The Illusion of Safety: A Security Audit of LLMSecConfig

Matthew Ball
mgball@uci.edu

G Heemmanshuu Dasari
ghdasari@uci.edu

Andrew Collins
collina2@uci.edu

Nicholas Evangelos Georggin
ngeorggi@uci.edu

Abstract

LLMSecConfig is a recently developed security pipeline that claims to take in and fix security threats in Kubernetes YAML files. We rigorously analyze the pipeline of LLMSecConfig, which is composed primarily of the Checkov security threat assessor and a large language model. We come up with test cases compiled in YAMLTrap, which can produce pipeline failures that may highlight shortcomings in the pipeline's overall design. We also determine areas where the pipeline can improve and where our research may lead us. In short, we are trying to determine what flaws exist so that researchers may prevent those issues from taking place in future implementations.

1 Introduction

Kubernetes is the preferred container orchestration platform in most organizations. A survey by CNCF in 2020 [8] shows a substantial increase from 78% to 96% in enterprises adopting Kubernetes. Businesses with several operating locations adopt Kubernetes for its portability and flexibility. These features make Kubernetes ideal for implementation in various settings, such as on-premises, private clouds, public clouds, or hybrid configurations. It is used to provision applications in a wide range of domains: time series forecasting, edge computing, and high-performance computing.

Kubernetes uses manifests [7], which are typically YAML/JSON files that define resources in the cluster. These resources, like Pod, Deployment, Service, ConfigMap, Secret, Role, and NetworkPolicy, are declarative — the security posture depends heavily on configuration [6], [21]. These configuration files can be complex, and mistakes can lead to critical vulnerabilities, allowing attackers to gain privileged access to the host system, move between services and containers, and steal vast amounts of data from users and the company. For instance, enabling `privileged: true` grants containers nearly unrestricted access to the host, enabling container breakout attacks. Similarly, allowing a writable root filesystem (`readOnlyRootFilesystem: false`) makes it easier for attackers to persist malware or tamper with binaries.

Other frequent misconfigurations [19] include enabling `hostNetwork: true`, which exposes the pod to potential network sniffing or spoofing attacks, and failing to override the default service account, which may grant more permissions than needed. The absence of enforcement mechanisms like PodSecurity Standards (or their deprecated predecessor, PodSecurityPolicies) further increases the risk, allowing insecure pods to be deployed without restriction. These examples highlight a broader issue: many Kubernetes environments remain vulnerable not due to flawed code but because of easily preventable misconfigurations.

Misconfigured Kubernetes manifests can have severe consequences. One of the primary risks is lateral movement, where attackers compromise a single pod and then move across the cluster through service accounts or unrestricted network access. Without mechanisms like NetworkPolicies, an attacker can easily communicate with and exploit other services. Privilege escalation is another serious concern — running containers as root or with settings like host PID or host IPC can allow attackers to interfere with host-level processes. Similarly, broad RBAC (Role-Based Access Control) permissions make it easier to gain higher privileges within the cluster.

These misconfigurations can also lead to denial-of-service (DoS) and data extraction attacks. Sensitive data such as credentials or API keys can be exposed if secrets are hardcoded in manifests or left unprotected in environment variables. Poor access control on volumes can result in unauthorized reads of confidential logs or files. Without CPU and memory limits, a pod can overwhelm the node, either unintentionally or maliciously, disrupting other workloads. These threats are often not isolated — they can compound rapidly, meaning that even one vulnerable YAML file can jeopardize the security of the entire Kubernetes cluster. Kubernetes security configurations are hard to get right as Small security settings like `readOnlyRootFilesystem: true` are easy to miss. Adding

to that, most companies using CI/CD pipelines can amplify these mistakes [4], rapidly deploying insecure configs across their clusters. Also, most developers prioritize speed, while security teams often lack full visibility, which can lead to inconsistent policy enforcement. On top of it all, Kubernetes itself evolves quickly, and security best practices change just as fast, making it hard for teams to keep up.

A lot of tools exist for linting, some of which we will discuss later. However, there are no automated solutions to actually fix the detected misconfigurations, which still remains a laborious task. This can only get more time-consuming as the scale of deployments becomes bigger and bigger. This is where LLMSecConfig [23] comes in, offering a promising route to fixing misconfigurations in Kubernetes using Large Language Models (LLMs). The authors reported high pass rates in correcting popular charts. However, it mainly focuses on well-defined and highly used charts. As organizations increasingly adopt new CI/CD pipelines, there is a significant risk that using systems like LLMSecConfig could report that a cluster "is fixed" yet still harbor exploitations that an adversary could use to harm the business or users. Addressing this gap and rigorously analyzing where these tools fall short is vital to ensure the future security and reliability of container deployments.

To that end, we critically evaluate LLMSecConfig's automated repair pipeline for Kubernetes misconfigurations, going beyond testing on the commonly used Kubernetes manifests.

We make the following key contributions:

Evaluation of LLMSecConfig We use several maliciously created YAML files to test the effectiveness of LLMSecConfig, focusing on possible real-world scenarios where such mistakes can happen. We present and discuss cases of our successful and failed attacks.

YAMLTrap We present the collection of our YAMLs into the YAMLTrap dataset, effective for evaluating the performance of tools that automatically reconfigure YAML files.

Critique of LLMSecConfig & Future directions While not intended as part of our goals, we encountered several significant issues in other areas of LLMSecConfig with regard to their implementation and usage of tools, which hindered our own evaluation. We present and discuss these issues and possible attacks because of the same.

2 Related Works

Kapetanidou et al. [14] provides an extensive analysis of Kubernetes security tools, which together offer multiple levels of defense in the Kubernetes development lifecycle. These tools can be categorized into *static analysis*, *enforcement*, and *runtime protection*. Static analysis tools like *Kube-bench* ensure clusters adhere to the CIS benchmarks, while *Kubesec* [10] evaluates the security of YAML manifests — making them well-suited for catching misconfigurations early in the CI/CD pipeline. Tools like Kube-hunter simulate attacker behavior, identifying weak points such as exposed ports or insecure API access. Trivy [3] combines features of both Kube-hunter and Kube-bench.

For policy enforcement, OPA (Open Policy Agent) with Gatekeeper [9] allows teams to apply custom rules at the Kubernetes API level, preventing insecure configurations like running containers as root. At the runtime layer, Falco detects anomalies such as suspicious system calls, and KubeArmor leverages Linux Security Modules (e.g., AppArmor, SELinux) to actively restrict malicious behavior. These tools, when used together, create a robust "defense-in-depth" posture — addressing threats from code to cluster runtime. However, the aforementioned problem of manual fixing still remains.

When researching harmful ways to configure YAMLs, we discovered some helpful tips collected by Ruud van Asseldonk. He explained that YAML files and the different syntactical elements in their structure make it "such a complex format with so many bizarre and unexpected behaviors, that it is difficult for humans to predict how a given [YAML file] will parse" [22]. For instance, van Asseldonk describes that using data in the same statement while being of different types (i.e. using a numerical digit in a string when an integer is required) can cause YAML files to throw unforeseen errors, as well as dereferencing variables that had never been declared, and how including obselete syntax that has been overwritten by updates. This work was not presented in a Kubernetes format, but it is still a beneficial starting line from which we can discover faulty YAMLs.

3 Approach

3.1 LLMSecConfig

While Static Analysis Tools (SATs) like Checkov [18] are effective at detecting security misconfigurations in Kubernetes manifests, there is little to no work in automatic. LLMSecConfig, proposed as early as February 2025 by Ye et al. [23], is the first framework to address this gap by combining SATs with Large Language Models (LLMs). The system uses a Retrieval-Augmented Generation (RAG) pipeline to generate configuration repairs that aim to maintain both security compliance and operational functionality.

The framework gathers three types of contextual inputs for each misconfiguration, as shown in the architecture in Fig. 1. First, the Checkov SAT output provides the security issues, including policy IDs (e.g., CKV_K8S_93) that were violated. It also provides error locations and severity levels. Second, the source code for each security policy is retrieved from Checkov's GitHub repository. Third, documentation from Prisma Cloud is added to the prompt. This documenta-

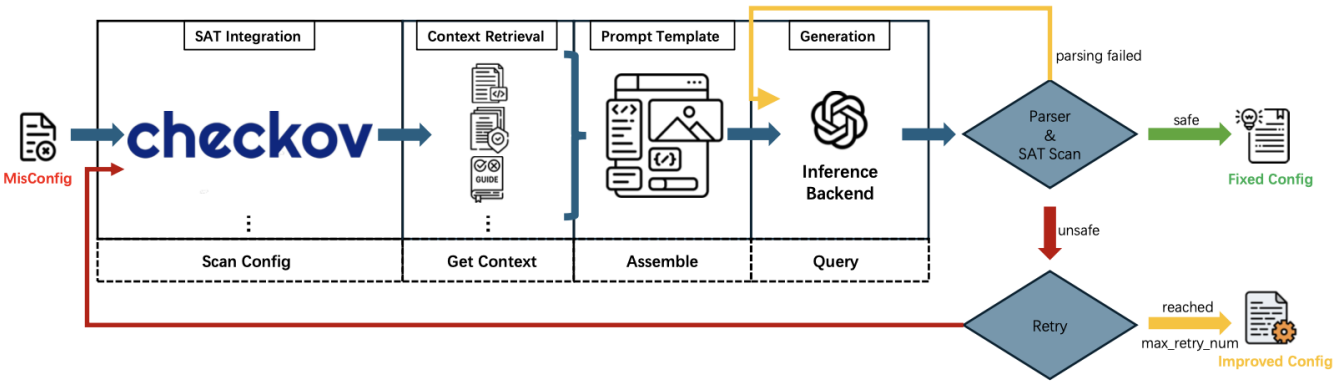

Figure 1: Architecture of LLMSecConfig, as presented in Ye et al. [23].

tion contains practical remediation tips, usage patterns, and secure configuration examples. Together, these elements are compiled into a prompt and passed to an LLM, which then proposes a fix. LLMSecConfig validates the output for YAML correctness and checks whether security issues have been resolved before iterating further.

Despite its high reported repair success rate (94% with Mistral Large 2), LLMSecConfig cannot yet be blindly relied upon in production. Adversarial inputs designed to bypass LLM security reasoning have not been explored yet, and there is no way so far to ensure that these prompts will work across diverse Kubernetes clusters. Additionally, prior work like GenKubeSec [15] shows that while RAG methods can improve precision and recall, these systems still struggle with niche configurations - areas where manual intervention would still be required. This would pose a problem as most organizations take publicly available YAML files and add them to make their own niche configuration. Here, the authors collect their data from the most frequently used files on ArtifactHub and get a high success rate.

3.2 Dataset Collection

To start compiling our harmful YAML collection, we referred to our research on harmful qualities that YAML files may possess. For instance, van Asseldonk [22] presented what he affectionately called the "yaml document from hell" containing a variety of disjoint sections each containing various YAML formatting, syntax, type, and miscellaneous errors that may cause scanners – such as LLMSecConfig – to notify us of failures. Our first 10 YAML files were composed on an instance of the entire "document from hell," as well as individual corresponding to the different individual errors that were described in the documentation. We had also seen the trends in this research, such as what values could be used to throw errors, and made some configuration files with values of our own put in, rounding out our first class of faulty files.

As we have stated in the previous section, these sorts of harmful YAMLs involve the types of data being mismatched, as well as dereferencing variables that were never declared. We had also made YAML files of various size through either having large swaths of data inside or very few keywords in the hopes that one of them would cause errors. In order to have a larger sample size of 40 additional files for future testing, we fed these specifications into GPT-4.1 [17] to see if it could generate a sampling of similarly structured YAML files.

So, we have a class of 50 harmful YAML files with which to run against the pipeline. However, there is one issue that we discovered in this regard: none of the YAML files we have were in Kubernetes format. The LLMSecConfig pipeline assumes that the YAMLs represent Kubernetes pods assuming that this would be used in pod construction in the long run. In light of this development, we repeated the previous step and asked GPT-4.1 [17] to construct 40 new YAML files with the added condition that they would be adapted into the Kubernetes YAML configuration file format. However, we still believed that the misformatted YAMLs were worth a try and were still curious to see how they might perform. With this, we have accumulated our 90 YAML files for our data collection: YAMLTrap.

We additionally want to make clear that we did not ask GPT to make our entire YAMLTrap collection with no work. We had a basis for harmful features based on our own human-produced research that we wanted to replicate. We simply used GPT so that we could accumulate YAML files similar in structure and content to the ones we had already come up with that were also adequately unique from one another so we did not simply repeat the file step again and again (for files 11-90). We also used it for simple format adaptations from generic YAMLs to Kubernetes formatted YAMLs (for files 51-90). It was and is always our intention to comply with all generative AI usage, academic honesty, and citation guidelines in this regard, and we believe that we have done so appropriately.

3.3 Evaluation Setup

The pipeline works by iteratively updating the input YAML file and returning it at the end of the run coupled with a JSON file containing a report. This JSON report contains how many of the security tests the corrected YAML passed and failed. If there is a failure, then the JSON gives a comprehensive description of what the YAML was incapable of performing, what security risk it is associated with, and various metadata.

To perform our tests, we take each YAML file in YAML-Trap and run an instance of the pipeline with the YAML file as the input. After it runs, we then check the JSON returnable at the end to see how that file performed. If there is a failure denoted in the JSON file, then it means that LLMSecConfig could not fully fix the YAML file that it operated on. This could either mean that the YAML had pre-existing qualities that produced a significant security risk or that the LLM tried to make corrections that would not pass the Checkov scan. A number of failure cases occurred during our runs, which denotes that a harmful YAML being placed into the pipeline was successful in being harmful. We have outlined these cases in the following section.

4 Evaluation

4.1 Failed attacks

Of all the test files that we have compiled in YAMLTrap, a staggering number of them were corrected by LLMSecConfig. There were approximately 90 tests, but only 4 failed, giving LLMSecConfig a success rate of 95.6% against YAMLTrap. Our intention was to break the system with a majority of the tests, but we were pleasantly surprised. This showed us that the underlying claim for LLMSecConfig is accurate. Checkov and the LLM can effectively iterate between checking for errors and then generating a new YAML file that removes the security risk that the previous iteration involved. However, this is not absolute, and there are a number of cases in which the YAML remained harmful. We honestly believe that these attacks making their way through LLMSecConfig are more consequential than the number of tests that were caught as intended by the pipeline structure.

4.2 Successful attacks

This second malicious YAML attack we ran was very critical, as it failed three entire checks when running it against Audit LLM SecConfig, for the LLM of gpt-4o-mini. The whole premise of this YAML file was to create a Kubernetes Deployment resource. Most of the metadata and spec of the YAML file seemed normal and valid; however, the main issues arose in the inner spec from the "template" field. The three detected failures are "Capabilities not dropped" (CKV_K8S_37), "Secrets exposed as environment variables" (CKV_K8S_35), and

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secret-dump-logger
spec:
  replicas: 1
  selector:
    matchLabels:
      app: leak
> template: ···
```

Figure 2: Malicious YAML, seemingly safe at the start

"Missing NetworkPolicy for Pod" (CKV2_K8S_6), and as we will discuss, all three of these checks pose a huge security risk if this YAML file were to be deployed in an actual Kubernetes cluster with important secrets/credentials.

The first check that failed was that the Linux capabilities were not dropped when the containers ran, leaving them with more privileges than what was needed. Capabilities are kernel-level privileges (such as "NET_RAW" or "SYS_ADMIN"), which can be abused by many containers, even those that do not have root-level access or something equivalent [5]. What is very dangerous about this type of attack is that it has multiple surfaces/entry points from which it can come. The two main attack entry points include an exploit in the app and a supply chain attack. An exploit within an app is when an attacker gains access to abuse Linux capabilities, becoming able to set raw sockets, modify system time, and mount devices, among many other things. A supply chain attack is when an attacker is able to compromise a base image, and proceeds to silently abuse the gained capabilities, which is dangerous because a naive user would not know what is going on until it is too late [11]. This type of attack is quite realistic, although not as much as the ones we will discuss below. The root cause of it is not using pod security standards (such as PodSecurityAdmission or OPA/Gatekeeper), or even simply over trusting the default container permissions.

The second problem that led to a failed check was that in this YAML file, there was a sensitive secret loaded (called "MY_SECRET"), using a secret key reference (shown in the Figure above). This is insecure because environment variables are visible to anyone who has access to the Kubernetes cluster, or even just access to execute in that specific container [12]. On top of that, the container is literally displaying (via "echo") the secret to the standard output, which can be captured by

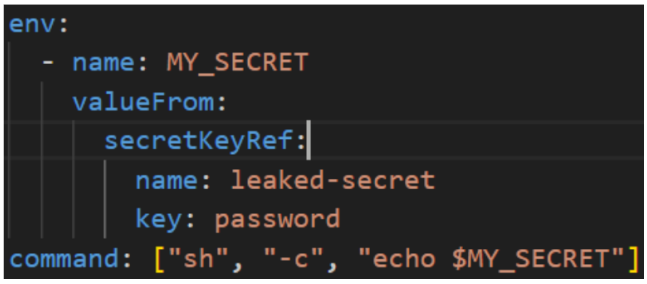

Figure 3: Malicious YAML, exposing secrets

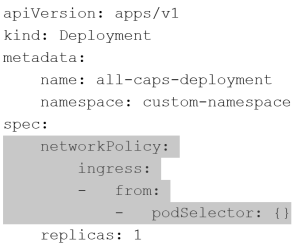

Figure 4: Segment of an LLM-updated YAML file from our YAMLTrap test cases. The highlighted portion emphasizes that a network policy has been added to the file by the LLM and was not originally included. This YAML file was rejected by the pipeline for "lack[ing] an associated NetworkPolicy" as dictated by Checkov, even though one is clearly declared here.

a logging system. What is very crucial to know about this is that secrets remain in memory as plain strings and may not be cleared, so if an attacker gains the ability to access a string variable with an important secret/credential, they can do a lot of harm. Furthermore, if there was any RBAC misconfiguration, then attackers can easily read the secrets, as there would be no safeguards in place to prevent them from doing so [13]. In regard to the realism of the entry points, this attack would be a common, realistic misconfiguration. One common example is displaying database credentials. Imagine an entire database being destroyed because an attacker was able to output the "DBADMIN" credentials and deleted every piece of information within the database.

Last but not least, the third check that failed involved the missing NetworkPolicy. This check is discussed in the previous malicious YAML attack in our paper, so feel free to check out that section for more details on this type of attack if you haven't already.

The error that has occurred with all of the faulty YAMLs that we have tested involves a YAML file that does not possess the data that allows it to establish a network policy. When information is communicated among Kubernetes pods, it must follow an information protocol that dictates which information gets sent between which pods and how. This is similar to how one would use an internet protocol to communicate information between clients and servers over an internet connection; if there is no network policy that has been assigned to a Kubernetes YAML, then it will just default to constructing a pod that cannot send or receive information with other pods to err on the side of caution [1]. While this provides a temporary solution to the problem, it is not very useful if this pod is to be used for any worthwhile Kubernetes operations. In constructing a network policy, the YAML file will require the standard issue specification, API, kind, and metadata keywords, but will also require the keywords 'ingress' and 'egress', each hold data concerning which pods by which ports it would receive data from and which pods by which ports and under what rules it would send data respectively [1]. Both of these are required to be sufficiently outlined so that standard communication between the pods can take place. Say, for example, that we have the network policy omitted in the YAML file that is input into the pipeline. The standard operating procedure would be for Checkov to detect that no network policy is present. It would then transmit this to the large language model to update the YAML to include a network policy. However, this puts the specifications of the network policy completely within the LLM's hands while being disjoint from Checkov. See Fig. 4 for an illustrated representation. There exists the potential for disagreement between the two integral parts of the pipeline. The LLM can make an addition to the YAML that Checkov will flag as a failure no matter what. When Checkov tells the LLM to actually create a network policy, the LLM may stick with the network policy it had added the first time, since, as far as the LLM is concerned, the YAML already has a network policy, so there is no need to construct a new one.

This exploitation is novel because it highlights a glaring flaw in the recently developed and reportedly effective pipeline: misconfigurations brought up by miscommunication between the two primary components. The crux of the network policy's inclusion and eventual rejection stems from the fact that the large language model is responsible for inserting a network policy if one does not exist in the input YAML. It is given this information by the Checkov scan. There are two requirements for this attack to cause a failure of the pipeline: a YAML without a network policy, and the LLM being able to construct a network policy that is faulty. It is very easy to create YAMLs without network policies: just do not include the network policy setup and 'ingress' and 'egress' fields. All that the LLM has to do is create some network policy fields that do not have properly formatted information. This can happen without influence, similar to what has happened during our test cases; we never guided the LLM to add in a faulty network policy. So, we can either depend on random LLM variance to work in our favor, or we can send adversarial prompts into the LLM to make it so that a faulty network policy is most likely to be added. This structure is not so different from what we had reviewed in class in Week 6. The paper we read and discussed in class gives us the example of LLMs that can

return classifications that it believes are part of a particular class but would seem different to humans, leading to data that is not robust for outside observers [2]. What would happen if we applied this design to our current situation? The LLM would take in the YAML file and change it so that it includes a network policy. However, the Checkov scanner believes that this network policy is not satisfactory, so it rejects this YAML and prompts the LLM to try again: this updated YAML is not robust in the eyes of Checkov. However, the LLM may not change this YAML because, as far as it is concerned, the YAML already has a network policy, and there is no need to give it one if one already exists: the YAML is accurate in the eyes of the LLM. This process would end up iterating for the entire lifespan of the pipeline run. Just like in [2], the LLM will believe that the network policy is sufficient through a perceived standard accuracy, but the outside observer Checkov will believe that the network policy is not sufficient through a perceived lack in robust accuracy. The LLM and Checkov cannot have different benchmarks for determining the mere existence of something as integral as a network policy. There should not be any misunderstanding between the two as the pipeline may get caught up on frivolous matters such as this instead of what it would actually intend to do: fix the YAML.

4.3 Criticism of pipeline structure

In our evaluation, we found several cricitcal issues in the design and implementation of LLMSecConfig. The first noticeable issue is the poor documentation, making the installation and setup process difficult and cumbersome (in terms of setting up the system based on the instructions in the ReadMe.md). There are missing dependencies to run the application that are not found in the included requirements.txt, requiring users to manually debug, find, and install missing dependencies to get the LLMSecConfig provided pipeline working. These are some simple oversights that complicate the process of onboarding new users and could be easily fixed, providing a more enjoyable and seamless experience.

There is also a lack of information on how to set up and integrate Checkov and LIteLLM, two components that are essential to run their pipeline. The absence of this information (e.g., no info on where to put secret keys) leaves a lot of vulnerability in the system and potential for adversarial attacks. On top of this, there are also file names that are misleading and don't really do what the name would entail, making the system less transparent/easy to use for developers or future work on the project.

Stability and fault-tolerance were also major concerns, as no error handling is done and would just cause the entire system to crash, giving little to no valuable feedback. An example of this was when downloading the YAMLs they used from the proxies; they sometimes would miss one or two files, which, when they would come up during processing, would cause the entire application to crash and stop. Issues like this could potentially consume resources with tasks that never finish.

LiteLLM itself is also a security and architectural concern. Being an open-source proxy for commercial LLMs gives well-informed attackers the chance to take advantage of less informed users when errors are found. The service also runs on a predefined port and could suffer from man-in-the-middle attacks, as attackers could implement a service on that port and maliciously inject their own configurations, trying to pass it off as a passed/improved deployment file. Overall, LLMSecConfig provides a great concept, but the implementation lacks robustness, ease of use, and security. These issues and the lack of transparency in how files are changed significantly reduce their usefulness in real-world deployments.

5 Future Work

Building on our project, there are several promising directions for future research and the development of large language models (LLMs) tools that help in the deployment and safety of distributed applications (using tools like Helm, Kubernetes, Podman, Docker, etc.). One key question, though, is whether LLMs are an appropriate or optimal solution to this problem or if alternative approaches should be considered (e.g., decision trees, support vector machines, or a hybrid system with custom rules built in). Future work should explore and investigate these alternative approaches, including different machine learning (ML) models, and measure how better or worse they are at finding and fixing misconfigurations compared to LLMs. Comparative studies could also reveal alternative approaches that companies or users could take that would be more reliable and cost-efficient. A significant challenge with this is the need for an expansive training dataset that consists of both labeled misconfigured files and their correct representation. This dataset does not exist and would be a big step forward in this field. Training smaller domain-specific models or an ensemble of models can also identify key features and potential improvements in fixing deployment files in niche and adversarial environments.

Another key area we considered for future work involves building trust in systems like LLMSecConfig within developer communities. Currently, developers may be hesitant to adopt a black-box system like LLMSecConfig that makes significant changes to its configuration files without some sound reasoning and logs/context that show the changes and improvements the model says it made. Adding a version control system or easy-to-read log output to each file within the pipeline and each iteration through the LLM would significantly improve transparency and go a long way in building trust in larger developer communities. Another improvement to the current system could be to include an interactive option where users can approve, reject, or modify the suggested changes from the LLM during the repair process. This user interface could provide more experienced users with the abil-

ity to provide more context or handle exceptional niche cases needed for their application, as well as ways for the model to work around them. This human-in-the-loop approach is also widely popular in enterprise environments where mistakes carry serious consequences.

Other minor works can be done to improve the overall impact and usability of tools/systems like LLMSecConfig. For example, to broaden the utility, future work should expand beyond YAML files and YAMLTrap to include additional configuration formats, at least covering the variety of formats supported by the static analysis tools (SAT) checkers. Building a hybrid system with heuristics or rules for known edge cases or issues would provide a minimum level of security and some highly needed safeguards. Integration or middleware for integration into already existing tools would help with adoption. Adding additional SAT programs into the system, either as a standalone or ensemble model, could further provide security to this landscape.

6 Conclusion

The project evaluates LLMSecConfig, which is a tool to automate fixing misconfiguration in Kubernetes manifest files using Checkov and LLMs. In this project, the security audit of LLMSecConfig has shown us that using LLMs for security checks on Kubernetes still has a ways to go before developers can blindly trust the results. We developed a new dataset of adversarial manifest files titled YAMLTrap, which has helped reveal key failure points and security flaws in the pipeline.

There were also key flaws in the architecture that were pointed out, including unpredictable retry behavior in the event of parsing issues and excessive crashes due to missing YAML files in predefined JSONs. Also, some edge cases were able to slip past the SAT and LLM due to miscommunications.

Overall, our contributions included creating a novel dataset of adversarial YAML files and conducting a series of substantive tests on the use of LLMs in security ops through LLMSecConfig's architecture. Through this audit, we provided a detailed critique of the design and its coverage of a diverse set of scenarios when it comes to security vulnerabilities in Kubernetes. We hope that the work in this paper helps future research build upon this system to provide more transparency and robust tools for the community.

References

[1] Network Policies. https://kubernetes.io/docs/concepts/services-networking/network-policies/. [Accessed 10-06-2025].

[2] D. Tsipras L. Engstrom B. Tran A. Ilyas, S. Santurkar and A. Madry. Adversarial Examples Are Not Bugs, They Are Features, 2019.

[3] Aquasecurity. Trivy. https://github.com/aquasecurity/trivy, 2025. Accessed 13 June 2025.

[4] A. K. Arani, T. H. M. Le, M. Zahedi, and M. A. Babar. Systematic literature review on application of learning-based approaches in continuous integration. *IEEE Access*, 2024.

[5] Prisma Cloud by Palo Alto Networks. BC_K8S_34 - minimize the admission of containers with capabilities assigned. https://docs.prismacloud.io/en/enterprise-edition/policy-reference/kubernetes-policies/kubernetes-policy-index/bc-k8s-34, 2024.

[6] E. Casalicchio. Container orchestration: A survey. In *Systems Modeling: Methodologies and Tools*, pages 221–235. Springer, 2019.

[7] Carmine Cesarano and Roberto Natella. KubeFence: Security Hardening of the Kubernetes Attack Surface. https://arxiv.org/abs/2504.11126, 2023. Accessed: 2025-06-13.

[8] Cloud Native Computing Foundation. CNCF Survey 2020: The State of Cloud Native Development. https://www.cncf.io/wp-content/uploads/2020/12/CNCF_Survey_Report_2020.pdf, 2020. Accessed: 2025-06-13.

[9] CNCF. Open policy agent gatekeeper. https://github.com/open-policy-agent/gatekeeper, 2025. Accessed 13 June 2025.

[10] controlplane. Kubesec. https://github.com/controlplaneio/kubesec, 2024. Accessed 13 June 2025.

[11] OWASP Foundation. K02 – supply chain vulnerabilities. https://owasp.org/www-project-kubernetes-top-ten/2022/en/src/K02-supply-chain-vulnerabilities, 2022.

[12] OWASP Foundation. K03 – overly permissive rbac configurations. https://owasp.org/www-project-kubernetes-top-ten/2022/en/src/K03-overly-permissive-rbac, 2022.

[13] OWASP Foundation. K08 – secrets management. https://owasp.org/www-project-kubernetes-top-ten/2022/en/src/K08-secrets-management, 2022.

[14] I. A. Kapetanidou, A. Nizamis, and K. Votis. An evaluation of commonly used kubernetes security scanning tools. In *Proceedings of the 2nd International Workshop on MetaOS for the Cloud-Edge-IoT Continuum (MECC '25)*. ACM, 2025.

[15] Ehud Malul, Yair Meidan, Dudu Mimran, Yuval Elovici, and Asaf Shabtai. GenKubeSec: LLM-Based Kubernetes Misconfiguration Detection, Localization, Reasoning, and Remediation, 2024.

[16] Oshrat Nir. Unraveling the State of Kubernetes Security in 2023. https://www.armosec.io/blog/unraveling-the-state-of-kubernetes-security-2023/. [Accessed 08-05-2025].

[17] OpenAI. Gpt-4.1 (version as of 01 june 2025) [large language model]. https://platform.openai.com/playground/prompts?models=gpt-4.1.

[18] Prisma Cloud. Checkov. https://github.com/bridgecrewio/checkov, 2025. Accessed 13 June 2025.

[19] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. Security misconfigurations in open source kubernetes manifests: An empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 32(4):1–36, May 2023.

[20] T. Ritchford. Yaml is an extremely bad choice for any configuration file because it's wildly unpredictable. https://tomswirly.medium.com/yaml-is-an-extremely-bad-choice-for-any-configuration-file-because-its-wildly-unpredictable-d37969d20fef, 2021.

[21] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman. Xi Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices. In *2020 IEEE Secure Development (SecDev)*, pages 58–64. IEEE, 2020.

[22] R. van Asseldonk. The yaml document from hell. https://ruudvanasseldonk.com/2023/01/11/the-yaml-document-from-hell/, 2023. [Accessed 01-06-2025].

[23] Ziyang Ye, Triet Huynh Minh Le, and M. Ali Babar. LLMSecConfig: An LLM-Based Approach for Fixing Software Container Misconfigurations, 2025.