy (The KubeArmor Authors, 2025b) restricts resource access, while Tetragon Tracing Policy (The Tetragon Authors, 2025b) traces execution at the kernel level. Although these follow a common architectural pattern, they differ in identifying policy subjects,

Table 1
The comparison of Network Policy Structures across Kubernetes (The Kubernetes Authors, 2025), Cilium (Cilium Authors, 2025a), and Calico (Tigera, 2025b) (NP: Network Policy, NS: Namespace, SA: Service Account).

Aspect	K8s NP (The Kubernetes Authors, 2025b)	Cilium NP (Cilium Authors, 2025b)	Calico NP (Tigera, 2025a)
Principle	Label-based isolation	Endpoint and Entity-aware control	Expression-based control
Policy Scope	NS	NS, Cluster-wide	NS, Cluster-wide
Selector Type	Label-based	Label-based	Label-based, Expressions-based (CEL)
Policy Subject	Pod, NS	Endpoint, NS, Entity	Pod, NS, SA, VM, Host
Control Scope	Ingress, Egress	Ingress, Egress, L7	Ingress, Egress, L7
Rule Targets	Pod, NS, IP, CIDR	Endpoint, Entity, IP, CIDR , FQDN	Pod, NS, SA, VM, Host, IP, CIDR, FQDN
L3/L4 Support	Port/Protocol, IP, CIDR	Port/Protocol, IP, CIDR, FQDN, Entity	Port/Protocol, IP, CIDR, Network Set
Protocol	TCP, UDP, SCTP	TCP, UDP, SCTP, ICMP, ANY	TCP, UDP, SCTP, ICMP
L7 Support	None	HTTP, DNS, Kafka, gRPC	HTTP
Action	Allow (implicit deny)	Allow, Deny	Allow, Deny, Log, Pass
Priority	None	Deny-first Precedence	Order (Numeric)

Table 2
Comparison of System Policy Structures across KubeArmor (The KubeArmor Authors, 2025a) and Tetragon (The Tetragon Authors, 2025a) (NS: Namespace).

Aspect	KubeArmor Policy (The KubeArmor Authors, 2025b)	Tetragon Tracing Policy (The Tetragon Authors, 2025b)
Principle	Label-centric System Control	Attribute-based Kernel Tracing
Policy Scope	NS, Cluster-wide, Host-wide	NS, Cluster-wide
Selector Type	Label-based	Attribute-based, K8s-aware filtering
Policy Subject	Pod, Container, Node	Process, Binary, Container, NS, PID
Control Scope	File, Process, Network, Capability, Syscall	Hookpoint (Kprobe, Tracepoint, Uprobe, Syscall)
Rule Targets	File Paths, Process Binaries, Network Protocols, ...	Function Arguments (Kprobe/Uprobe), Syscall Arguments, Return Values, Binary Paths, Kernel Events, ...
Action	Allow, Audit, Block	Sigkill, Signal, Override, Post, Trace, ...
Priority	None	Context-based or Hook-order Priority

defining control boundaries, specifying interaction targets, and enforcing decisions. Table 1 and 2 summarize these differences, which we analyze below.

3.2. How policies identify and select their subjects

All policies begin by identifying their subjects, the resources to which enforcement rules will apply. Selector types vary along a spectrum from simple label matching to complex expression-based and attribute-based targeting.

Network policy selectors progress through three levels of expressiveness. Kubernetes NP uses straightforward key–value label matching where selectors directly correspond to pod and namespace labels, maintaining simplicity but limiting targeting to exact matches. Cilium NP extends this foundation through two complementary mechanisms. First, its endpoint selectors enables label-based selection across heterogeneous resources—pods, virtual machines, and host network interfaces—where administrators specify label selectors to identify target resources. Second, entity selectors bypass label-based targeting entirely by providing direct references to predefined network scopes: `host` directly designates the node itself, `world` designates external networks, and `cluster` designates all cluster resources, eliminating the need to enumerate IPs or construct label selectors for these common network boundaries. Calico NP is not only label-based but also adopts CEL (Common Expression Language), allowing complex logical conditions that combine multiple attributes using AND, OR, and NOT operators. This enables selectors to specify nuanced conditions, such as selecting pods within a namespace that match pattern C, have label A, but do not have label B.

In system policies, KubeArmor Policy maintains consistency with Kubernetes resource models by using label-based selectors for pods, containers, and nodes. This supports policies that fit cluster-native deployment practices and allows policies at the namespace, cluster, or host level. In contrast, Tetragon Tracing Policy uses attribute-based targeting, identifying subjects by runtime characteristics such as binary paths, command-line arguments, parent process relationships, container identifiers, and process IDs. This approach enables fine-grained control based on execution context, independent of pod labels, and can specify cases such as "processes in namespace X executing binary Y with argument Z".

3.3. What policies control and who they interact with

Rules define control boundaries and interaction targets. Network policies structure control through traffic directionality and layer-specific specifications, while system policies differ in whether they restrict resource access or trace kernel execution, leading to approaches in target specification and conditions.

Network policies establish control scope through explicit traffic directionality. Ingress rules govern incoming traffic to policy subjects, and egress rules manage outbound traffic. Target specification mainly relies on selector-based mechanisms that query resources matching specific labels. Policies also provide auxiliary support for specific scenarios. All policies support IP-based targeting through CIDR ranges to control external cluster traffic. Cilium provides FQDN-based specifications that reference external services through domain names, with the policy engine monitoring DNS queries and dynamically generating traffic control rules based on resolved IP addresses. Calico supports FQDN targeting only in its enterprise edition. Calico also offers network sets that group IP ranges and enable reference via CEL expressions for managing large-scale IP groups. At the L7 layer, support diverges significantly. Cilium provides the broadest protocol-specific controls, including HTTP path and header matching, DNS query filtering, Kafka topic authorization, and gRPC method restrictions. Calico restricts L7 capabilities to HTTP traffic control in its enterprise edition, while Kubernetes NP provides no application-layer traffic management. Matching conditions across all policies involve port numbers and protocol types (TCP, UDP, SCTP,

ICMP). L7-capable policies add HTTP-specific attributes such as request methods and URL paths.

System policies embody two fundamentally different control paradigms. KubeArmor Policy organizes control scope around resource categories, including file system operations, process executions, network activities, Linux capabilities, and system calls. Within each category, rules specify targets through paths, binary names, network protocols, and capability identifiers. File operation controls use prefix or recursive matching to restrict access to specific paths. Process controls limit the execution of particular binaries or prevent the spawning of interpreter shells. Capability controls prevent containers from acquiring privileged kernel capabilities that could enable container escape or privilege escalation. Tetragon Tracing Policy controls around kernel instrumentation points, including kprobes for kernel functions, tracepoints for static kernel events, uprobes for user-space functions, and system call hooks. Rather than specifying resource paths, policies define hooks into kernel execution flows with targets specified through function arguments, return values, and kernel event parameters that serve as matching conditions for runtime filtering based on execution context. This hook-based model provides observability into kernel behavior beyond resource path specifications, such as detecting privilege escalation attempts through capability checks or identifying memory manipulation through specific sequences of kernel functions.

3.4. How policies make decisions and establish priorities

Policies determine enforcement responses through action models and, when necessary, establish rule precedence through priority mechanisms. Action semantics vary between network and system policies, where network policies primarily provide traffic control decisions while system policies distinguish between preventive enforcement and runtime observability.

Network policies share a common foundation: all providers implement an allowlist model in which traffic is permitted only when explicitly allowed, with unmatched traffic implicitly denied by default. However, they differ in how these actions are defined and enforced. Kubernetes NP implements this model without explicit action fields, where denial is an emergent enforcement behavior of the policy engine rather than an expressible rule action. In practice, a policy selecting all pods with no ingress or egress rules specified effectively enforces a default-deny posture by isolating the selected pods from all traffic. While it provides no priority fields, multiple policies targeting the same resource are evaluated as a union. Beyond this, Cilium and Calico introduce explicit deny as an expressible rule action, enabling finer-grained enforcement beyond implicit denial alone. Cilium NP expresses this through dedicated spec fields (ingress, egress, ingressDeny, egressDeny), where any explicitly expressed deny rule takes precedence over allow rules, regardless of policy order, although it provides no per-policy priority fields. Calico NP provides four actions in its action field: allow and deny, log enables traffic observation without enforcement, and pass introduces hierarchical policy evaluation by terminating processing at the current tier and delegating decisions to subsequent tiers or default behaviors. It supports per-policy numeric priority specifications where lower values are evaluated first.

When policies from multiple providers coexist in the same cluster, however, their relative enforcement order is determined by the position of each provider's enforcement mechanism within the kernel's packet processing pipeline (Kim et al., 2025a). Kubernetes NetworkPolicy rules are enforced through iptables chains that are traversed before those of other policy providers, whose enforcement hooks are attached at subsequent points in the processing flow. Consequently, Kubernetes NetworkPolicy is evaluated first; packets dropped at this stage do not reach subsequent provider-level enforcement, while packets permitted here are subject to further evaluation by the policies of other providers.

System policies embody distinct action philosophies. KubeArmor Policy provides three actions: allow, block, and audit, where audit logs operations without blocking. While it lacks explicit priority fields, it is determined by the policy type (allow versus block) and the default security posture configured at the namespace or cluster level. Policy specificity further refines precedence, with more detailed rules automatically taking precedence over broader patterns. Tetragon Tracing Policy separates enforcement actions from observability. Enforcement actions include sigkill for immediate process termination, signal for delivering specific signals to target processes, and override for modifying system call return values or arguments. For observability, event tracing serves as the default behavior where all matching kernel events are automatically traced and logged when no explicit enforcement action is specified, with the post action extending this by exporting detailed event data to external analysis systems. Priority is determined by hook attachment order and the specificity of hook conditions when multiple hooks match the same kernel event.

Table 3
Impact of missing information elements on the completeness of network policy intent prompts (✓: fully specified, ▲: partially specified, ✗: not specified).

Essential Element	Minimal (Level 1)	Partial (Level 2)	Complete (Level 3)
Policy Subject	✓	✓	✓
Action	✓	✓	✓
Traffic Direction	✓	✓	✓
Target Specification	▲	✓	✓
Port Information	✗	✓	✓
Protocol	✗	▲	✓
Subject Labels	✗	✗	✓
Target Labels	✗	✗	✓
Policy Type	✗	✗	✓

3.5. Essential elements and ambiguity patterns

The structural analysis shows that security policies share common architectural patterns. This consistency helps identify which elements are mandatory for enforcement and which are optional refinements. Users typically express policy requirements by referencing these structural components in natural language. However, omitting these elements creates ambiguity, reducing the reliability and accuracy of automated policy generation. We categorize the ambiguity caused by omitted elements and examine how specification completeness impacts policy generation reliability, using network policies as an example. Table 3 presents the essential elements for network policies, including policy type, subject, action, traffic direction, target specification, label information, port, and protocol. We examine controlling egress traffic from database (postgresql) pods to web service (nginx) pods across three levels.

At the minimal specification level (Level 1), only foundational elements may be provided, such as allowing egress traffic from a postgresql pod to an endpoint. This exhibits three types of ambiguity. Scope ambiguity arises from the absence of a target specification, leaving unclear whether the intent is to allow all outbound traffic or only traffic to specific destinations. Identification ambiguity arises from missing label specifications, rendering it impossible to construct accurate selectors since policy names do not guarantee a match with actual resource labels. Control scope ambiguity results from missing port and protocol information, preventing the precise definition of which services and traffic patterns to control. Additionally, an unspecified policy type complicates translation into the appropriate format for the target environment.

At the partial specification level (Level 2), more concrete information may be provided, such as allowing the postgresql pod to communicate with the nginx pod on port 80. The defined destination and communication unit reduce scope ambiguity compared to Level 1. However, significant ambiguity remains. Missing protocol specification creates uncertainty about whether the port applies to TCP, UDP, or

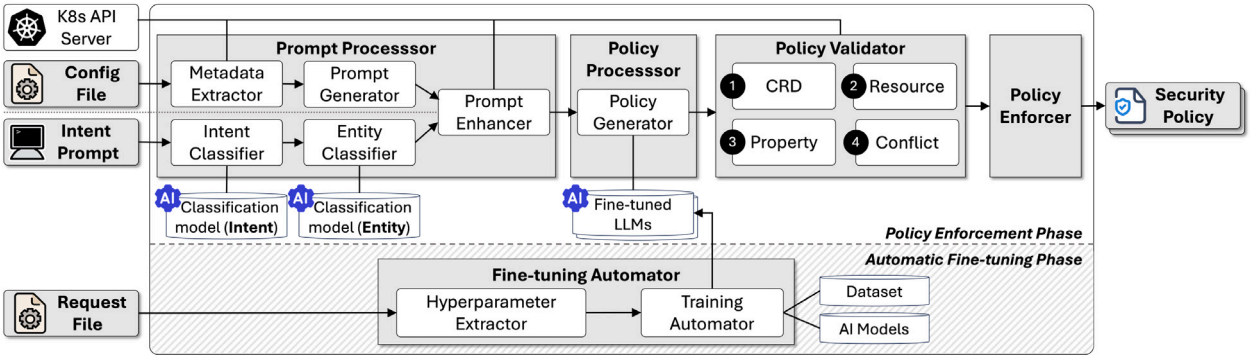


Fig. 3. The overall architecture and its workflow of KubeTeus+ with five key components: (*i*) fine-tuning automator, (*ii*) prompt processor, (*iii*) policy processor, (*iv*) policy validator, and (*v*) policy enforcer. Additionally, our system includes two operational phases: automatic fine-tuning and policy enforcement.

other protocols, potentially resulting in overly broad enforcement. More critically, label specifications are still absent, which sustains semantic uncertainty regarding resource scope, despite concrete pod names reducing some identification ambiguity. At the complete specification level (Level 3), all essential elements are explicitly provided, such as a Cilium Network Policy that allows TCP traffic on port 80 from pods labeled 'db: postgresql' to pods labeled 'app: nginx'. This eliminates all ambiguities—labels enable precise resource identification, protocol defines the traffic pattern clearly, and policy type ensures correct format output. Only at this level can accurate policies matching user intent be generated.

This analysis demonstrates that security policy generation requires information across multiple dimensions: scope specification, resource identification, control boundaries, and format. As essential elements are progressively omitted, ambiguity compounds across these dimensions, ultimately compromising the transformation from user intent to accurate executable policy. The categorization of ambiguity patterns across specification levels establishes the foundation for designing entity extraction mechanisms that identify essential elements from natural language and prompt enhancement strategies that systematically resolve specification incompleteness.

4. KubeTeus+ designs

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 security policies tailored for cloud-native environments, which are characterized by their dynamic and complex nature.

4.1. Design considerations

R1: Advanced Intelligent Policy Generation. First, it should accurately recognize user intentions and simultaneously understand the cluster states in real time to intelligently generate security policies. The framework must comprehend not only the communication patterns between microservices but also their runtime behaviors and resource access requirements to produce comprehensive security control. Additionally, it should provide a highly automated interface that analyzes service relationships and runtime security posture within the cluster using only the configuration files, minimizing user intervention. To achieve this unified intelligent policy generation, we adopt natural language processing technologies, including large language models and prompt engineering, to interpret the security requirements of users across multiple policy domains and convert them into specific policies tailored to the dynamic characteristics of cloud-native environments.

R2: Support for Diverse Security Policy Providers. Second, it should support various security policy providers, including Container Network Interfaces (CNIs) for network policies and runtime security tools for system policies, without being dependent on any specific implementation. To achieve this, the system should automatically recognize the unique characteristics and functions of each policy provider and generate security policies optimized for the cluster environment. Additionally, we provide a methodology for maintaining consistency by automatically detecting changes in policy providers and distributing corresponding security policies, taking into account the dynamic nature of container and microservice environments. This approach enables users to create and manage security policies consistently without the need to manually configure complex settings for each provider employed.

R3: Multi-step Policy Correctness Validation. Third, it should include a mechanism to verify whether the generated security 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, it checks the syntactic correctness of the security 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 security policies are enforced, thereby preventing service disruptions or security vulnerabilities that could arise from incorrect policy configurations.

R4: Policy Conflict Detection. Fourth, it should include a mechanism to verify that the newly generated security policy does not conflict with existing policies within the same policy domain before enforcement. For this, the system employs an adaptive conflict detection approach that is not tied to any specific policy type, performing set-theoretic formal analysis of policy relationships across multiple dimensions where dimensions represent the comparable attributes defined by each policy type. It first identifies whether existing policies target the same resources, then analyzes their rules to detect three types of conflicts. This proactive conflict detection mechanism enables administrators to maintain policy consistency and prevent contradictory enforcement behaviors before policies are deployed to the cluster.

4.2. System architecture and workflow

This section presents the overall architecture and workflow of KubeTeus+. As illustrated in Fig. 3, 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 comprises a hyperparameter extractor

Table 4
Summary of datasets for security policy generation and entity classification.

Type	Sample	Structure	#Origin	#Train	#Test	#Total
Network	Policy	{intent, policy}	857	132,899	33,225	166,124
	Intent	{type, intent, tags}		40,048	10,012	50,060
System	Policy	{intent, policy}	739	72,960	18,340	91,300
	Intent	{type, intent, tags}		40,048	10,022	50,070

and a training automator. The hyperparameter extractor analyzes the configuration file containing fine-tuning parameters provided by the cloud administrator. If specific parameters are not specified by the administrator, it automatically extracts optimized hyperparameters using Optuna (Akiba et al., 2019), which efficiently identifies the optimal parameter combination through 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, the training automator utilizes AutoTrain (Hugging Face, 2025a) to automatically fine-tune a specific LLM. The fine-tuned model is then used to generate security 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 security 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 pods in the cluster. In the user mode, the user provides the policy intent in natural language without needing to understand complex policy syntax. The *prompt processor* first employs an intent classifier based on Bidirectional Encoder Representations from Transformers (BERT Devlin et al., 2018), as shown in Fig. 4, to perform binary classification determining whether the user intent is requesting network policy generation or system policy generation. Following this classification, an entity classifier, which adopts the same BERT-based architecture but with different output layers for entity type classification, extracts the necessary entities from the input sentence.

In config mode, the metadata extractor 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. Based on the extracted information and cluster state, the system analyzes service relationships and communication patterns associated with the target resources for network policy generation, or leverages vulnerability scanning tools (e.g., Grype Anchore, 2025) to conduct vulnerability analysis on container images, identifying vulnerable files and processes for system policy generation. The prompt generator then transforms this structured resource information into natural language, automatically generating an initial prompt using predefined templates that correspond 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 security policies optimized for the cluster's current state. The generated policies are then converted into the formats compatible with each security policy provider by the policy converter.

The generated policy is validated by the *policy validator* before being enforced in the cluster, following a four-step process. The custom resource definition (CRD The Kubernetes Authors, 2025a) 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. The property validator ensures the policy is accurately applied to the intended resources by verifying that the policy details align with the affected resources. Finally, the conflict validator detects conflicts with existing policies within the same policy domain by comparing policy subjects and rule specifications based on set-theoretic verification. If a validation failure is detected, an error is logged, and a correction plan is provided based on its cause. The security 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.

5. KubeTeus+ details

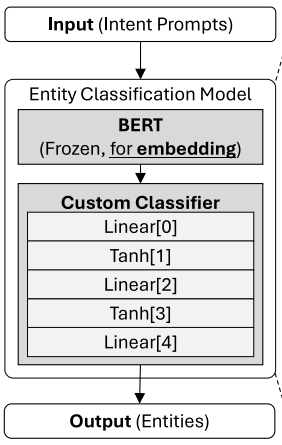
5.1. Generative AI-based model learning

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

Dataset Construction and Preprocessing. Since existing general language model datasets do not adequately reflect the specialized structure and terminology of security policies for cloud-native environments, we developed a custom dataset to train the learning model on various security policy structures and grammars from diverse policy providers. For this, we collected actual security policies from open sources, enabling the model to learn real-world network and system 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, we first collected network policies of Kubernetes (The Kubernetes Authors, 2025), Cilium (Cilium Authors, 2025a), and Calico (Tigera, 2025b) CNIs, along with system policies of KubeArmor (The KubeArmor Authors, 2025a), from open-source platforms such as GitHub to construct the initial dataset. As shown in Table 4, a total of 857 original network policies and 739 original system policies were collected and parsed based on their policy structures to separate them into selectors and rules. During this process, we removed duplicate elements through exact-match comparison across all fields to ensure data uniqueness, and the resulting samples were randomly partitioned into training and test sets at an approximately 8:2 ratio. Next, we applied shuffling and combination to the refined data to increase the dataset size. We applied semantic-preserving transformation principles (Zhang et al., 2025) and performed independent recombination of selectors and rules based on these principles. This ensured semantic validity by maintaining functional consistency between selectors and rules, and all resulting policies were verified to be deployable in real cluster environments.

This approach resulted in 166,124 network policy samples and 91,300 system policy samples. These samples were constructed as input and output pairs (intent prompt, policy output) for fine-tuning LLMs to enable them to generate security policies from natural language descriptions. And, we trained the classification models by randomly sampling approximately 50,000 intent prompts from each policy type, ensuring balanced representation across both network and system domains. For these intent samples, we performed two types of labeling.



	Type	Entity Tag	Example
1	common	POLICY	network/system policy
2		LABEL	app: nginx1
3		POD_NAME	nginx1 pod
4		NAMESPACE	kube-system
5		ACTION	allow, deny, audit...
6	network	PORT	80, 443
7		PROTOCOL	TCP, UDP, ICMP
8		TRAFFIC_DIRECTION	ingress, egress
9		CIDR	192.168.0.0/16
10		ENDPOINT	endpoints
11		HTTP_PATH	/api/v1
12		HTTP_METHOD	GET, POST
13		DOMAIN	example.com
14	system	RESOURCE_TYPE	process, file, ...
15		FILE NAME	/etc/shadow, /etc/resolv.conf
16		PATH	/bin/wget, /var/log/

Fig. 4. The design of the entity classification model for security policy-specialized NER.

First, we labeled each prompt with its policy type (network or system) to train the intent classifier for binary classification. Second, we assigned entity labels corresponding to 16 entity types (Fig. 4) using the BIESO tagging system (J. Li et al., 2020) in NER, enabling the entity classifier to identify policy-relevant components. This labeling task was performed through a combination of automated scripts and manual review, where the structural diversity of the collected policies naturally promotes linguistic variation across the generated prompts, mitigating the risk of template overfitting.

Domain-Specific NER. To generate accurate security 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 Fig. 4. This model is specialized for the security policy domain and can effectively identify the policy-related entities that general NER models could miss across both network and system policy contexts. 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 an alternating linear-tanh structure, comprising five layers (as shown in Fig. 4). The linear layers progressively transform BERT embeddings into task-specific feature representations optimized for entity classification, while tanh activations introduce non-linearity that enables complex decision boundaries for distinguishing between similar entity categories. The tanh function was chosen over alternatives like ReLU because its zero-centered bounded activation stabilizes gradient flow during training and prevents activation saturation that could hinder learning subtle distinctions between entity types. The five-layer configuration enables hierarchical feature learning where initial layers identify entity boundaries, intermediate layers distinguish coarse-grained categories (common vs. network vs. system entities), and the final layer performs fine-grained classification. A shallower three-layer architecture would lack sufficient representational capacity to differentiate the diverse entity classes, particularly when entities share similar surface forms but serve different semantic roles in policy specifications. Conversely, deeper networks with seven or more layers risk overfitting on the training data and may suffer from vanishing gradients despite tanh's relatively stable gradient properties.

The model is trained to identify 16 entity classes organized into three categories: common entities shared across both policy types, network-specific entities such as TRAFFIC_DIRECTION and PROTOCOL, and system-specific entities such as FILE_NAME and PATH. These domain-specific entities, not covered in general NER systems, were selected based on the structural analysis in Section 3 and significantly improve the accuracy of prompt calibration and the security policy generation processes.

Automatic LLM Fine-tuning. The intent prompt is refined through the NER process described above and then transformed into a security 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 fine-tuning efficiently. For this, we utilize Low-Rank Adaptation (LoRA Hu et al., 2021). 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 (Akiba et al., 2019) 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% Hu et al., 2021. 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 5. 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 security policies. As a result of the automated fine-tuning, we effectively specialize the model for security policy generation, updating only about 0.5% of the overall model parameters. This approach enables rapid adaptation and expansion when adopting new LLMs.

5.2. Customized prompt engineering

KubeTeus+ achieves accurate security 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 policy providers. Fig. 5 shows how our system refines the initial intent prompt using network policy as an example, though the process applies similarly to system policies. First, the system performs binary classification to determine the policy type (A-1). This classification directs the subsequent processing pipeline to apply the appropriate entity extraction

Table 5
The summary of LLMs fine-tuned for security policy generation.

Provider	Model name	Size	#Params
Meta	Meta-Llama-3-8B-Instruct (Meta AI, 2025a)	16 GB	8.03B
DeepSeek	Deepseek-coder-7b-instruct-v1.5 (DeepSeek AI, 2025)	14 GB	6.91B
MistralAI	Mistral-7B-Instruct-v0.2 (Mistral AI, 2025)	15 GB	7.24B
Google	Codegemma-7b-it (Google, 2025)	17 GB	8.54B
CodeLlama	CodeLlama-7b-Instruct-hf (Meta AI, 2025b)	14 GB	6.74B

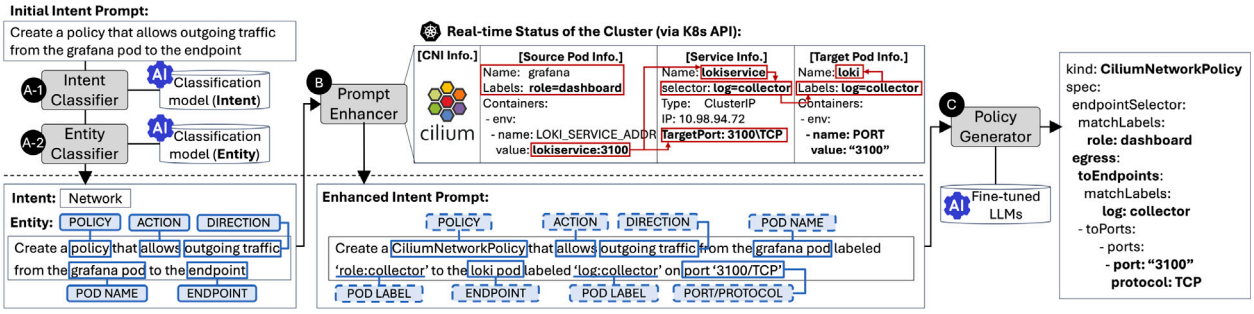


Fig. 5. The example procedure for generating a network policy, from prompt enhancement to the use of fine-tuned LLMs. The entities in solid lines are reconfigured to entities in dotted lines.

and enhancement strategies for each policy type. Following this classification, the key entities are extracted from the initial intent prompt using the domain-specific NER model (A-2). In this example, entities critical to the network policy, such as 'grafana' (POD_NAME), 'outgoing' (DIRECTION), and 'endpoint' (ENDPOINT), are accurately identified from the initial prompt.

Next, the real-time status of the cluster is considered based on the extracted entities. First, our system automatically detects the CNI currently installed in the cluster via the Kubernetes API. This detection is achieved by querying the cluster configuration or checking for the existence of specific CNI-related custom resources. Then, the system queries detailed information about the related resources through the Kubernetes API, such as the label of the 'grafana' pod (role=dashboard) and associated service information. This step also includes inferring the relationships between these resources—for instance, identifying the target 'loki' pod through service endpoint matching and tracing the communication path based on the service's selector and port configurations. 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 policy providers employed in a cluster have their own grammar and functions, complicating security policy generation. During the intent prompt engineering step, the system identifies provider-specific keywords included in the prompt and automatically converts them to match the detected provider. For network policies, the term 'endpoint' is a Cilium-style keyword. However, if the currently installed CNI is Calico, the prompt enhancer automatically converts the term to the Calico equivalent, 'podSelector.' Similarly, for system policies, provider-specific resource specifications are normalized—for instance, resolving abbreviated file names into complete paths compatible with the target policy provider's schema. As a result, the fine-tuned LLM generates a policy that adheres to the target provider's grammar and structure based on this enhanced intent prompt. This approach allows users to generate correct policies without needing to know the syntax of a specific provider or when using terminology different from the provider currently installed on the cluster. Furthermore, when the policy provider used in the cluster transitions to a different provider, the system automatically converts the existing policies to align with the new provider while preserving their original intent. For instance, network policies can be migrated from Cilium to Calico format by transforming provider-specific constructs while maintaining the underlying security semantics.

5.3. Multi-step policy correctness validation

KubeTeus+ implements a four-step validation procedure to ensure the safety of security policy enforcement: custom resource definition (CRD) validation, resource validation, property validation, and conflict validation. Among these, the first three steps covered in this section focus on verifying the correctness of individual policies themselves. The fourth step, conflict validation, focuses on proactively detecting logical conflicts between policies and is addressed separately in Section 5.4. However, all validation steps are applied not only to policies generated by our system but also to those written by administrators, ensuring thorough validation of all security 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, CRD validation, checks the grammatical correctness and structural integrity of the policy. As shown in Fig. 6, it verifies the existence and format of required fields such as 'kind', 'metadata', and 'spec', ensuring that the structure complies with the policy provider's CRD schema. This includes validating selector structures and rule specifications, such as 'podSelector' and 'egress' rules for network policies, or 'process' and 'matchLabels' for system policies. 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, it verifies whether a pod with the label 'app: httpd' is present in the cluster.

Finally, property validation checks whether the detailed properties of the policy align with the actual runtime environment. For network policies, 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 validates IP ranges specified

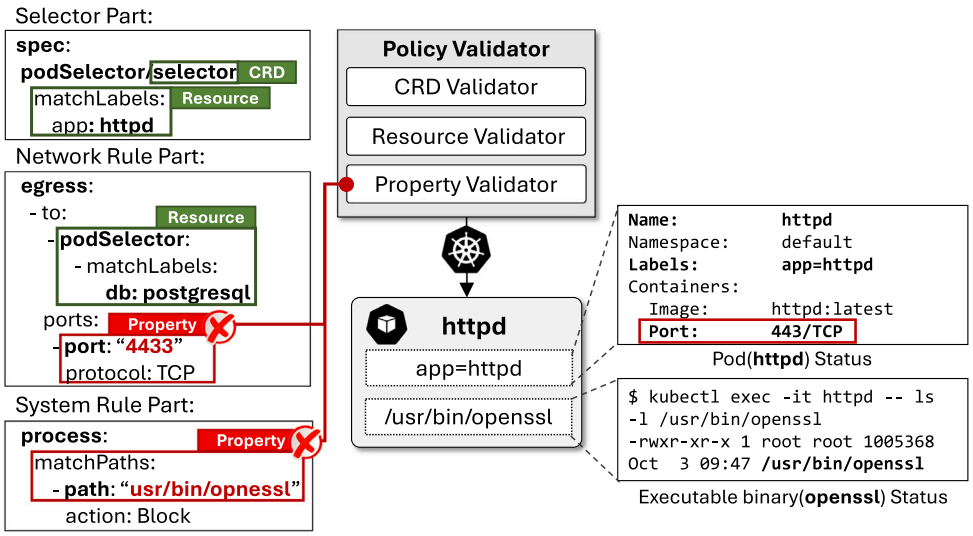


Fig. 6. The example of multi-step validation for detecting the misconfigured security policy.

in the policy to ensure they accurately target the intended network segment. For system policies, this includes confirming that target file paths or directories actually exist in the container, verifying that specified processes or binaries are present and executable in the container image, and checking that file permissions match the policy specifications.

For example, the port 4433 specified in the network policy's rule does not match the actual listening port 443 of the target httpd pod, leading to error detection and prevention of policy enforcement. The system then calculates the correlation between the error log and actual resource properties to suggest the correct port information. In the system policy case, the specified executable path '/usr/bin/opnessl' contains a typo that does not match any binary in the container image. The system identifies the closest matching path '/usr/bin/openssl' and suggests this correction to the administrator.

5.4. Policy conflict validation

To detect the three types of policy conflicts discussed in C4 (Section 2.4), KubeTeus+ proposes a set-theoretic conflict verification process. Extending beyond the individual policy validation (Section 5.3), this process models relationships between the newly generated policy and existing policies through formal set operations, identifying potentially conflicting rule pairs before deployment to the same resource. Rather than relying on simple string comparisons, this approach generalizes set relationships between fields to adaptively apply to each policy type. This process is detailed in Algorithm 1.

The algorithm operates on two inputs: a newly generated policy P_{new} and a set of existing policies P_{existing} within the same policy domain. Each policy P is represented as a tuple of selector S and rule set R, where each rule $r \in R$ is a field vector $r = (f_1, f_2, \ldots, f_n)$ with $f_i \in F$. The comparable field set F defines the subspace of attributes used for conflict analysis and varies by policy type: for network policies, F includes direction, target, port, and protocol; for system policies, F includes scope, resource type, path pattern, and source; for tracing policies, F comprises hook point, condition, function arguments, and return value. This formulation assumes that although each policy type has a distinct schema, they can be mapped onto a homomorphic comparison space defined by their respective field sets. When constructing the comparison space, each rule's action field *r.action* is populated in accordance with each provider's action vocabulary (Section 3.4): for providers that support explicit deny action fields, *r.action* is set directly from the rule specification, while for providers that express denial implicitly through rule structure, *r.action* is set to *deny* for rules covering traffic not explicitly permitted (i.e., the complement of the explicitly allowed set).

The algorithm begins by extracting the comparable field set F specific to the new policy's type, establishing the comparison space for subsequent set operations (line 1). This algorithm focuses on intra-type conflicts within the same policy domain, allowing existing policies to be iterated over to compare only policies of the same type. It filters by type to skip policies of different types (lines 2–5). For each type-matched policy P_i, the algorithm identifies the potential conflict zone through selector overlap analysis. The selectors S_{new} and S_i represent resource sets such as pods targeted by each policy (lines 6–7). If their intersection is empty, the policies govern disjoint resource spaces with no conflict possibility, and the algorithm skips to the next policy (lines 8–9). Conversely, a non-empty intersection indicates that both policies may apply simultaneously to the overlapping resource subset, representing a potential conflict zone (line 11). The algorithm then extracts rule sets scoped to this overlapping zone (lines 12–13). With the conflict-relevant rule sets identified, it performs three conflict detection procedures operating at different granularities. Redundancy detection operates at the policy level, verifying whether every rule in the new policy is entirely redundant with some rule in the existing policy through exact equality across all fields (lines 14–18). Specifically, the algorithm checks whether for every rule $r \in R_{\text{new}}$, there exists a corresponding rule $r' \in R_i$ such that $r[f] = r'[f]$ holds for all comparable fields $f \in F$ (line 15). This strict equality ensures genuine semantic equivalence. When complete redundancy is confirmed, the new policy introduces no additional enforcement semantics beyond what P_i already provides, and the algorithm records a redundancy conflict with the overlapping selector context (line 16) and sets a flag to prevent double-counting in subsequent checks (line 17).

Algorithm 1: Policy Conflict Detection through Set-theoretic Verification

Input: P_{new} (new policy), $P_{existing}$ (list of existing policies)
Output: C (conflict report set)
 # Define comparable fields by policy type
1: $F \leftarrow$ GetComparableFields(P_{new}.type)
 # Iterate over existing policies (type must match)
2: **for** $P_i \in P_{existing}$ **do**
3: **if** P_i.type $\neq P_{new}$.type **then**
4: **continue**
5: **end if**
 # Selector overlap check and rule extraction
6: $S_{new} \leftarrow$ ExtractSelector(P_{new})
7: $S_i \leftarrow$ ExtractSelector(P_i)
8: **if** $S_{new} \cap S_i = \emptyset$ **then**
9: **continue**
10: **end if**
11: $S_{overlap} \leftarrow S_{new} \cap S_i$
12: $R_{new} \leftarrow \{r \in P_{new}.\text{rules} \mid r.\text{selector} \in S_{overlap}\}$
13: $R_i \leftarrow \{r \in P_i.\text{rules} \mid r.\text{selector} \in S_{overlap}\}$
 # Policy-level redundancy check (exact equality for all fields)
14: *has_redundancy* $\leftarrow$ false
15: **if** $\forall r \in R_{new},\ \exists r' \in R_i :\ \bigwedge_{f \in F} r[f] = r'[f]$ **then**
16: AddConflict(C, Redundancy, P_{new}, P_i, $S_{overlap}$)
17: *has_redundancy* $\leftarrow$ true
18: **end if**
 # Rule-level explicit conflict check (same conditions, different actions)
19: *has_explicit* $\leftarrow$ false
20: **for** $r \in R_{new}$ **do**
21: **for** $r' \in R_i$ **do**
22: **if** $\left(\bigwedge_{f \in F \setminus \{action\}} r[f] = r'[f]\right) \wedge (r.action \neq r'.action)$ **then**
23: AddConflict(C, Explicit, P_{new}, P_i, (r, r'))
24: *has_explicit* $\leftarrow$ true
25: **end if**
26: **end for**
27: **end for**
 # Correlation conflict detection (partial overlap)
28: $R_{match} \leftarrow \emptyset$
29: **for** $r \in R_{new}$ **do**
30: **for** $r' \in R_i$ **do**
31: *matched* $\leftarrow 0$
32: **for** $f \in F$ **do**
33: **if** $(r[f] \cap r'[f] \neq \emptyset) \wedge (r[f] \neq r'[f])$ **then**
34: *matched* $\leftarrow$ *matched* $+ 1$
35: **end if**
36: **end for**
37: **if** $0 <$ *matched* $< |F|$ **then**
38: $R_{match} \leftarrow R_{match} \cup \{(r, r')\}$
39: **end if**
40: **end for**
41: **end for**
42: **if** $R_{match} \neq \emptyset \wedge \neg$*has_redundancy* $\wedge \neg$*has_explicit* **then**
43: AddConflict(C, Correlation, P_{new}, P_i, R_{match})
44: **end if**
45: **end for**
46: **return** C

Explicit conflict detection operates at the rule level, identifying contradictory action specifications under identical matching conditions by examining individual rule pairs. For this, it initializes a flag to track explicit conflicts (line 19) and iterates over all rule pairs between the new and existing rule sets through nested loops (lines 20–21). For each pair (r, r'), it checks whether all non-action fields are equal while actions differ (line 22). This captures scenarios where identical conditions yield opposing enforcement decisions, such as one rule allowing TCP on port 443 to a specific target while another rule denies the same traffic pattern. When such an explicit conflict is detected, the algorithm records the specific conflicting rule pair (line 23) and sets the explicit conflict flag (line 24). Correlation conflict detection operates at the rule level to identify partial overlaps that create ambiguous enforcement boundaries.

The algorithm initializes an empty set to collect partially overlapping rule pairs (line 28) and iterates over all combinations of rules from both rule sets through nested loops (lines 29–30). For each rule pair, a counter tracks the number of fields exhibiting partial overlap (line 31). The algorithm examines each field in the comparable field set (line 32), incrementing the counter when the field satisfies two conditions: non-empty intersection and non-equality (lines 33–34). This dual-condition mechanism correctly classifies three representative correlation patterns: single values or narrow ranges within broader ranges (port 8080 vs 8000–9000), value sets partially overlapping with ranges (ports {80,85,90} vs 80–90), and sub-ranges within larger ranges (ports 85–88 vs 80–90). After examining all fields for a rule pair, if the matched count falls strictly between zero and the total field count, indicating some but not all fields correlate, the rule pair is added to the match set (lines 37–38). After examining all rule pairs, if the match set is non-empty and neither redundancy nor explicit conflicts were detected for this policy pair, the algorithm records a correlation conflict (lines 42–43). The exclusion conditions prevent redundant conflict reporting when policies are already classified under more specific conflict types.

Finally, the algorithm returns the complete conflict report containing all detected conflicts across policy pairs (line 46). This structured report provides administrators with actionable insights for policy refinement or precedence specification, categorizing conflicts by type, identifying the specific policies and rules involved along with their provider information, and indicating the scope of resource overlap. Administrators can interpret detected conflicts in light of each provider's enforcement semantics (Section 3.4) to determine the appropriate course of action. By detecting conflicts proactively during the policy generation phase rather than after deployment, the mechanism supports preventive security management that maintains consistency across heterogeneous policy types within the cluster.

6. Evaluation

6.1. Implementation

We have implemented a prototype of KubeTeus+ using a combination of Go and Python to verify its feasibility and effectiveness. Currently supported policies include Kubernetes Network Policy (The Kubernetes Authors, 2025b), Cilium Network Policy (Cilium Authors, 2025b), and Calico Network Policy (Tigera, 2025a), covering both Layer 3–4 and Layer 7 and KubeArmor Policy (The KubeArmor Authors, 2025b) for system-level container runtime security. To enhance the initial intent prompt, we developed an entity classification model based on BERT (Devlin et al., 2018) and RoBERTa (Liu et al., 2019). In addition, the five LLMs shown in Table 5, which were used to generate the security policies, were fine-tuned in advance. In summary, to support the design features described in Section 4.1, we implemented a fine-tuning automator, a prompt processor, a policy processor, a policy validator, and a policy enforcer, in approximately 5300 lines of code.

6.2. Evaluation environments

We evaluated KubeTeus+ in both CPU and GPU environments to assess its performance across different computational settings. The GPU environment utilized a high-performance server equipped with an NVIDIA Hopper H100 80 GB PCIe, which was employed both for fine-tuning the LLMs and for evaluating the framework's policy generation performance. The CPU environment consisted of an AMD EPYC 9224 CPU with 256 GB of RAM and a 2TB SSD, which was used to deploy the microservice and evaluate the system's operation under realistic deployment conditions. The Kubernetes cluster consisted of one master node and two worker nodes, each hosted on three Ubuntu 24.04 virtual

Table 6
The summary of cloud-native test application demos used for evaluation.

Application		MSA	#Service	Domain
DeathStarBench (Gan et al., 2019)	Social Network	✓	27	Social Media Platform
	Hotel Reservation	✓	19	Hotel Booking System
Sock Shop (OCP Power Demos, 2025)		✓	14	Online Retail
Online Boutique (Google Cloud, 2025c)		✓	11	E-commerce Platform
Martian Bank (Cisco, 2025)		✓	6	Financial Services
Bookinfo (The Istio Authors, 2025)		✓	4	Book Catalog

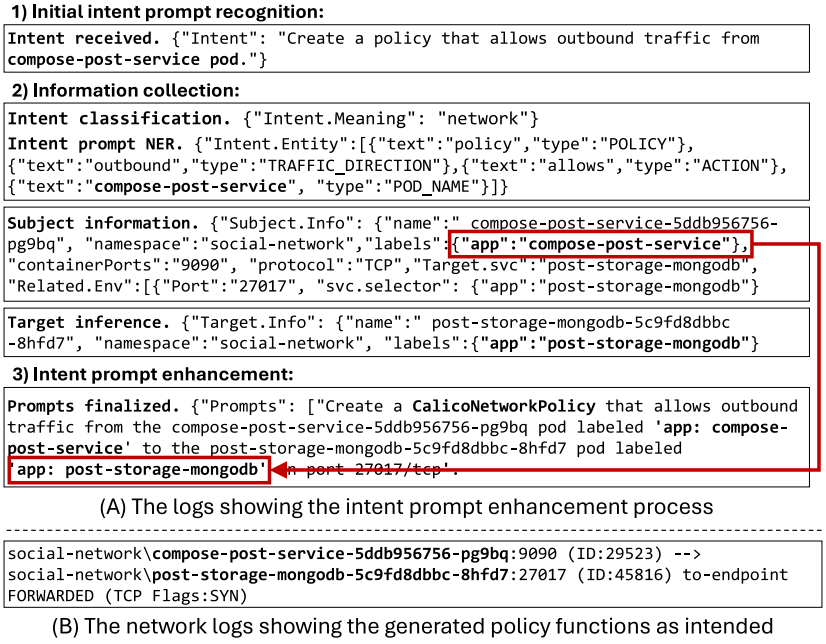


Fig. 7. The results of enhanced prompting (A) and network log from correct policy enforcement (B).

machines. For the container-related configuration, we used containerd (The containerd Authors, 2025) as the container runtime and selected Cilium (Cilium Authors, 2025a) and Calico (Tigera, 2025b) as the target CNIs to provide inter-container communication, along with KubeArmor (The KubeArmor Authors, 2025a) as the runtime security engine for system policy enforcement. We evaluated the generation of four types of security policies: Kubernetes Network Policy, Cilium Network Policy, Calico Network Policy, and KubeArmor Policy. KubeTeus+ was deployed on the master node, while the test applications were distributed across the two worker nodes.

We tested six cloud-native applications with different scales, functionalities and architectural complexities, as summarized in Table 6. These applications were selected as they exemplify diverse architectural patterns in production environments, ranging from small-scale demonstrations with 4 services to complex systems with 27 services. This microservice demo enabled comprehensive evaluation of security policy generation, validation, and enforcement processes under conditions similar to real-world microservice deployment scenarios.

6.3. 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 Social Network from DeathStarBench (Gan et al., 2019), 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.

Fig. 7 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 Fig. 7 (A)-2, KubeTeus+ identifies that the `app: compose-post-service` pod is related to the `app: post-storage-mongodb` pod through service endpoint analysis. Consequently, the enhanced prompt includes important details not specified in the initial prompt, such as the specification of the `CalicoNetworkPolicy` type, accurate pod labels, specific traffic flow directions, and the target port (27017) for MongoDB communication. The accuracy of this enhanced prompt is verified through actual network logs shown in Fig. 7 (B). The traffic patterns recorded in the logs precisely match the pod relationships specified in the enhanced prompt, confirming successful TCP connection establishment from the compose-post-service pod to the post-storage-mongodb pod. This demonstrates that KubeTeus+'s prompt engineering capability effectively generates accurate and detailed network policies in complex microservice environments.

Validation of the Policy Itself. This evaluation assesses the ability of our system to validate the CRD, resource, and property aspects of security policies, using system policies as the evaluation target. Fig. 8 illustrates the test scenarios in the Online Boutique Demo (Google Cloud, 2025c) and results of the validation process. The top of Fig. 8 presents information about the 'paymentservice' pod in the 'onlineboutique' namespace, which uses the specific container image. However, vulnerability scanning reveals that this image contains the c-ares library

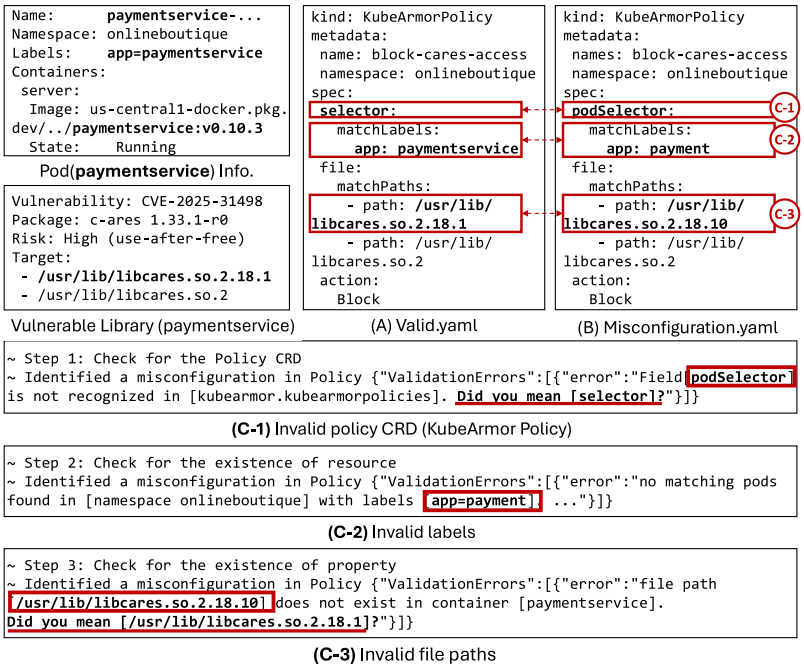


Fig. 8. The results of policy validation scenarios (cve-2025-31498 National Institute of Standards and Technology, 2025): valid (A), misconfigured (B), and the detection logs (C).

version 1.33.1-r0, which is affected by CVE-2025-31498 (National Institute of Standards and Technology, 2025), a high-risk use-after-free vulnerability. This vulnerability allows remote attackers to manipulate DNS response handling, potentially leading to memory corruption and remote code execution. To mitigate this threat, we created a system policy designed to block access to the vulnerable library files, and we now evaluate whether our system can properly validate this policy before enforcement.

Fig. 8 (A) shows a valid KubeArmor policy that correctly uses the `selector` field with proper labels (`app: paymentservice`) and accurately specifies the vulnerable library paths, while (B) depicts an incorrectly configured policy with multiple errors: an unsupported field, incorrect pod labels, and a mistyped file path. When attempting to enforce policy (B), our system successfully detects these misconfigurations through a multi-step validation process. During the CRD validation stage, the system identifies the use of the unsupported field `podSelector` instead of the correct `selector` field for KubeArmor policies (C-1). In the resource validation stage, the system detects that the misspelled label value `app: payment` does not match any existing pods in the cluster, where the correct label should be `app: paymentservice` (C-2). Finally, in the property validation stage, the system queries the actual container filesystem and identifies that the specified file path `libcares.so.2.18.10` does not exist in the '*/usr/lib/*' directory, suggesting the correct filename `libcares.so.2.18.1` based on the container's actual file structure (C-3). Errors identified at each validation stage are immediately reported to the administrator via logs, along with suggested corrections, enabling the rapid identification and resolution of issues. These results demonstrate that our system effectively prevents potential security misconfigurations before policy enforcement through multi-layered validation, ensuring accurate runtime security policy management.

Validation of Conflicts between Policies. To evaluate the policy conflict detection of KubeTeus+, we conducted tests using Calico Network Policies in the Sock Shop (OCP Power Demos, 2025). Fig. 9 illustrates three types of policy conflicts that can occur when attempting to enforce new policies alongside existing ones in the cluster. Fig. 9 (A) shows an existing network policy already deployed in the cluster, which allows ingress traffic from the 'front-end' pod to the 'carts' service on port range 8000–8100 using TCP. Based on this existing policy, we developed three new policies to illustrate each type of conflict. The first conflict type is redundancy, which occurs when a new policy with a different name (`allow-ingress-8000-8100`) but identical specifications to the existing policy (A) is deployed. As shown in the detection result (D-1), our system detects that all rules in the new policy are fully covered by the existing policy `ingress-8000-8100`, making it redundant. The detection result confirms complete overlap across all comparable fields, including selector, target, action, protocol, and port range.

Fig. 9 (B) demonstrates an explicit conflict through the opposite action policy, which specifies a `Deny` action for the same traffic flow that the existing policy explicitly allows. The system identifies this contradiction in enforcement actions for identical matching conditions, as shown in the detection result (D-2), where the same selector, source, and port range specifications yield opposite actions. Fig. 9 (C) shows a correlation conflict through the subset port policy, which specifies a single port (8080) that falls within the existing policy's broader port range (8000–8100). This creates a partial overlap where the new policy targets a subset of traffic already governed by the existing policy. The detection result (D-3) identifies this port-based correlation, highlighting that port 8080 falls within the existing policy's port range, creating potential ambiguity in enforcement precedence. For each conflict scenario, our system provides detailed conflict reports specifying the conflict type, overlapping scope, and reasoning, enabling administrators to make informed decisions about policy modifications or precedence settings before deployment. These results demonstrate that our proactive conflict detection mechanism effectively maintains policy consistency and prevents contradictory enforcement behaviors in complex microservice environments.

6.4. Performance

Classification Model Performance. In this evaluation, we tested the performance of both the intent classifier and entity classifier by assessing their accuracy in correctly identifying policy types and extracting key entities from intent prompts. As shown in Table 4, the test dataset was constructed by randomly sampling approximately 20% of

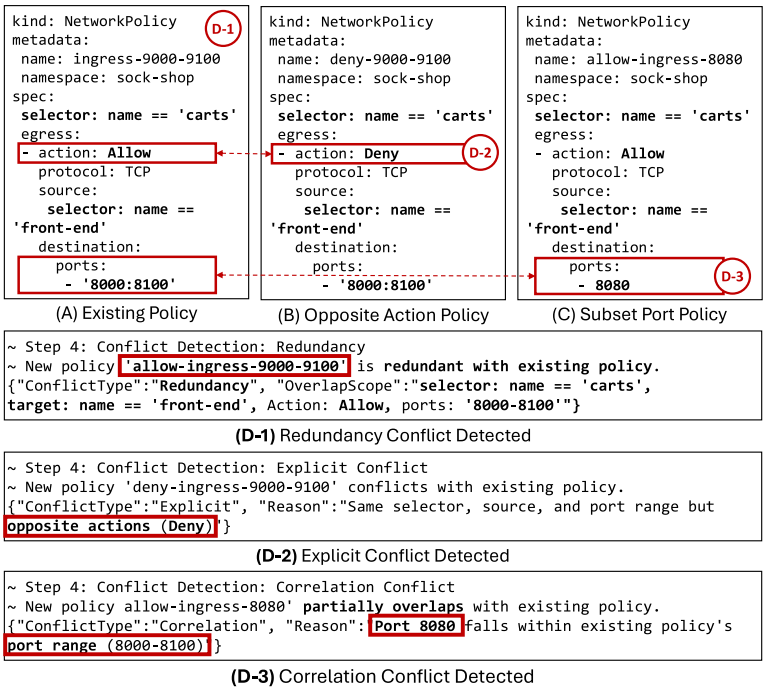


Fig. 9. The results of policy conflict detection scenarios: existing policy (A), opposite action policy (B), subset port policy (C), and detection logs (D).

Table 7
The summary of intent and entity classifier performance with metrics: Accuracy, Precision, Recall, and F1 Score.

Task	Model	Acc.	Prec.	Rec.	F1.
Intent	BERT	0.982	0.985	0.973	0.980
	RoBERTa	0.973	0.977	0.969	0.973
Entity	BERT	0.972	0.975	0.972	0.974
	RoBERTa	0.961	0.969	0.962	0.965

intent samples from each of the network and system policy datasets and mixing them. The test dataset for the intent classifier consisted of intent prompt texts paired with their corresponding policy type labels, while the entity classifier test set included intent prompts with their associated entity tag sequences. Using these test sets, we compared the performance of models based on BERT and RoBERTa across both classification tasks. During the evaluation process, the intent classifier predicted policy types for the given intent prompts, while the entity classifier predicted entity tags. These predictions from both models were compared against the actual labels (ground truth) included in the test set.

As summarized in Table 7, the BERT-based model achieved 98.2% accuracy, 98.5% precision, 97.3% recall, and a 98.0% F1 score in distinguishing between network and system policy intents for the intent classification task. For the entity classification task, the BERT-based model achieved 97.2% accuracy, 97.5% precision, 97.2% recall, and a 97.4% F1 score across 16 entity classes spanning common, network-specific, and system-specific categories. The RoBERTa model showed similarly high performance across both tasks, albeit slightly lower than BERT. Both models in each task demonstrated F1 scores exceeding 97%, indicating a balance between precision and recall.

The remaining approximately 3% of misclassification cases occur across both tasks. For intent classification, misclassifications primarily arise when prompts contain terminology strongly associated with another policy domain. For example, a system policy intent such as "deny execution of wget" was occasionally misclassified as a network policy, as the network-related tool name biased the classifier toward the network domain despite the intent targeting process-level restrictions. For entity classification, errors typically involve minor formatting variations that deviate from common conventions, such as compound label values containing hyphens (e.g., "app=frontend-v2") being subject to incorrect token boundary detection, or abbreviated namespace references not matching their full canonical forms. However, KubeTeus+ addresses these edge cases through automated rule-based pattern analysis to complement the classification models' predictions.

These results demonstrate that KubeTeus+ can accurately identify policy types and extract crucial entities across diverse security policy contexts, including pod names, labels, traffic directions, port numbers for network policies, and file paths, process names, capabilities for system policies.

Fine-tuned LLM Performance. To evaluate the accuracy of security policies generated based on enhanced intent prompts, we compared the performance of base models and fine-tuned models using BLEU (Papineni et al., 2002), METEOR (Banerjee and Lavie, 2005), ROUGE (Lin, 2004) (ROUGE-1, ROUGE-2, ROUGE-L), and chrF++ (Popović, 2017) metrics. Given that even minor syntactic errors in security 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 to measure how accurately the generated policy structure and token sequences match the reference policies, METEOR to evaluate whether the policy rules preserve the intended semantics such as correctly mapping allow/deny actions and accurately reflecting traffic directions or resource access patterns, ROUGE metrics to assess the completeness of essential policy components such as selectors, rules, and targets without omissions, and chrF++ to verify character-level precision in YAML syntax including proper spelling and formatting of field names to prevent typographical errors that could cause policy rejection.

For network policy generation, as shown in Fig. 10, 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 ± 0.03 to 0.76 ± 0.02) and its ROUGE-1 score improving by 23% (from 0.71 to 0.87 ± 0.01).

The codegemma-7b model showed similar enhancements. These results indicate that fine-tuning has effectively specialized the models for

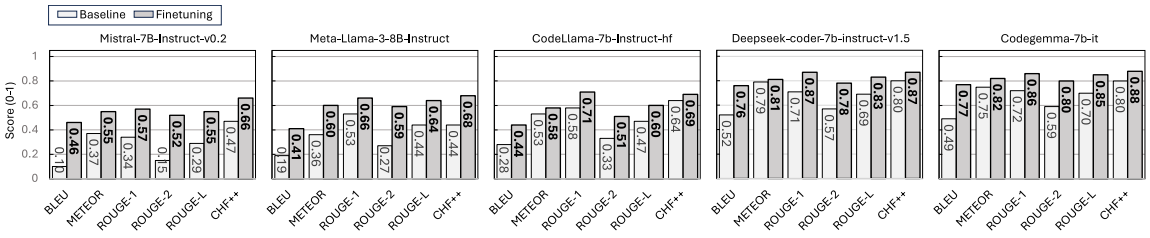


Fig. 10. The summary of the fine-tuned LLMs performance for network policy generation with metrics: BLEU, METEOR, ROUGE-1, ROUGE-2, ROUGE-L, and chrF++.

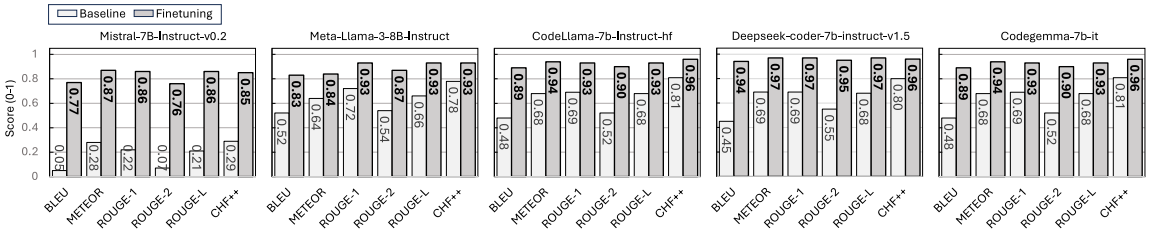


Fig. 11. The summary of the fine-tuned LLMs performance for system policy generation with metrics: BLEU, METEOR, ROUGE-1, ROUGE-2, ROUGE-L, and chrF++.

network policy generation tasks. On average, we observed performance improvements of approximately 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 across diverse CNI environments. For system policy generation, as shown in Fig. 11, the models demonstrated even more pronounced improvements compared to network policies. The deepseek-coder-7b model achieved exceptional performance with BLEU and METEOR scores increasing by 109% (from 0.45 to 0.94) and 41% (from 0.69 to 0.97), respectively. The codegemma-7b model similarly achieved substantial improvements with BLEU increasing by 85% (from 0.48 to 0.89 ± 0.02). Notably, the Mistral-7B model, which showed the weakest baseline performance, demonstrated the most dramatic improvement, with its BLEU score increasing by 1,440% (from 0.05 ± 0.02 to 0.77 ± 0.03) after fine-tuning.

These results indicate that fine-tuning has effectively specialized the models for system policy generation tasks. On average, we observed performance improvements of approximately 150% in BLEU scores, 40% in METEOR scores, and 35% in ROUGE-1 scores. Notably, these improvements are observed consistently across all four metrics rather than concentrated in a single dimension; the parallel gains in precision-oriented (BLEU, chrF++), recall-oriented (ROUGE), and semantics-aware (METEOR) metrics indicate that fine-tuning improves the overall policy quality rather than superficially inflating lexical overlap while leaving structural or semantic errors undetected.

In conclusion, this multi-dimensional consistency provides stronger evidence of generation quality than any single metric in isolation. The simultaneous gains in BLEU and chrF++ confirm that fine-tuned models produce outputs with both correct token sequences and precise character-level syntax, while METEOR improvements indicate preserved rule semantics and ROUGE gains verify that critical policy sections are not omitted. A policy achieving high scores across all four perspectives is unlikely to contain field-level errors that would render it semantically incorrect despite superficial lexical similarity. These results demonstrate the potential of our system as a reliable security policy generation tool across diverse CNI and runtime security environments. The consistently high performance across both network and system policies validates the generalizability of our fine-tuning approach to heterogeneous policy domains.

Ablation Study. To evaluate the contribution of each component underlying KubeTeus+'s cognition-driven architecture, we conducted an ablation study by systematically disabling fine-tuning (FT) and prompt enhancement (PE) across four conditions: both enabled (A), FT disabled (B), PE disabled (C), and both disabled (D). FT refers to the domain-specialized model learning (Section 5.1), and PE refers to the entity classification, context-aware prompt enhancement, and provider keyword normalization stage (Section 5.2). We evaluate two representative models selected from the five fine-tuned models to contrast distinct generalization profiles: CodeGemma-7b-it, a code-specialized model, and Mistral-7B-Instruct-v0.2, a general-purpose instruction model, across both network and system policy domains. In conditions C and D, PE is fully deactivated; the test prompts used for these conditions are derived from the same test set by removing the enriched elements that PE supplies, reducing them to the minimal specification level (Level 1) defined in Table 3, where policy type, subject, and target labels, port, and protocol are absent. Table 8 quantifies the performance degradation under each removal condition, and Table 9 captures whether that degradation extends to structural deployability through CRD validation pass rates across three criteria: spec structure presence (S1), selector field correctness (S2), and rule section completeness (S3).

As shown in Table 8, FT removal (B) already produces considerable decline: CodeGemma system policy BLEU decreases by approximately 50% (0.85 ± 0.03 to 0.42 ± 0.04) and Mistral by 42% (0.78 ± 0.03 to 0.45 ± 0.04), demonstrating that PE-based prompt enrichment remains indispensable even when the model has undergone domain-specialized fine-tuning. PE removal (C) produces a yet more severe collapse, with CodeGemma system policy BLEU falling by 79% (0.85 to 0.18) and Mistral by 78% (0.78 to 0.17), compared to the 50% and 42% drops observed under condition B. This larger margin indicates that specification completeness, which PE supplies, is a stronger determinant of policy correctness than model specialization alone: even a well fine-tuned model cannot reliably populate critical policy fields when the intent is reduced to the minimal level (Section 3). This asymmetry between B and C is particularly informative for system policies: the relatively larger drop under B compared to network policies (0.85 to 0.42 vs. 0.76 to 0.64) suggests that system policies depend more heavily on fine-tuning to capture provider-specific schema structures, as their target format is less represented in general pretraining corpora. For network policies, the gap between B and A is more moderate (0.76 to 0.64), reflecting that code-specialized pretraining provides a stronger foundation for the relatively more common YAML-based network policy syntax, yet condition C still causes substantial degradation, with

Table 8
Ablation study on the contribution of fine-tuning (FT) and prompt enhancement (PE) to intent-policy translation accuracy (✓: enabled, ✗: disabled, MET: METEOR, R-1/R-2/R-L: ROUGE-1/ROUGE-2/ROUGE-L).

				(a) Network Policy						(b) System Policy					
Model	Condition	FT	PE	BLEU	MET	R-1	R-2	R-L	chrF++	BLEU	MET	R-1	R-2	R-L	chrF++
CodeGemma	A (Full)	✓	✓	**0.76**	**0.81**	**0.85**	**0.80**	**0.84**	**0.87**	**0.85**	**0.89**	**0.89**	**0.85**	**0.89**	**0.91**
-7b-it	B (w/o FT)	✗	✓	0.64	0.70	0.82	0.72	0.79	0.82	0.42	0.66	0.68	0.46	0.65	0.72
	C (w/o PE)	✓	✗	0.49	0.62	0.66	0.50	0.63	0.69	0.18	0.22	0.24	0.17	0.23	0.24
	D (w/o Both)	✗	✗	0.14	0.29	0.35	0.15	0.32	0.40	0.06	0.18	0.16	0.07	0.16	0.23
Mistral-7B	A (Full)	✓	✓	**0.44**	**0.50**	**0.51**	**0.40**	**0.49**	**0.67**	**0.78**	**0.87**	**0.87**	**0.77**	**0.87**	**0.86**
-Instruct-v0.2	B (w/o FT)	✗	✓	0.23	0.45	0.55	0.25	0.44	0.57	0.45	0.60	0.64	0.50	0.61	0.69
	C (w/o PE)	✓	✗	0.31	0.41	0.43	0.31	0.43	0.58	0.17	0.24	0.23	0.16	0.22	0.28
	D (w/o Both)	✗	✗	0.05	0.13	0.15	0.06	0.14	0.20	0.02	0.10	0.07	0.02	0.06	0.12

Table 9
CRD schema structural validation pass rates across ablation conditions (S1: spec structure presence, S2: selector field correctness, S3: rule section completeness).

Model	Condition	Network policy			System policy		
		S1	S2	S3	S1	S2	S3
CodeGemma	A (Full)	0.98	0.97	0.98	0.92	0.93	0.93
-7b-it	B (w/o FT)	0.95	0.95	0.95	0.94	0.85	0.87
	C (w/o PE)	0.93	0.82	0.93	0.29	0.26	0.20
	D (w/o Both)	0.62	0.63	0.63	0.25	0.15	0.18
Mistral-7B	A (Full)	0.92	0.92	0.90	0.94	0.94	0.89
-Instruct-v0.2	B (w/o FT)	0.90	0.77	0.88	0.71	0.67	0.71
	C (w/o PE)	0.83	0.82	0.82	0.40	0.33	0.27
	D (w/o Both)	0.30	0.27	0.27	0.29	0.04	0.16

CodeGemma BLEU dropping by 36% (0.76 to 0.49), confirming that PE's role in label disambiguation and provider-specific field resolution cannot be substituted by pretraining familiarity alone. When both components are removed (D), scores fall to BLEU 0.02–0.14, confirming that neither component alone is sufficient.

Table 9 reveals that the degradation under each removal condition manifests as a distinct class of structural failure. Under condition C, system policy pass rates collapse substantially, with CodeGemma S1 dropping by 68% (0.92 to 0.29) and Mistral S1 by 57% (0.94 to 0.40), indicating that without PE-supplied context, models generate structurally invalid configurations that cannot be applied to a real cluster, not merely inaccurate ones. Notably, S2 shows the sharpest drop within this condition, consistent with the absence of label and resource information that PE normally supplies and that selectors directly depend on. Under condition B, structural pass rates decline more moderately, and the drop is more pronounced in S2 and S3 than in S1, suggesting that FT removal degrades field-level and rule-level precision while basic schema structure remains partially intact from pretraining. Network policies are more resilient across all criteria under condition C, with CodeGemma S1 declining by only 5% (0.98 to 0.93), again reflecting their greater structural accessibility from pretraining, but condition B reveals that provider-specific selector alignment still requires fine-tuning to be reliably correct.

These results demonstrate that FT and PE are both necessary and serve complementary roles: fine-tuning alone falls short when the intent lacks the contextual grounding that PE provides, and PE alone cannot achieve the structural field precision that fine-tuning establishes. Together, they operationalize the cognition-driven design stated in the framework's core premise: the ability to understand security requirements through domain-specific entity recognition and perceive real-time cluster states, as realized by PE, must be coupled with the capacity to generate structurally precise and provider-aligned policies, as established by FT. The ablation results empirically validate that neither cognitive function alone is sufficient to realize accurate and deployable policy generation, justifying the integrated architecture of KubeTeus+.

Processing Time Breakdown To evaluate runtime performance, we measured the end-to-end processing time from the moment the user intent is entered until the validated policy is deployed. Although we implemented both local serving and the Hugging Face pipeline-based large language model (LLM) inference approach (Hugging Face, 2025b), for this evaluation, we used the local serving approach to measure pure processing time, which is unaffected by network overhead. For each application listed in Table 6, we created multiple test cases using minimal user prompts that require comprehensive prompt enhancement by KubeTeus+, covering diverse scenarios including service-to-service communication patterns for network policies and container runtime security requirements for system policies. This resulted in approximately 120 distinct policy generation test cases across all applications. To ensure statistical reliability, we conducted 10 independent runs for each test case on both hardware configurations and calculated the average processing time.

Fig. 12 presents the end-to-end processing times across different hardware configurations. GPU environments achieve a total processing time of approximately 28.8 s, while CPU-only setups require around 69 s, demonstrating a 2.4× performance gap. This disparity originates entirely from model inference stages: classification (intent and entity recognition using BERT-based models) takes 5.12 s on GPU versus 14.17 s on CPU, while LLM inference requires 22.18 s and 53.25 s respectively. In contrast, prompt engineering and validation stages maintain identical absolute execution times (< 1 s each) across both configurations. Consequently, classification and LLM inference together account for over 95% of KubeTeus+'s total processing time, demonstrating that computational resources significantly impact model-dependent stages and suggesting that processing time can vary substantially depending on GPU availability.

Resource Overhead. Beyond processing time, we also assessed resource consumption during policy generation to evaluate deployment feasibility. During LLM inference, the fine-tuned 7B-parameter models require approximately 14–17 GB of GPU VRAM under FP16 precision, which is well within the capacity of modern datacenter GPUs. Peak GPU utilization reaches approximately 95%–100% during inference but is sustained only for the duration of each policy generation request (approximately 22 s on GPU, as shown in Fig. 12), after which resources are released. CPU and memory overhead for the remaining stages—including NER-based classification, prompt engineering, Kubernetes

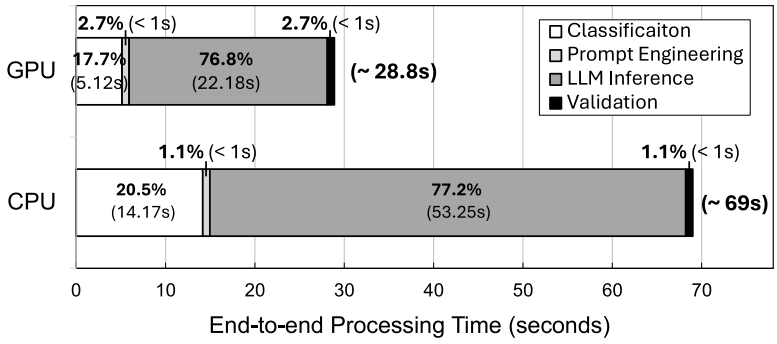


Fig. 12. The breakdown of end-to-end processing time across GPU and CPU environments: classification (intent and entity recognition), prompt engineering, LLM inference, and four-step validation.

API queries, and multi-step validation—remains minimal, collectively consuming under 2 GB of RAM.

However, as a comprehensive end-to-end policy generation pipeline, KubeTeus+ appears to provide a performance-security trade-off worth considering for cloud providers managing Kubernetes services with automated policy generation, platform engineers managing complex microservice deployments, and DevOps teams seeking to eliminate manual configuration errors and reduce administrative overhead.

7. Related work

Intent-Based Management. Due to the increasing complexity of network management, research on intent-based approaches has been actively pursued (Jacobs et al., 2021; Collet et al., 2022). Jacobs et al. (2021) proposed a system that translates natural language intents into an intermediate language form, which is then converted into network configurations, using machine learning and operator feedback to verify alignment with operator goals. Collet et al. (2022) proposed an approach to automatically learn and deploy predictive models tailored to network management objectives. Research utilizing LLMs has also gained attention (Dzeparoska et al., 2023; Van Tu et al., 2024). Dzeparoska et al. (2023) employed few-shot learning with LLMs to incrementally decompose intents and automate application management through policy-based abstraction. Van Tu et al. (2024) applied in-context learning with LLMs to intent-based configuration in NFV environments, enabling intent translation without retraining the LLM through JSON templates. Han et al. (2025) proposed NLI2Conf, an intent-driven network configuration updating model that integrates graph and language models to handle forwarding strategies and performance constraints, though it focuses on network configuration rather than security policy generation. However, these studies are not well-suited to the dynamic nature of container-based microservices and require considerable user effort in formulating intents, lacking comprehensive automation for both network and system policy domains.

Advanced Security Policy Generation. Several studies have explored automated policy generation in containerized environments, primarily focusing on either network or system policies in isolation (Li et al., 2021; Xu et al., 2023; Lee and Nam, 2023). For network policies, Li et al. (2021) developed a system that automatically generates access control policies by analyzing interactions between microservices using a static analysis-based request extraction. Xu et al. (2023) 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. Lee and Nam (2023) introduced an automated network policy discovery framework that generates a minimal set of network security policies using network logs. Kim et al. (2025b) proposed an intent-driven network policy generation framework that leverages LLMs and domain-specific NER models to automate policy creation from natural language intents. Ghavamnia et al. (2020) uses static code analysis to generate Seccomp system call policies for Docker containers by identifying required system calls from containerized applications and their dependencies. Yang et al. (2024) propose association-based dynamic system call filtering that validates and enforces minimal system calls for containers at runtime using eBPF-based monitoring. Recently, Kim and Lee (2024) proposed a unified framework integrating network, system, and cluster-level policies through adapter-based approaches. While this framework provides centralized management, it introduces additional complexity by requiring administrators to learn yet another policy schema. Moreover, open-source tools have been developed for policy management. Otterize Intents Operator (Otterize, 2025) declaratively defines intended access policies and converts them into network and Kafka policies. Open Policy Agent (OPA The Open Policy Agent Authors, 2025) enables policy-based access control across various environments. GateKeeper (The Gatekeeper Authors, 2025) utilizes OPA to validate requests in Kubernetes clusters. However, these works primarily focus on past data to generate security policies, making it challenging to reflect nuanced policy intentions and remain confined to single policy domains or introduce additional complexity through new schemas.

Policy Validation. Research on policy verification has also been proposed to ensure correctness before enforcement (Y. Li et al., 2020; Li et al., 2022; Kang and Shin, 2022). Y. Li et al. (2020) and Li et al. (2022) 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 and Shin (2022) introduced a method for automatically verifying container network policies using a novel graph structure that represents policies. For conflict detection, Zahoor et al. (2023) proposed a formal approach using Event-Calculus to identify redundancy in Kubernetes authorization policies for RBAC and ABAC modes. Togay et al. (2021) introduced a firewall policy anomaly detection framework that employs logic programming to detect intra-firewall anomalies such as shadowing, redundancy, generalization, and correlation within traditional packet-filtering firewalls through static rule-set analysis. However, these systems either verify policies only after deployment or focus on structural validation without conflict detection, and remain limited to specific policy types or environments, lacking comprehensive pre-deployment analysis that integrates policy correctness verification and inter-policy conflict detection across diverse security policy domains.

Unlike previous works, KubeTeus+ is an LLM-based security 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 both network and system policy. This automated approach minimizes user intervention, reduces human error, and enhances efficiency. By combining multi-step validation encompassing CRD, resource, and property verification with set-theoretic conflict

detection, our system proactively prevents potential misconfigurations and inter-policy conflicts before deployment, ensuring reliable policy enforcement in dynamic microservice architectures. By automating LLM improvements, KubeTeus+ adapts to evolving cloud environments, enabling efficient security policy management, particularly in generating container-aware policies through real-time resource relationship inference.

8. Limitation and discussion

Extensibility to New Policy Providers. For integrating a new CNI or runtime security tool into KubeTeus+, the structural analysis in Section 3 confirms that existing security policy providers share a high degree of common structural patterns across both network and system policies, meaning that field mapping, keyword normalization, and prompt-level adaptation require virtually no additional effort. The primary integration cost therefore lies in securing an LLM capable of generating policies in the target provider's syntax and constructing a fine-tuning dataset for it. Once a provider-specific dataset of intent–policy pairs is prepared, the automated fine-tuning pipeline (Section 5.1) automates the process from hyperparameter search to model training, concentrating the manual effort solely on dataset preparation. In cases where such training data is scarce, leveraging large-scale commercial LLMs that already possess knowledge of the target provider's policy schema through their API offers a practical alternative, enabling integration without dedicated fine-tuning.

Robustness to Adversarial and Malformed Intents. KubeTeus+ directly translates natural-language intents into enforceable security policies, making the integrity of intent inputs a prerequisite for trustworthy policy generation. The current system operates under the assumption that intent prompts originate from authorized administrators within a controlled environment. Even when adversarial or malformed intents are provided, our multi-step validation pipeline detects structural errors at the policy level, inconsistencies with the live cluster state, and logical conflicts with existing policies, thereby preemptively blocking enforcement. This validation applies equally to policies generated by the LLM and those authored manually by administrators, serving as a pre-deployment verification tool. However, the possibility that a generated policy passes all validation steps yet exhibits a subtle discrepancy between the security scope originally intended and its actual enforcement scope cannot be entirely ruled out. Systematically defining what security risks such discrepancies can lead to in cloud-native environments and analyzing their impact represents an important research challenge. Building on this foundation, applying the concept of intent drift, as discussed in the intent-based networking domain, to the security policy generation pipeline and developing techniques that continuously verify the alignment between intents and policies in dynamically changing container environments constitutes a direction for future work.

Cross-Domain Policy Conflict Detection. Our current conflict detection mechanism identifies conflicts within a single policy domain but does not address cross-domain conflicts between network and system policies. In practice, policies from different security domains can create inconsistencies that weaken the overall security posture. For example, a network policy may allow TCP traffic on port 8080 to a pod, while a system policy blocks the process listening on that port, negating the network policy. Similarly, a system policy may permit access to sensitive configuration files, but a network policy could block communication to the service that needs those files, causing operational issues. These cross-domain conflicts cannot be detected by reviewing policies in isolation. Addressing them would require a unified semantic model that maps policy effects across network and system layers, enabling comprehensive conflict analysis across multiple domains.

Priority-Based Policy Enforcement. Although we have identified architectural differences in priority models across policy providers, as discussed in Section 3, the current system does not support priority-based enforcement when applying policies. This limitation is especially significant when policies from multiple providers coexist in the same cluster, particularly in multi-CNI environments, such as those using Multus (Multus Authors, 2025). While these environments allow for more flexible and tailored policy configurations, they also increase complexity when multiple policies target the same resources with conflicting enforcement actions. Without a unified precedence framework to coordinate provider-specific priority models, the system cannot determine which policy should prevail. Implementing priority-based enforcement requires identifying related policies and integrating business requirements and operational factors that determine precedence. For example, security-critical policies may need to override application-specific policies, or emergency response policies may need to take precedence over normal operational rules. Our current conflict detection mechanism can identify policy conflicts between different CNI providers targeting the same labels, but it does not fully coordinate enforcement based on these findings. As a result, administrators must manually analyze conflict reports and determine enforcement order according to their organization's operational priorities. Extending our framework to include a unified precedence model that considers both provider-specific priorities and cross-domain policy relationships would address this gap. It would also enhance conflict resolution, enable automated policy precedence decisions across diverse security domains, and improve the reliability and consistency of policy enforcement.

9. Conclusion

This paper presents an automated software framework, KubeTeus+, for generating security policies in 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. Our system supports diverse security policy providers including Container Network Interfaces (CNIs) for network policies and runtime security tools for system policies, enabling comprehensive security management across multiple policy domains. Additionally, beyond policy generation, our system includes a multi-step validation process to prevent misconfigurations and a set-theoretic conflict detection mechanism to identify inter-policy conflicts in advance. KubeTeus+ has demonstrated impressive results in both entity classification and policy generation across network and system policy. 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 security policies. Furthermore, the ablation study validates that each component of the cognition-driven architecture independently contributes to generation quality, with prompt enhancement and fine-tuning serving complementary roles that together enable accurate and deployable policy generation. This research provides a valuable reference implementation for intent-based security policy generation and suggests a new direction for comprehensive security policy management in cloud-native environments.

The authors have made the code publicly accessible at Kim and Lee (2025a) and the data at Kim and Lee (2025b), and plan to update this with an expanded version upon this paper.

CRediT authorship contribution statement

Bom Kim: Writing – review & editing, Writing – original draft, Validation, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Seungwon Shin:** Supervision, Writing – review & editing. **Seungsoo Lee:** Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2025-16069415).

Data availability

The data and code supporting this study are openly available at https://github.com/cclab-inu/KubeTeus and https://huggingface.co/datasets/cclabinu/kubeteus.

References

Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M., 2019. Optuna: A next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

Alliance, C.S., 2025. Top threats to cloud computing - deep dive 2025. https://cloudsecurityalliance.org/artifacts/top-threats-to-cloud-computing-2025.

Almehmadi, A., El-Khatib, K., 2015. On the possibility of insider threat prevention using intent-based access control (IBAC). IEEE Syst. J. 11 (2), 373–384.

Anchore, 2025. Grype. https://github.com/anchore/grype.

Asia, T., 2023. 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/.

Banerjee, S., Lavie, A., 2005. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In: Proceedings of the Acl Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/Or Summarization. pp. 65–72.

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

Cilium Authors, 2025a. Cilium. https://cilium.io/.

Cilium Authors, 2025b. Cilium Network Policy. https://docs.cilium.io/en/latest/security/policy/.

Cinque, M., Della Corte, R., Pecchia, A., 2022. Micro2vec: Anomaly detection in microservices systems by mining numeric representations of computer logs. J. Netw. Comput. Appl. 208, 103515.

Cisco, 2025. Martian Bank: A microservices demo application. https://github.com/cisco-open/martian-bank-demo.

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

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

DeepSeek AI, 2025. DeepSeek-Coder-7B-Instruct-v1.5. https://huggingface.co/deepseek-ai/deepseek-coder-7b-instruct-v1.5.

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

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

Dzeparoska, K., Lin, J., Tizghadam, A., Leon-Garcia, A., 2023. LLM-based policy generation for intent-based management of applications. In: 2023 19th International Conference on Network and Service Management. CNSM, IEEE, pp. 1–7.

Flexential, 2025. 2025 state of AI infrastructure report. https://www.flexential.com/resources/report/2025-state-ai-infrastructure.

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

Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., et al., 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 3–18.

Ghavamnia, S., Palit, T., Benameur, A., Polychronakis, M., 2020. Confine: Automated system call policy generation for container attack surface reduction. In: 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020). pp. 443–458.

Google, 2025. CodeGemma-7b-it. https://huggingface.co/google/codegemma-7b-it.

Google Cloud, 2025a. 2025 research report: State of AI infrastructure. https://services.google.com/fh/files/custombib/google_cloud_state_of_ai_infra_report.pdf.

Google Cloud, 2025b. AI business trends 2025. https://services.google.com/fh/files/custombib/google_cloud_ai_trends.pdf.

Google Cloud, 2025c. Online Boutique: A cloud-first microservices demo application. https://github.com/GoogleCloudPlatform/microservices-demo.

Han, R., Wang, J., Sun, H., Jiang, Z., Qi, Q., Zhuang, Z., Zhang, Y., Liao, J., 2025. Network copilot: Intent-driven network configuration updating for service guarantee. In: IEEE INFOCOM 2025-IEEE Conference on Computer Communications. IEEE, pp. 1–10.

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

Her, J., Kim, J., Kim, J., Lee, S., 2025. An in-depth analysis of eBPF-based system security tools in cloud-native environments. IEEE Access.

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

Hugging Face, 2025a. AutoTrain. https://huggingface.co/autotrain.

Hugging Face, 2025b. Transformers Pipelines. https://huggingface.co/docs/transformers/main_classes/pipelines.

IDC, 2025. Worldwide quarterly enterprise infrastructure tracker: Buyer and cloud deployment. https://my.idc.com/getdoc.jsp?containerId=IDC_P31615.

Intel, 2025. AI infrastructure in 2025: Balancing datacenter and cloud investments. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2025-02/idc-ai-infrastructure-balancing-dc-and-cloud-investments-brief.pdf.

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

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

Kang, H., Shin, S., 2022. Verikube: Automatic and efficient verification for container network policies. IEICE Trans. Inf. Syst. 105 (12), 2131–2134.

Khan, M.A., 2016. A survey of security issues for cloud computing. J. Netw. Comput. Appl. 71, 11–29.

Kim, B., Kim, J., Lee, S., 2025a. Exploring security enhancements in kubernetes CNI: A deep dive into network policies. IEEE Access.

Kim, B., Lee, S., 2024. KUBEAEGIS: A unified security policy management framework for containerized environments. IEEE Access.

Kim, B., Lee, S., 2025a. KubeTeus^{+}. https://github.com/cclab-inu/KubeTeus.

Kim, B., Lee, S., 2025b. KubeTeus^{+} Dataset. https://huggingface.co/datasets/cclabinu/kubeteus.

Kim, B., Park, H., Lee, S., 2025b. KUBETEUS: An intelligent network policy generation framework for containers. In: IEEE INFOCOM 2025-IEEE Conference on Computer Communications. IEEE, pp. 1–10.

Kim, J., Ujcich, B.E., Tian, D.J., 2023. Intender: Fuzzing {Intent-Based} Networking with {Intent-State} transition guidance. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 4463–4480.

Lee, S., Nam, J., 2023. Kunerva: Automated network policy discovery framework for containers. IEEE Access.

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

Li, Y., Hu, X., Jia, C., Wang, K., Li, J., 2022. Kano: Efficient cloud native network policy verification. IEEE Trans. Netw. Serv. Manag..

Li, Y., Jia, C., Hu, X., Li, J., 2020. Kano: Efficient container network policy verification. In: 2020 IEEE Symposium on High-Performance Interconnects. HOTI, IEEE, pp. 63–70.

Li, J., Sun, A., Han, J., Li, C., 2020. A survey on deep learning for named entity recognition. IEEE Trans. Knowl. Data Eng. 34 (1), 50–70.

Lin, C.-Y., 2004. Rouge: A package for automatic evaluation of summaries. In: Text Summarization Branches Out. pp. 74–81.

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

Ma, K., Yang, G., Xiang, Y., 2020. RCBAC: A risk-aware content-based access control model for large-scale text data. J. Netw. Comput. Appl. 167, 102733.

Mckinsey & Company, 2024. AI power: Expanding data center capacity to meet growing demand. https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/ai-power-expanding-data-center-capacity-to-meet-growing-demand.

Meta AI, 2025a. Meta-Llama-3-8B-Instruct. https://huggingface.co/meta-llama/Meta-Llama-3-8B-Instruct.

Meta AI, 2025b. CodeLlama-7b-Instruct-hf. https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf.

Mistral AI, 2025. Mistral-7B-Instruct-v0.2. https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2.
Multus Authors, 2025. Multus CNI. https://github.com/k8snetworkplumbingwg/multus-cni.
National Institute of Standards and Technology, 2025. CVE-2025-31498. https://nvd.nist.gov/vuln/detail/cve-2025-31498.
OCP Power Demos, 2025. Sock Shop: A microservices demo application. https://github.com/ocp-power-demos/sock-shop-demo.
Otterize, 2025. Otterize Intents Operator. https://github.com/otterize/intents-operator.
Papineni, K., Roukos, S., Ward, T., Zhu, W.-J., 2002. Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. pp. 311–318.
Popović, M., 2017. chrF++: words helping character n-grams. In: Proceedings of the Second Conference on Machine Translation. pp. 612–618.
Qi, S., Kulkarni, S.G., Ramakrishnan, K., 2020. Assessing container network interface plugins: Functionality, performance, and scalability. IEEE Trans. Netw. Serv. Manag. 18 (1), 656–671.
Ryu, S., Do, H., Kim, Y., Lee, G.G., Ok, J., 2024. Key-element-informed sLLM tuning for document summarization. arXiv preprint arXiv:2406.04625.
Santos, J., Reppas, E., Wauters, T., Volckaert, B., De Turck, F., 2025. Gwydion: Efficient auto-scaling for complex containerized applications in kubernetes through reinforcement learning. J. Netw. Comput. Appl. 234, 104067.
Sheng, S., Che, K., Mi, A., Wan, X., 2023. Network security mechanism optimization strategy in cloud native scenario. In: 2023 6th International Conference on Electronics Technology. ICET, IEEE, pp. 614–618.
Siino, M., Giuliano, F., Tinnirello, I., 2025. Integrating large language models into network testbeds: A novel approach for automated experimentation and optimization. J. Netw. Comput. Appl. 104281.
Singh, A., Chatterjee, K., 2017. Cloud security issues and challenges: A survey. J. Netw. Comput. Appl. 79, 88–115.
Singh, S., Jeong, Y.-S., Park, J.H., 2016. A survey on cloud computing security: Issues, threats, and solutions. J. Netw. Comput. Appl. 75, 200–222.
Singh, V., Peddoju, S.K., 2017. Container-based microservice architecture for cloud applications. In: 2017 International Conference on Computing, Communication and Automation. ICCCA, IEEE, pp. 847–852.
Srirama, S.N., Adhikari, M., Paul, S., 2020. Application deployment using containers with auto-scaling for microservices in cloud environment. J. Netw. Comput. Appl. 160, 102629.
Sun, P., 2020. Security and privacy protection in cloud computing: Discussions and challenges. J. Netw. Comput. Appl. 160, 102642.
Sun, H., Huang, Q., Sun, J., Wang, W., Li, J., Li, F., Bao, Y., Yao, X., Zhang, G., 2024. {AutoSketch}: Automatic {Sketch-Oriented} compiler for query-driven network telemetry. In: 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). pp. 1551–1572.
The containerd Authors, 2025. containerd: An open and reliable container runtime. https://github.com/containerd/containerd.
The Gatekeeper Authors, 2025. Gatekeeper. https://github.com/open-policy-agent/gatekeeper.
The Istio Authors, 2025. Bookinfo Application. https://istio.io/latest/docs/examples/bookinfo/.
The KubeArmor Authors, 2025a. KubeArmor. https://github.com/kubearmor/KubeArmor.
The KubeArmor Authors, 2025b. KubeArmor Policy. https://docs.kubearmor.io/kubearmor/documentation/security_policy_specification.
The Kubernetes Authors, 2025. Kubernetes: Production-grade container orchestration. https://kubernetes.io/.
The Kubernetes Authors, 2025a. Extend the Kubernetes API with CustomResourceDefinitions. https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/.
The Kubernetes Authors, 2025b. Kubernetes Network Policy. https://kubernetes.io/docs/concepts/services-networking/network-policies/.
The Open Policy Agent Authors, 2025. Open Policy Agent. https://www.openpolicyagent.org/.
The Tetragon Authors, 2025a. Tetragon. https://github.com/cilium/tetragon.
The Tetragon Authors, 2025b. Tetragon Tracing Policy. https://tetragon.io/docs/concepts/tracing-policy/.
Tigera, 2025a. Calico Network Policy. https://docs.tigera.io/calico/latest/network-policy/get-started/calico-policy/calico-network-policy.
Tigera, 2025b. Project Calico. https://www.tigera.io/project-calico/.
Togay, C., Kasif, A., Catal, C., Tekinerdogan, B., 2021. A firewall policy anomaly detection framework for reliable network security. IEEE Trans. Reliab. 71 (1), 339–347.
Van Tu, N., Yoo, J.-H., Hong, J.W.-K., 2024. 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, pp. 1–8.
van Vugt, T.M., Malik, T., 2023. A practical analysis of open-source security tools in microservice kubernetes environments. In: 2023 Cyber Research Conference-Ireland (Cyber-RCI). IEEE, pp. 1–8.
Xu, K., Zhang, B., Li, J., He, H., Ren, R., Ren, J., 2025. DRacv: Detecting and auto-repairing vulnerabilities in role-based access control in web application. J. Netw. Comput. Appl. 104191.
Xu, S., Zhou, Q., Huang, H., Jia, X., Du, H., Chen, Y., Xie, Y., 2023. Log2Policy: An approach to generate fine-grained access control rules for microservices from scratch. In: Proceedings of the 39th Annual Computer Security Applications Conference. pp. 229–240.
Yang, S., Kang, B.B., Nam, J., 2024. Optimus: association-based dynamic system call filtering for container attack surface reduction. J. Cloud Comput. 13 (1), 71.
Yang, H., Zhan, K., Bao, B., Yao, Q., Zhang, J., Cheriet, M., 2021. Automatic guarantee scheme for intent-driven network slicing and reconfiguration. J. Netw. Comput. Appl. 190, 103163.
Zahoor, E., Chaudhary, M., Akhtar, S., Perrin, O., 2023. A formal approach for the identification of redundant authorization policies in kubernetes. Comput. Secur. 135, 103473.
Zeng, Q., Kavousi, M., Luo, Y., Jin, L., Chen, Y., 2023. Full-stack vulnerability analysis of the cloud-native platform. Comput. Secur. 129, 103173.
Zhang, X., Lin, Z., Hu, X., Wang, J., Lu, W., Zhou, D.-Y., 2025. SECON: Maintaining semantic consistency in data augmentation for code search. ACM Trans. Inf. Syst. 43 (2), 1–26.