KubeScribe: LLM-Driven Automation of Runtime Security Policies in Cloud-Native Environments

Jaeyoung Lee
Department of AI-Based Convergence
Dankook University
leeja042499@dankook.ac.kr

Jaehyun Nam*
Department of Computer Engineering
Dankook University
namjh@dankook.ac.kr

***Abstract*—Cloud-native infrastructures built on Kubernetes accelerate deployment and operations but broaden the runtime attack surface beyond traditional defenses. Runtime security solutions enforce fine-grained policies at the kernel and orchestration layers, yet policy authoring remains complex due to heterogeneous schemas and rapidly evolving workloads. Existing automation approaches focus mainly on network rules or syscall-level seccomp profiles, leaving system-level runtime policies unexplored. We present *KubeScribe*, the first framework to automate file-, process-, and syscall-aware runtime security policies using large language models. *KubeScribe* integrates runtime log analysis with LLM-based synthesis and applies dual validation (CRD schema and resource-level checks) to ensure safe deployment. Evaluations on KubeArmor, Tetragon, and Cilium show compile success rates improving from below 10% to above 66% and coverage exceeding 90%, with substantial gains in BLEU, ROUGE, and METEOR metrics. These results demonstrate that *KubeScribe* provides a practical and generalizable path toward automated runtime security in Kubernetes environments.**

***Index Terms*—Runtime Security, Policy Automation, LLM**

I. Introduction

Cloud-native computing has reshaped the design and deployment of modern software systems. Containers encapsulate application code, libraries, and dependencies into portable units, enabling lightweight, reproducible execution across heterogeneous environments. Orchestration platforms such as Kubernetes [1] have become the de facto standard for managing these workloads, offering automated scaling, rolling updates, and resilient deployment strategies [2]. These capabilities accelerate software innovation, yet also create infrastructures that are highly dynamic and distributed. The runtime attack surface expands as containers are created and destroyed in real time, service meshes and container network interfaces (CNIs) dynamically reconfigure network paths, and workload behavior changes with each software update [3]. Conventional defenses such as firewalls, intrusion detection systems, and image scanning, while valuable at the perimeter, are insufficient to prevent runtime compromise once workloads are active.

To address these challenges, the focus of cloud-native security has shifted toward runtime monitoring and enforcement [4]. Runtime security solutions instrument workloads using eBPF to observe fine-grained system behavior, including system calls, file I/O, process executions, and network flows. By enforcing policies at the kernel or orchestration layer, these solutions can detect and block malicious activity as it occurs. Representative systems include KubeArmor [5] and Tetragon [6], which provide host-level controls over files and processes, and Cilium [7] or Kubernetes Network Policy [8], which regulate inter-service communication. Together, these systems form the backbone of runtime protection for Kubernetes environments. Their effectiveness, however, hinges on the availability of precise, workload-aware policies. Writing such policies is a complex and error-prone task: schemas differ across enforcement engines, legitimate behaviors evolve rapidly with code and configuration changes, and mistakes can either disrupt application availability through excessive restriction or expose vulnerabilities through under-specification.

The difficulty of authoring policies has motivated research on automation. Log-based approaches [9], [10] attempt to infer policies by analyzing observed runtime events, achieving reasonable accuracy for specific engines. However, they lack expressiveness, are tied to engine-specific log schemas, and cannot generalize across heterogeneous platforms. LLM-based approaches [11] offer greater flexibility by translating natural language prompts into policies, but in practice they require operators to specify exact resource paths and arguments, which is impractical in complex deployments. More importantly, existing work has concentrated primarily on network policies or syscall-only seccomp profiles. While useful in isolation, these abstractions do not capture the contextual enforcement semantics required by modern runtime security engines. To the best of our knowledge, no prior research has demonstrated LLM-driven automation of *system-level* runtime security policies for engines such as KubeArmor or Tetragon, which enforce contextual rules across files, processes, and syscalls.

This paper presents *KubeScribe*, a Kubernetes-native framework that advances policy automation beyond prior work by supporting not only network rules but also fine-grained system-level runtime security policies. Unlike previous approaches focused narrowly on network segmentation or syscall whitelisting, *KubeScribe* targets comprehensive enforcement engines including KubeArmor and Tetragon, enabling contextual enforcement of files, processes, and syscalls. To our knowledge, this is the first study to apply large language models (LLMs) to the automation of system-level runtime policies, filling a critical gap in current research.

* Jaehyun Nam (namjh@dankook.ac.kr) is the corresponding author.

KubeScribe introduces two key innovations. First, it tightly integrates runtime log analysis with LLM synthesis, enriching abstract user intents with observed execution paths and workload context before translating them into engine-specific policies. This hybrid design overcomes the brittleness of purely log-based methods, which lack generality, and purely LLM-based methods, which lack contextual grounding. Second, it implements a robust dual validation pipeline that combines CRD schema verification with resource-level checks. This ensures that generated policies are not only syntactically correct but also semantically aligned with actual workloads, providing deployable outputs that avoid service disruption. Together, these mechanisms deliver higher accuracy, stronger reliability, and safer operation.

We implement and evaluate *KubeScribe* on Kubernetes microservice deployments with KubeArmor, Tetragon, and Cilium. Using QLoRA-fine-tuned LLMs, *KubeScribe* raises compile success rates from below 10% to above 66% and achieves over 90% coverage. It also improves BLEU, ROUGE, and METEOR scores, demonstrating linguistic fidelity and practical deployability. These results confirm that *KubeScribe* provides a unified, low-overhead, and effective path toward automated runtime protection in cloud-native environments.

Contributions. This paper makes the following contributions:

- We design an end-to-end automated pipeline spanning log collection, intent parsing, policy synthesis, validation, and enforcement, enabling integrated runtime security policy generation.
- We introduce a hybrid approach that couples runtime log analysis with large language model synthesis, transforming abstract intents into precise, engine-specific policies without requiring manual resource specification.
- We propose a dual validation pipeline that combines CRD schema checks with resource-level consistency verification, preventing syntax errors and reducing false positives and negatives to ensure safe deployment.

Paper Organization. The remainder of this paper is organized as follows. Section II introduces runtime security and LLM fine-tuning, and outlines challenges. Section III presents the design of *KubeScribe*. Section IV shows evaluation results. Section V discusses related work, and Section VI concludes.

II. Background and Motivation

A. Cloud-Native Environments and Runtime Security

Cloud-native architectures combine containerized microservices, ephemeral workloads, automated CI/CD pipelines, autoscaling, and service meshes, which continuously and often unpredictably reshape system topology and trust boundaries in highly dynamic environments. Traditional monolithic systems maintained relatively stable host perimeters, where network firewalls and pre-deployment checks could effectively mitigate many common risks. In cloud-native settings, however, the constantly shifting attack surface makes conventional defenses inadequate, while threats such as container escapes, supply chain compromises, and configuration drifts introduce fundamentally new and dangerous privilege exposures [3], [4].

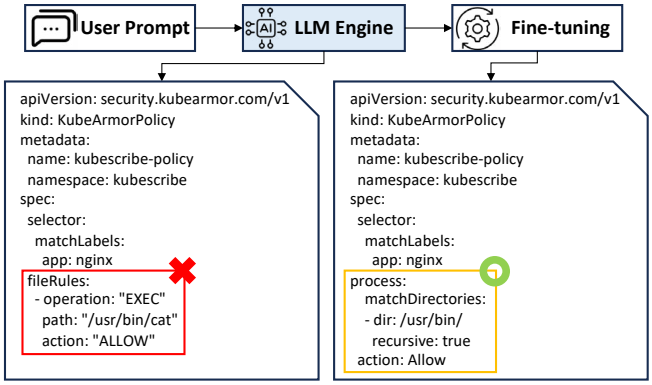


Fig. 1: Impact of QLoRA fine-tuning on policy generation. The fine-tuned LLM generates accurate and generalized policies aligned with user intent, unlike the overly specific rules of an unfine-tuned model.

Runtime security addresses this gap by enabling pipelines that monitor execution-time behaviors with low latency. eBPF-based mechanisms insert lightweight probes into the kernel to expose contextual event streams, including system calls, file access attempts, process creation, and network connections. These events can be aggregated into workload-specific behavioral profiles, supporting behavior-aware policies and real-time responses. By bridging the gap between deployment-time configurations and runtime activities, runtime security has become indispensable for constraining the attack surface of production infrastructures.

B. Large Language Models with Fine-tuning

Large Language Models (LLMs) have expanded from natural language tasks into domains requiring structured outputs such as code or configuration generation [12]. However, pre-trained LLMs are trained on general-purpose corpora, limiting their ability to capture domain-specific formats and structural conventions [13]. Without adaptation, they often generate syntactically valid but semantically inconsistent security policies, where strict schemas and field relationships must be preserved.

Fine-tuning has emerged as a practical approach for adapting general-purpose models to specialized domains [14]. Methods such as LoRA (Low-Rank Adaptation) and QLoRA (Quantized LoRA) [15], [16] provide parameter-efficient adaptation without retraining full models. These methods allow models to internalize domain-specific structures, improving the fidelity of generated artifacts such as Kubernetes policies. Figure 1 shows that QLoRA fine-tuning produces generalized and reusable KubeArmor policies aligned with operator intent, whereas unfine-tuned models often generate brittle rules. In short, while LLMs can encode complex patterns, their reliable use in runtime security depends on lightweight fine-tuning and exposure to structured policy corpora.

C. Challenges in Policy Automation

Policy-based security frameworks are the de facto means of enforcing runtime protection in cloud-native systems. How-

ever, writing such policies requires precise specification, and the quality of manually authored rules is highly dependent on operator expertise. In practice, policy quality varies significantly with operator skill, leading to risks such as blocking essential services or permitting malicious actions. To mitigate this challenge, recent studies have explored automated policy generation. Existing approaches can be broadly classified into log-based methods and LLM-based methods.

1) Limitations of Log-based Approaches: Log-based approaches [17]–[19] generate policies by collecting and analyzing runtime events. When logs are comprehensive, they can achieve relatively high accuracy for the target security engine. However, these methods are inherently tied to the event formats and schemas of specific engines, making consistent cross-engine policies difficult to achieve. They also assume that observed logs sufficiently represent legitimate workload behavior, which is not always the case in dynamic environments. Furthermore, logs often lack ground truth labels distinguishing benign from malicious actions, forcing tools to treat all observed behaviors as valid. This limitation undermines reliability and introduces risks of codifying insecure practices into policies. As a result, log-based automation struggles in multi-engine environments where systems such as KubeArmor, Tetragon, and Cilium must coexist and remain consistent.

2) Limitations of LLM-based Approaches: LLM-based approaches [11], [20] generate policies directly from natural language prompts, offering tool-agnostic flexibility and broad adaptability. In theory, this paradigm allows operators to express high-level intents, which the model translates into precise enforcement rules. In practice, however, operators often lack the deep system-level insight required to craft accurate prompts. When prompts are underspecified or ambiguous, generated policies may misrepresent essential security requirements. Such mis-specified policies risk either blocking legitimate workloads or failing to prevent adversarial behavior. Moreover, while Zero-Trust principles [21] advocate denying all actions by default, their practical adoption still depends on detailed knowledge of workload behavior to define exceptions. This knowledge cannot be assumed to reside solely with operators, especially in large-scale deployments with heterogeneous services. Consequently, LLM-based automation alone is insufficient for safe runtime enforcement without additional grounding in execution context.

III. *KubeScribe* Design

Prior log- or LLM-based approaches to policy generation remain incomplete and do not scale reliably in Kubernetes. *KubeScribe* addresses this gap by combining runtime log analysis with LLM-driven synthesis, enriching abstract intents with execution context and translating them into engine-specific policies. This integration reduces manual effort, prevents misspecification, and delivers expressive, enforceable policies across heterogeneous runtime security tools.

A. Overall Architecture

Figure 2 illustrates the five-stage pipeline of *KubeScribe*: Log Collection, Intent Parsing, Prompt Building, Policy Generation, and Validation. Runtime events are continuously collected and stored in a structured database, providing the foundation for workload-aware enforcement. User prompts are translated into structured specifications, enriched with execution context, and reformulated into engine-specific inputs for LLM synthesis. The resulting policies undergo dual validation, including CRD schema checks and resource-level verification, before deployment. Policies that fail validation enter an automatic regeneration loop, ensuring that only syntactically correct and semantically consistent versions reach production. By combining log-driven context with LLM synthesis and layered validation, this architecture enables precise and dependable policy creation in dynamic Kubernetes environments.

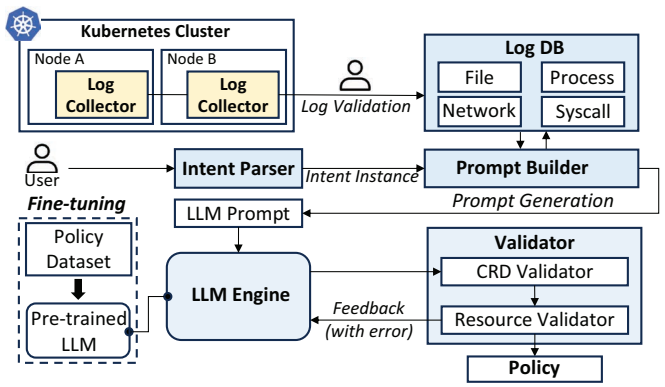


Fig. 2: Overall architecture of *KubeScribe*, showing the pipeline from log collection and intent parsing to LLM-based policy generation and validation.

In contrast to prior approaches limited to log-based inference or one-shot LLM translation, *KubeScribe* integrates context enrichment with validation throughout the pipeline to guarantee deployable outputs. This design directly addresses the twin challenges of policy accuracy and operational safety in production-scale Kubernetes clusters.

B. Log Collection

The Log Collection stage forms the foundation of *KubeScribe* by capturing runtime events that are later transformed into enforceable policies. Unlike prior systems relying on coarse or single-source logs, *KubeScribe* gathers heterogeneous event streams in real time, including system calls, file accesses, and network connections from Kubernetes workloads. These logs are preprocessed to extract relevant attributes such as pod metadata, syscall arguments, and resource identifiers, while noisy or redundant entries are filtered out. Each entry is annotated with a validity tag that distinguishes between allowed and disallowed events, ensuring clarity in policy synthesis. Newly observed behaviors in the operational environment can be flagged for administrator approval, balancing automation with safety. The processed logs are organized into domain-specific tables for processes, files, and network flows, which then serve as reliable references for intent enrichment and validation in later stages, enabling policies that reflect both workload semantics and operational safety.

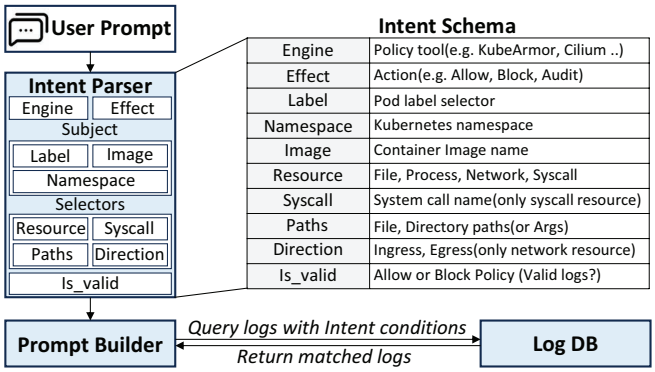


Fig. 3: Workflow of the User Intent Parser, which converts abstract user prompts into a structured intent schema. Extracted schema fields guide prompt building and enable querying of the log database for relevant execution traces.

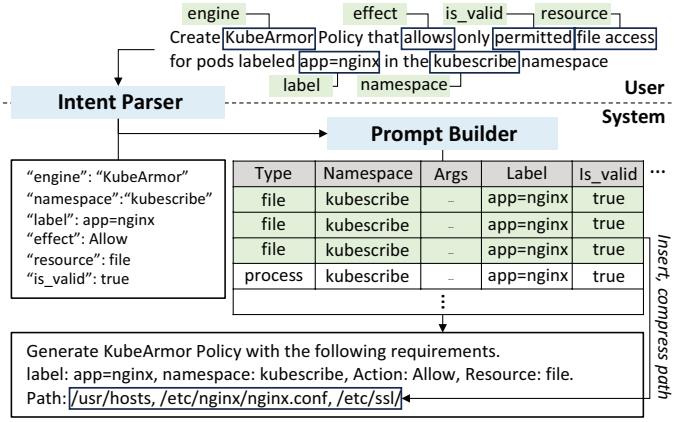
C. Intent Parsing

The Intent Parsing stage translates high-level user prompts into structured specifications that drive policy generation. As shown in Figure 3, the parser converts free-form input into an Intent Instance aligned with a predefined JSON Schema. Enforcing the schema not only provides a predictable format for downstream automation but also prevents incomplete or ambiguous inputs from propagating errors. Extracted attributes include the target engine, namespace, labels, network direction, and intended enforcement action, all mapped to common domains such as file, process, and network control to remain engine-agnostic. The parser augments this information with log queries, grounding abstract intent in observed execution traces. Leveraging a pre-trained LLM, the parser performs semantic interpretation of diverse prompt styles and resolves ambiguities that rule-based methods cannot, ensuring that operator intent is captured consistently and safely for subsequent policy synthesis.

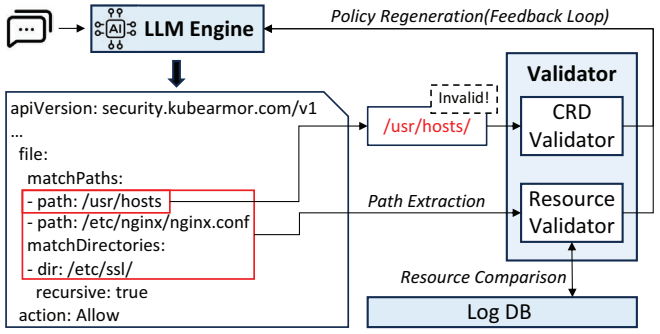
D. Prompt Building and Policy Generation

KubeScribe constructs LLM-ready prompts by combining the structured Intent Instance with enriched log data. The Prompt Builder reformulates extracted attributes into a compact and uniform representation for the fine-tuned LLM. Each prompt consists of two parts: (i) the target security engine and its scope (namespace and labels), and (ii) contextual workload data such as file paths, IP addresses, and ports drawn from the log database. To remain efficient, workload attributes are compressed using prefix summarization and clustering, ensuring that prompts capture essential patterns without exceeding token limits. Figure 4(a) illustrates an example where access paths from pods in the `kubescribe` namespace are aggregated and compressed before inclusion.

The LLM Engine then generate executable policies for multiple enforcement backends. Because prompts follow a standardized schema, the model delivers stable outputs across heterogeneous engines despite their differing grammars. Compact instruction-tuned LLMs were adapted using the QLoRA



(a) Intent parsing and prompt construction



(b) Policy validation workflow

Fig. 4: Example of policy generation and validation.

method, with training datasets covering host- and network-level corpora that encode both syntactic rules and semantic constraints. Generated policies are subjected to validation prior to deployment, and any failed outputs trigger an automated regeneration cycle. This loop enables policy synthesis that is both expressive and robust in production environments.

E. Policy Validation

The Policy Validation stage ensures that generated policies are both syntactically correct and semantically meaningful. As shown in Figure 4(b), validation proceeds in two layers. The first is CRD validation, which enforces compliance with a predefined JSON Schema and guarantees compatibility with the Kubernetes API server. The Second is resource validation, which checks that every file, process, and network resource referenced in the policy is consistent with the structured log database introduced in Section III-B. This step prevents deployment of rules that reference nonexistent or irrelevant entities, thereby reducing false positives and negatives. When inconsistencies are detected, the validator provides feedback to the LLM Engine, initiating an automated regeneration cycle. Through this closed loop, *KubeScribe* ensures that only policies that are executable, consistent, and aligned with observed workloads are admitted into production environments.

TABLE I: Summary of LLMs used in this study with QLoRA

Model Name	#Params	#Trainable
DeepSeek-Coder-7B-Instruct-v1.5 [22]	6.92B	15.7M
Meta-Llama3-8B-Instruct [23]	6.75B	16.7M
Mistral-7B-Instruct-v0.2 [24]	7.26B	13.6M

TABLE II: Summary of Dataset used for fine-tuning LLMs.

Scope	Engine	#Total	#Train	#Test	#Size
Host-Level	KubeArmor	2249	2142	107	17.8M
	Tetragon	1822	1737	85	16.2M
Network-Level	Cilium	1726	1645	81	13.6M
	Tetragon	1761	1678	83	15.7M

IV. Evaluation

A. Evaluation Setup

KubeScribe is implemented in 3.2K lines of Python. The Log Collector, built on Falco, runs as a DaemonSet to capture system call level events. The Intent Parser uses GPT-4 to convert natural language into structured intent instances. The LLM Engine, based on Hugging Face Transformers with QLoRA fine-tuning, generates policies for KubeArmor, Tetragon, and Cilium, covering host and network enforcement.

LLM Engine. We employed three 7B instruction-tuned models (DeepSeek-Coder, CodeLlama3, and Mistral). Table I summarizes parameter counts with QLoRA, and Table II provides dataset details. Policies were collected from GitHub and docs, then augmented with recombined paths and Kubernetes metadata. Five prompt templates improved robustness.

Experiments. Experiments ran on a server with an NVIDIA RTX 4090 GPU (24 GB VRAM). The cluster had one master and two worker nodes (8 vCPUs, 16 GB RAM each), with the GPU dedicated to one worker. The stack used Kubernetes v1.29 and containerd v1.7. We benchmarked on Google's Online Boutique [25], a demo with 10+ microservices using HTTP/REST and gRPC. This setup allowed us to assess the effectiveness of *KubeScribe* in a microservice environment.

B. Intent-to-Policy Generation with Feedback Validation

Here, we describe the operational workflow of *KubeScribe*. Figure 4(a) illustrates how a refined prompt is derived from an abstract user request. For example, a user may request a policy that restricts file access to authorized paths for pods labeled `app=nginx` in the `kubescribe` namespace using the `KubeArmor` engine. The Intent Parser converts this request into a structured schema, which the Prompt Builder uses to extract valid paths from the system log database and construct a detailed LLM input prompt.

The generated prompt is then processed by the LLM Engine to generate a security policy. This policy is forwarded to the Validator, which performs both syntactic and resource-level checks. Figure 4(b) shows the validation process: the CRD Validator first detects schema or syntax errors, and the Resource Validator ensures that the specified paths are valid. For instance, if a directory path ending with "/" is provided instead of a file path, the CRD Validator flags a syntax error. When such an error occurs, the Validator returns feedback to the LLM Engine, prompting automatic correction. This example demonstrates the complete workflow, covering intent parsing, policy generation, and validation, and confirms the framework's practical applicability.

C. LLM Engine Quantitative Performance

We evaluated the policy generation models using natural language generation (NLG) metrics and the practical applicability of generated policies. Five NLG metrics were applied: BLEU, ROUGE-1, ROUGE-2, ROUGE-L, and METEOR. BLEU measures syntactic accuracy through n-gram precision, while ROUGE evaluates reproduction of key policy components, with ROUGE-1 assessing token presence, ROUGE-2 capturing field coherence, and ROUGE-L reflecting sequence similarity. METEOR estimates semantic similarity by considering stemming and lexical variation. These metrics complement one another by capturing syntax, structure, and semantics, enabling multidimensional evaluation. To assess applicability, we also report the compile rate, which measures whether generated policies can be applied to a cluster, and the coverage rate, which reflects how many essential paths from the ground-truth policy are included. Together, these metrics capture both linguistic quality and deployment feasibility.

Table III shows that fine-tuning substantially improved performance across all measures. At baseline, 7B-parameter models performed poorly, with Coverage below 40% and Compile below 10%, while GPT-4o, pretrained on a larger corpus, outperformed baseline models even without fine-tuning. After fine-tuning, all three models improved significantly: DeepSeek-Coder-7B achieved a BLEU score of 0.93 and ROUGE-L of 0.96, surpassing GPT-4o and recording a Compile rate of 75% with Coverage of 93%. CodeLLaMA-3-7B improved by about 30% across quality metrics, while Mistral-7B improved by an average of 34%. Initial Compile rates were near zero due to limited knowledge of policy schemas, but repeated fine-tuning raised them more than tenfold. Overall, all three models approached or exceeded GPT-4o performance after fine-tuning, with DeepSeek-Coder-7B showing the best results. These findings highlight the importance of task-specific fine-tuning for improving both fidelity and deployability of security policies.

V. Related Work

Policy Automation. Automation of container security policies has focused primarily on Seccomp and network rules. Seccomp policy generation [9], [10], [26] applies static or dynamic analysis to reduce the allowed syscall set, but since Seccomp only determines whether a syscall is invoked, it cannot capture contextual arguments. Network policy generation [11], [19] aggregates observed flows to derive compact rule sets, often coupled with pre-deployment validation, while recent methods [11] employ LLMs to translate user intent into network rules. These studies demonstrate the feasibility of automation, yet their scope is limited: Seccomp remains coarse, and network-centric automation neglects the host-level runtime controls that are essential for comprehensive security.

TABLE III: Automatic-metric results for three LLMs on two policy–generation datasets (higher is better).

Stage	Model	BLEU	ROUGE-1	ROUGE-2	ROUGE-L	METEOR	Compile %	Coverage %
Baseline	DeepSeek-Coder-7B-Instruct-v1.5	0.24	0.43	**0.29**	0.38	**0.38**	**2.1%**	**35.1%**
	Code LLaMA-3-7B-Instruct	**0.27**	0.47	**0.29**	0.41	**0.38**	1.4%	17.3%
	Mistral-7B-Instruct-v0.2	0.24	**0.5**	0.27	**0.43**	0.37	0%	25.4%
	OpenAI-gpt-4	0.49	0.74	0.65	0.7	0.6	49.8%	82%
Fine-tuning	DeepSeek-Coder-7B-Instruct-v1.5	**0.93**	**0.96**	**0.93**	**0.96**	**0.96**	**75%**	**93%**
	Code LLaMA-3-7B-Instruct	0.82	0.89	0.86	0.89	0.94	66%	91%
	Mistral-7B-Instruct-v0.2	0.72	0.85	0.78	0.81	0.89	70%	91%

This gap motivates *KubeScribe*, which extends automation beyond network and syscall whitelisting to fine-grained file, process, and system-level policies.

Runtime Security Solutions. Several runtime security systems exemplify different enforcement paradigms. KubeArmor [5] enforces file and process access control via LSM, Tetragon [6] leverages eBPF for contextual syscall- and event-level filtering, and Cilium [7] focuses on eBPF-based network policy enforcement. Each system addresses only part of the security spectrum, leaving administrators to manually reconcile multiple engines. In contrast, *KubeScribe* integrates these heterogeneous backends into a single automation pipeline, enabling intent-driven policies that span host-level and network-level enforcement while ensuring consistency across engines.

VI. Conclusion

This paper presented *KubeScribe*, an integrated framework for intent-driven automation of runtime and network security policies in Kubernetes environments. By enriching user intents with real-time logs and employing QLoRA-based fine-tuning, the framework generates accurate, engine-specific policies. A dual validation pipeline combining CRD schema checks with resource consistency verification ensures that only safe and deployable policies reach production. Our evaluation demonstrates significant improvements in policy accuracy, coverage, and reliability. Future work will extend the framework to additional security backends and enable dynamic policy adaptation based on real-time feedback, further advancing automated protection for cloud-native workloads.

Acknowledgment

This work was supported by the IITP(Institute of Information & Communications Technology Planning & Evaluation)-ICAN(ICT Challenge and Advanced Network of HRD) grant funded by the Korea government(Ministry of Science and ICT)(IITP-2025-RS-2023-00259867)

References

[1] "Kubernetes," https://kubernetes.io/.
[2] S. Deng, H. Zhao, B. Huang, C. Zhang, F. Chen, Y. Deng, J. Yin, S. Dustdar, and A. Y. Zomaya, "Cloud-native computing: A survey from the perspective of services," *Proceedings of the IEEE*, vol. 112, no. 1, pp. 12–46, 2024.
[3] P. Billawa, A. Bambhore Tukaram, N. E. Díaz Ferreyra, J.-P. Steghöfer, R. Scandariato, and G. Simhandl, "Sok: Security of microservice applications: A practitioners' perspective on challenges and best practices," in *Proceedings of the International Conference on Availability, Reliability and Security*, 2022, pp. 1–10.
[4] ARMO, "2025 State of Cloud Runtime Security: It's Time to Shift to Cloud-Native Approaches," 2025.
[5] Accuknox, "KubeArmor: Runtime Security Enforcement System for Cloud-Native Workloads," https://kubearmor.io/.
[6] Cilium, "Tetragon: eBPF-based Security Observability," https://tetragon.io/.
[7] "Cilium Network Policy," https://docs.cilium.io/en/latest/security/policy/.
[8] "Network Policies," https://kubernetes.io/docs/concepts/services-networking/network-policies/.
[9] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 443–458.
[10] M. Pancholi, A. D. Kellas, V. P. Kemerlis, and S. Sethumadhavan, "Timeloops: Automatic System Call Policy Learning for Containerized Microservices," *arXiv preprint arXiv:2204.06131*, 2022.
[11] H. P. Kim, Bom and S. Lee, "KUBETEUS: An Intelligent Network Policy Generation Framework for Containers," in *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*, 2025.
[12] M. Chen and et al., "Evaluating Large Language Models Trained on Code," *arXiv preprint arXiv:2107.03374*, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374
[13] S. Geng, H. Cooper, M. Moskal, S. Jenkins, J. Berman, N. Ranchin, R. West, E. Horvitz, and H. Nori, "Generating structured outputs from language models: Benchmark and studies," *arXiv e-prints*, pp. arXiv–2501, 2025.
[14] Z. Han, C. Gao, J. Liu, J. Zhang, and S. Q. Zhang, "Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey," *arXiv preprint arXiv:2403.14608*, 2024.
[15] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-Rank Adaptation of Large Language Models," in *Proceedings of the Tenth International Conference on Learning Representations (ICLR)*, 2022.
[16] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient Finetuning of Quantized LLMs," in *Advances in Neural Information Processing Systems (NeurIPS 2023)*, 2023, pp. 10 088–10 115.
[17] N. Lopes, R. Martins, M. E. Correia, S. Serrano, and F. Nunes, "Container hardening through automated seccomp profiling," in *Proceedings of International Workshop on Container Technologies and Container Clouds*, 2020, pp. 31–36.
[18] H. Zhu, C. Gehrmann, and P. Roth, "Access security policy generation for containers as a cloud service," *SN Computer Science*, vol. 4, no. 6, p. 748, 2023.
[19] S. Lee and J. Nam, "Kunerva: Automation network policy discovery framework for containers," *IEEE Access*, 2023.
[20] P. Sonune, R. Rai, S. Sural, V. Atluri, and A. Kundu, "LMN: A Tool for Generating Machine Enforceable Policies from Natural Language Access Control Rules using LLMs," *arXiv preprint arXiv:2502.12460*, 2025.
[21] V. Stafford, "Zero Trust Architecture," NIST Special Publication 800-207, 2020.
[22] "DeepSeek-Coder-7B-Instruct-v1.5," https://huggingface.co/deepseek-ai/deepseek-coder-7b-instruct-v1.5/.
[23] "Code LLaMA-7B-Instruct," https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf/.
[24] "Mistral-7B-Instruct-v0.2," https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2/.
[25] G. C. Platform, "Online Boutique Demo Microservices Application," https://github.com/GoogleCloudPlatform/microservices-demo/, 2023.
[26] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating seccomp filter generation for linux applications," in *Proceedings of the Cloud Computing Security Workshop*, 2021, pp. 139–151.