Kubernetes Misconfigurations in the Wild: Taxonomy, Evolution, and Automated Repair with Large Language Models

Anonymous Author(s)

Abstract

Kubernetes has become a central platform for orchestrating cloud-native applications, yet its declarative configuration model frequently introduces security misconfigurations that threaten system reliability and operational stability. Although automated detection tools are widely available, a systematic understanding of misconfiguration patterns and scalable correction mechanisms remains limited. This paper presents a comprehensive empirical study of Kubernetes security misconfigurations based on 2,662 developer-reported issues from Stack Overflow. From this dataset, we derive a structured taxonomy that captures recurring security weaknesses across configuration object types and misconfiguration categories. Using this taxonomy, we analyze how severity levels vary across objects and categories, and examine how security misconfigurations evolve between incubator and stable project stages. Our findings reveal that while some operational issues decrease as projects mature, critical security misconfigurations often persist or reappear, highlighting enduring risk patterns in cloud-native systems. Building on this empirical foundation, we evaluate the effectiveness of Large Language Models (LLMs) in automatically correcting Kubernetes security misconfigurations under progressively enriched contextual conditions. Results demonstrate that contextual grounding significantly improves correction accuracy, with the best standalone model achieving 89.06%. To further enhance structural and semantic reliability, we introduce Kubecurity, a schema-guided validation framework that enforces compliance with official Kubernetes specifications. By combining contextual LLM reasoning with deterministic schema enforcement, the proposed hybrid approach achieves 98.50% correction accuracy while substantially reducing newly introduced misconfigurations. Overall, this work advances both the understanding and automated remediation of Kubernetes security misconfigurations.

CCS Concepts

• **Security and privacy → Software and application security**; • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Keywords

Kubernetes, Cloud-native environments, Automated correction, Large Language Models (LLMs), Taxonomy construction, Severity analysis.

ACM Reference Format:

Anonymous Author(s). 2018. Kubernetes Misconfigurations in the Wild: Taxonomy, Evolution, and Automated Repair with Large Language Models. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Over the past decade, Kubernetes has emerged as the de facto standard for container orchestration in cloud-native environments, enabling automation, scalability, and elasticity across modern distributed systems [4][18][16]. Yet, the very flexibility that empowers these capabilities also introduces substantial configuration complexity [17][27]. As Kubernetes adoption continues to expand, ensuring correct and secure configurations has become a critical challenge.

The consequences of misconfiguration are both tangible and costly. Industry reports document severe financial impacts: one SaaS company incurred over $100,000 in monthly cloud expenses due to an autoscaling misconfiguration that triggered uncontrolled resource consumption [26]. The Cloud Native Computing Foundation estimates that remediating a misconfiguration in production can cost up to $15,900 per workload—over 600 times the cost of addressing the same issue during development ($25) [7]. Even seemingly minor errors, such as misaligned resource quotas or overly permissive access controls, can propagate across clusters and systematically degrade system reliability [24][5].

Despite configuration analysis tools, misconfigurations remain widespread [24][30]. This persistence largely stems from Kubernetes' continuous evolution: each release introduces new APIs, deprecates legacy features, and expands the configuration surface [28]. These changes require administrators to continually adapt clusters and deployment strategies to maintain compatibility, creating a management burden of interdependent parameters where subtle inconsistencies can cascade into critical failures.

Current solutions predominantly employ rule-based detection mechanisms that encode expert knowledge into predefined policies. While effective at identifying known anti-patterns, these tools exhibit limited interoperability—rules and labels are often platform-specific, hindering systematic reasoning across environments. More fundamentally, most approaches emphasize detection over remediation, delegating corrective actions to human operators. In dynamic environments characterized by continuous integration and deployment, this manual, iterative correction process becomes both time-consuming and error-prone.

Addressing these limitations requires automated mechanisms capable of context-aware correction, not merely detection. Recent

advances in machine learning, particularly large language models (LLMs), offer new capabilities for reasoning over configuration artifacts and generating specification-compliant repairs. In this work, we present a systematic investigation of Kubernetes security misconfigurations, spanning empirical characterization to automated correction. We address the following research questions:

- **RQ1:** How can empirical evidence from Stack Overflow inform a taxonomy of Kubernetes security misconfigurations?
- **RQ2:** How does security misconfiguration severity vary across object types and categories?
- **RQ3:** How do security misconfigurations evolve between incubator and stable project stages?
- **RQ4:** How effectively can LLMs correct Kubernetes security misconfigurations, and how does **Kubecurity** enhance their reliability?

Together, these questions aim to advance both structured understanding and reliable, automated correction of Kubernetes security misconfigurations.

2 Related work

Kubernetes misconfiguration detection has been extensively studied through empirical analysis and specialized tooling. Rahman et al. [24] examined 2,039 manifests across 92 repositories, identifying 1,051 misconfigurations in eleven categories, with 94% of practitioners reporting production incidents from such errors. Traditional detection employs rule-based static analysis tools including Datree [8], KubeScore [1], Snyk [29], and KubeLinter [2] for identifying common anti-patterns through predefined policies.

Recent work has advanced toward data-driven approaches. LLM-powered systems including Malul [19], Cohen [25], and GenKubeSec [20] demonstrate improved precision through semantic reasoning capabilities. KubeGuard [25] jointly analyzes manifests and runtime logs for privilege escalation detection, illustrating how AI-assisted systems achieve deeper contextual understanding.

Despite progress in detection, critical gaps persist in taxonomy coverage, severity assessment, and automated remediation. Existing taxonomies like SLI-KUBE [23] and Rahman et al. [24] provide classification frameworks but cover only a subset of misconfigurations detected by current tools, lacking comprehensive coverage. Furthermore, severity assessment remains particularly underexplored. Existing tools assign fixed severity levels to individual rules, which fails to reflect actual risk since violations of the same rule can have different severity depending on the affected object and deployment context. Additionally, most detection tools delegate remediation entirely to practitioners. While Polaris [9] offers automated fix suggestions through predefined correction rules, rigorous empirical validation of AI-powered correction frameworks regarding accuracy and schema compliance remains absent.

3 Methodology

This section outlines the methodology for the four research questions, integrating empirical data collection, mixed analyses, and automated correction evaluation.

3.1 RQ1 – Constructing a Taxonomy of Kubernetes Misconfigurations

This research question sought to construct a comprehensive taxonomy of Kubernetes misconfigurations grounded in real-world practitioner discussions. Stack Overflow served as the primary data source, providing a structured corpus of question-answer exchanges that capture configuration issues encountered in practice [14].

3.1.1 Data Collection Data collection followed a structured, iterative process led by two practitioners to ensure coverage and relevance. Using the Stack Exchange API, we extracted all Stack Overflow tags, which were jointly reviewed to identify those related to Kubernetes. The refined subset was then used to filter Kubernetes-specific posts while minimizing cross-technology noise. Practitioners also curated configuration and security-related keywords. Using these tags and keywords, we queried Stack Overflow's archive (2015–2024), retaining posts with at least one Kubernetes tag and one keyword, yielding 681 posts (Set A). Because many developers omit tags, a keyword-only search produced 4,287 additional posts. To isolate genuine misconfiguration discussions, a supervised Support Vector Machine (SVM)[13] classifier trained on Set A distinguished between general and misconfiguration posts, yielding 1,981 curated posts (Set B). Sets A and B formed the empirical foundation for taxonomy construction.

3.1.2 Taxonomy Construction To identify recurring themes and latent structures, we employed hierarchical BERTopic [12], a transformer-based topic modeling framework leveraging contextual embeddings from BERT to extract both coarse and fine-grained topics. We used the `paraphrase-MiniLM-L3-v2` model from Sentence Transformers for its balance between semantic expressiveness and computational efficiency.

To determine the optimal granularity, we tested several `min_topic_size` configurations (2–10) and computed intra- and inter-topic distances to assess clustering quality. Lower intra-topic distances indicate higher semantic cohesion, while higher inter-topic distances reflect better distinctiveness. A minimum topic size of five produced the most coherent and well-separated topics.

We then applied BERTopic's hierarchical reduction procedure via `hierarchical_topics()`, which recursively merges semantically similar clusters based on cosine similarity of their c-TF-IDF representations. This produced a multi-level hierarchy capturing both general themes and specialized subtopics.

The hierarchical structure was manually refined through iterative labeling and consolidation. Two practitioners independently assigned descriptive titles to parent topics and subtopics during the initial phase. Multiple review sessions were held to reconcile naming inconsistencies and resolve overlaps by consensus. This human-in-the-loop process ensured that the resulting taxonomy preserved both data-driven organization and contextual fidelity to Kubernetes configuration practices.

3.2 RQ2 – Analyzing Misconfiguration Severity Across Object Types and Categories

While taxonomy-based classification helps characterize Kubernetes misconfigurations, their severity cannot be inferred solely from type. Because each object plays a distinct operational role within the

cluster, this study examines how misconfiguration severity varies across object types and taxonomy categories.

3.2.1 Dataset Construction We collected 10,000 Kubernetes configuration files from public GitHub repositories spanning 2015–2024. For each year, we queried the GitHub API using a set of keywords composed of the tags and keywords identified in RQ1, and iteratively collected files until reaching 1,000 parsable YAML configurations returned by the API, without applying any popularity or star-based filtering. Only files that were syntactically correct and successfully validated against the Kubernetes schema using Kubeval were retained. Each validated configuration was then analyzed using the three state-of-the-art Kubernetes misconfiguration detection tools identified in prior comparative research [11]. These tools were selected for their high detection accuracy and broad rule coverage, ensuring comprehensive and reliable identification of misconfigurations across diverse Kubernetes objects.

3.2.2 Mapping and Quantitative Analysis To ensure conceptual coherence, all detected misconfigurations were mapped to the categories established by the taxonomy defined in RQ1. This mapping was guided by a manually constructed correspondence table aligning the category definitions used by the detection tools with those of our taxonomy. Subsequently, we constructed a two-dimensional analytical matrix representing the intersection between Kubernetes object types and taxonomy categories. For each cell in this matrix, we aggregated both the frequency and the severity of detected misconfigurations, enabling a structured comparison of configuration risks across object types and misconfiguration classes.

Severity levels were harmonized across tools and standardized into three categories *Low*, *Medium*, and *High* following established risk assessment frameworks such as NIST SP 800-30 [15], OWASP Risk Rating [22], and CVSS v3.1 [10]. Although each detection tool employs its own native scale (*Datree*: Low–Medium–High–Critical; *Snyk* and *kube-score*: Low–Medium–High), we adopted a normalization process to ensure conceptual consistency. In this process, Datree's *Critical* category was merged with the *High* level to establish a unified three-tier severity scale. Finally, a majority-based rule was applied to each intersection between Kubernetes object type and misconfiguration category: if a single level represented more than 50% of the detections within that intersection, it was designated as dominant; otherwise, the two most frequent levels were retained.

3.3 RQ3 – Evolution of Misconfigurations Across Project Maturity Levels

The third research question explores the evolution of Kubernetes configurations to elucidate how misconfigurations arise, persist, and are ultimately resolved as software projects mature. This investigation aims to determine whether configuration issues are effectively mitigated over time and if new ones continue to emerge, even within versions deemed stable.

3.3.1 Data Selection For this study, we focused on Kubernetes configuration data drawn from the official Helm chart repository. We selected this repository because it provides access to the configuration files of each project in both its incubator and stable phases. This characteristic makes it particularly suitable for analyzing the evolution of Kubernetes configurations as projects transition from early development to production maturity.

To ensure comparability, we included only projects that exist in both the incubator and stable phases and that preserve a consistent architectural structure between these two versions. This design constraint allowed us to isolate the impact of configuration evolution from architectural or structural differences, ensuring that any observed misconfiguration changes are attributable to configuration evolution rather than project redesign.

Following this selection process, 73 projects out of an initial pool of 282 were retained for analysis, representing a total of 664 Kubernetes configuration files. These files encompass a variety of application domains, offering a diverse and representative dataset for assessing how Kubernetes configurations evolve throughout the Helm chart lifecycle.

3.3.2 Misconfiguration Analysis To achieve a rigorous and comprehensive assessment of Kubernetes misconfigurations across both incubator and stable Helm charts, this study employed three well-established detection tools widely adopted in the Kubernetes ecosystem: Snyk, Datree, and Kube Score. These tools were selected for their complementary analysis capabilities [11].

To ensure consistency and avoid redundancy, identical misconfigurations reported by multiple tools were merged and counted once. When tools identified different categories in the same file, each unique detection was retained. If the same issue appeared with varying severity levels, the highest severity was kept to reflect a conservative assessment.

By integrating results from multiple detection tools through a unified consolidation process, this study provides a comprehensive view of misconfigurations across the Helm chart lifecycle. The analysis reveals how misconfigurations emerge, persist, or are corrected between incubator and stable phases, highlighting the Kubernetes objects most susceptible to recurrent or unresolved issues.

3.4 RQ4 – Evaluating the Capability of LLMs to Correct Misconfigurations

This research question seeks to assess the ability of Large Language Models (LLMs) to rectify Kubernetes misconfigurations through two principal experiments: first, by leveraging various types of contextual information; and second, by employing contextual information augmented through schema-guided correction mechanisms.

3.4.1 Prompt-Based Experimental Design To evaluate the corrective capabilities of LLMs, we conducted experiments analyzing how different levels of contextual enrichment affect model performance.

Misconfiguration-only : The LLM receives only the faulty YAML file without additional context. This configuration evaluates the model's intrinsic capacity to identify and correct misconfigurations purely from syntactic and structural patterns, representing a zero-shot reasoning scenario.

Type-augmented : The input combines the misconfigured file with the taxonomy-defined misconfiguration type (e.g., TLS not enforced or Missing NetworkPolicy). This minimal semantic guidance directs the model's reasoning toward a specific conceptual domain, improving the precision of repair suggestions.

Type with detailed Description : In this configuration, the prompt was further enriched with a detailed natural-language description of the misconfiguration (e.g., "The container lacks a liveness probe, which prevents Kubernetes from detecting and restarting unresponsive containers.").

For each experiment, the model generated a corrected configuration candidate. The resulting outputs were automatically validated against the official Kubernetes JSON schemas to ensure syntactic correctness and structural integrity. They were then reassessed using the detection tools employed in RQ2 and RQ3 namely Datree, Snyk, and KubeScore to verify the absence of any misconfigurations. A correction was considered successful only if it satisfied both schema validation and tool-based verification.

3.4.2 Schema-Guided Rule-Based Correction with Kubecurity As a complementary component to the LLM-based correction process, we developed Kubecurity, a schema-driven, rule-based correction engine designed to enforce deterministic compliance with official Kubernetes specifications. Each Kubernetes object type is formally defined by a JSON Schema that specifies its structure, required attributes, permitted data types, and contextual constraints. Kubecurity leverages this formal specification to automatically detect and repair configuration inconsistencies. By reasoning over the configuration tree, Kubecurity reconstructs the hierarchy, aligning each element with its parent and ensuring proper field ordering and indentation. Missing required fields are supplemented with default schema values, and type conflicts are reconciled by converting invalid entries into compliant forms. Invalid or obsolete sections are removed, and the configuration is revalidated against the reference schema for full compliance. Beyond correction, Kubecurity acts as a safeguard for LLM-based outputs. While LLMs provide context-aware fixes, their results may violate Kubernetes syntax or structure. Kubecurity runs before and after the LLM stage: initially to repair deterministically resolvable issues, and later to revalidate and fix residual inconsistencies. This dual validation ensures all configurations remain compliant with Kubernetes standards.

4 Results

In this section, we present and discuss our findings for each research question.

4.1 RQ1 – Taxonomy Presentation

To address the first research question, we implemented a multi-stage process encompassing data collection, topic modeling, and iterative refinement, as described in the methodology section.

The final taxonomy comprises five major categories and twenty-one subcategories, each capturing a distinct dimension of how configuration faults manifest in cloud-native systems. Together, these categories form a structured framework for understanding, diagnosing, and remediating configuration weaknesses across the Kubernetes ecosystem. Figure 1 provides a comprehensive overview of the proposed taxonomy.

Across the 2,613 relevant Stack Overflow posts analyzed, the distribution of misconfigurations was as follows: Access and Privileges (23.8%), Resource Management and Probes (21.5%), Image and Network Security (19.0%), Encryption and Permission Management (20.3%), and Filesystem Configuration (15.4%).

Access and Privileges: Emerged as the most frequently discussed category, this category includes misconfigurations related to authorization, privilege escalation, and RBAC policy management. When improperly configured, these issues can enable users or workloads to perform unauthorized operations, resulting in sensitive data exposure or cluster compromise.

Snippet 2: allowPrivilegeEscalation

```
    securityContext:
      allowPrivilegeEscalation: true  # Should
          be false
```

Here, `allowPrivilegeEscalation: true` enables the container to execute privileged system calls. If exploited, it may grant root access to the underlying host. Following the principle of least privilege, this field should be explicitly set to `false` and combined with restrictive RBAC policies to limit system exposure.

Resource Management and Probes: This category covers inefficiencies and risks caused by improper resource allocation, scheduling, or health monitoring configurations. Missing resource requests and limits may destabilize multi-tenant clusters, resulting in workload starvation or node overcommitment.

Snippet 3: Missing resources.requests or resources.limits

```
spec:
  containers:
  - name: unbounded-container
    image: myapp:stable
    resources: {}  # Missing requests and limits
```

This configuration omits both `resources.requests` and `resources.limits`, granting unrestricted resource consumption. Without these controls, a single container can monopolize CPU or memory, degrading the performance of co-located workloads. Defining appropriate resource boundaries prevents this issue and ensures predictable scheduling.

Image and Network Security: This category focuses on vulnerabilities linked to image provenance, network exposure, and security of inter-component communications. Misconfigurations here often facilitate the deployment of unverified images or expose services to public networks without encryption.

Snippet 4: Use of latest tag

```
spec:
  containers:
  - name: app
    image: myapp:latest  # Using 'latest' tag
        introduces inconsistency
```

Using the generic `latest` tag leads to unpredictable deployments, as different nodes may pull different image versions. This inconsistency can reintroduce known vulnerabilities or regressions. To ensure stability and traceability, images should be version-pinned and verified through trusted registries.

Encryption and Permission Management: This category encompasses misconfigurations in data protection, secrets handling, and transport-layer security. Weak encryption or plaintext secret storage undermines confidentiality and exposes sensitive credentials.

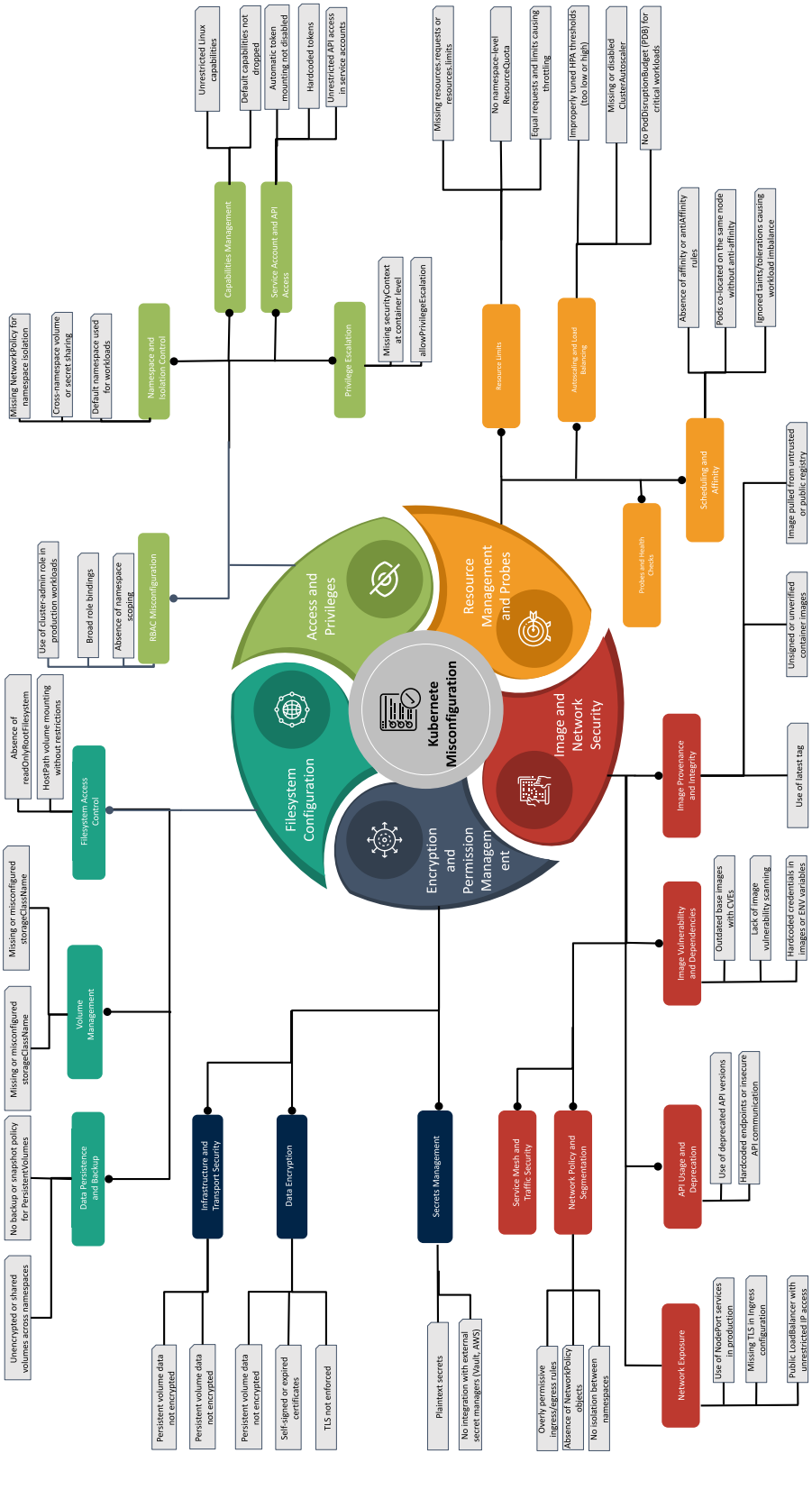

Figure 1: Kubernetes Misconfiguration Taxonomy

Snippet 5: Plaintext secrets

```
data:
  password: myplaintextpassword  # Should be
      encrypted
```

In this example, the secret value is stored as plaintext rather than encoded or encrypted, making it visible to anyone with read access to the manifest. All sensitive data should be base64-encoded or managed through external secret management systems (e.g., HashiCorp Vault, AWS Secrets Manager) to ensure confidentiality and compliance.

Filesystem Configuration: Representing the least frequent category, this class encompasses misconfigurations affecting storage and data persistence, including filesystem access control, volume management, and backup policies. Such errors can lead to unauthorized host access, data loss, or insecure persistence across namespaces. Improper filesystem configuration is among the most severe misconfiguration types, as it can directly compromise system integrity.

Snippet 1: Absence of readOnlyRootFilesystem

```
    securityContext:
      readOnlyRootFilesystem: false  # Should be
          true for safety
```

In this example, setting `readOnlyRootFilesystem: false` allows write access to the container's root filesystem. This increases the attack surface by permitting file modifications and potential privilege escalation through tampered binaries. To mitigate this risk, it is recommended to set `readOnlyRootFilesystem: true`, ensuring a secure, immutable filesystem environment.

4.2 RQ2 – Misconfiguration Severity Analysis

This section presents the results of the second research question, examining how the severity of Kubernetes misconfigurations varies across object types and taxonomy categories.

Across the 10,000 validated configuration files collected between 2015 and 2024, the three detection tools jointly reported 23,736 misconfigurations. Because several tools often identified the same issues, duplicates were consolidated to preserve analytical integrity. After deduplication in which each unique misconfiguration was counted once regardless of how many tools detected it,the final set contained 14,375 distinct misconfigurations.

Following the normalization process described in the methodology, all severity levels were harmonized into a three-tier classification (Low, Medium, High). The consolidated results indicate that 60.19% of the misconfigurations were Low, 38.27% were Medium, and 1.54% were High severity. This distribution suggests that most configuration errors affect reliability or maintainability, while nearly a quarter represent severe risks to security or cluster stability. This distribution indicates that the majority misconfigurations have limited operational impact, while a very small subset poses serious risks to cluster security or stability. Despite their low frequency, high-severity misconfigurations remain critical, as they often involve privilege mismanagement, insecure networking, or the exposure of sensitive information. Table 1 summarizes the empirical severity patterns across Kubernetes object types and taxonomy categories.

Building on this distribution, a deeper examination across Kubernetes object types and taxonomy categories reveals several empirical observations and severity patterns.

Pods and Deployments : exhibited the highest proportion of Medium to High-severity misconfigurations, predominantly within *Access and Privileges* and *Image and Network Security*. Common patterns included privileged containers, unsafe `hostPath` mounts, and unsecured image pull configurations. Although these High-severity cases account for only a small share of the total corpus, their operational implications ranging from container escape to unverified image execution make them disproportionately impactful.

Services and Ingresses : presented a mixed severity profile, dominated by Low and Medium levels within *Image and Network Security*. Typical misconfigurations involved missing TLS termination, open network ports, and unrestricted external exposure. Despite being less severe on average, these configurations increase the external attack surface, demonstrating that even moderate misconfigurations can have compounding security effects when applied to exposed components.

PersistentVolumes and **PersistentVolumeClaims :** were largely affected by *FileSystem Configuration* issues, mainly of Low and Medium severity. The majority of errors involved incorrect access modes or misaligned persistence settings. In a limited number of cases ($\approx$ 2%), High-severity faults were identified, typically when host-level paths were insecurely mounted—posing risks of unauthorized data access.

ServiceAccounts, Roles, and RoleBindings : formed the most critical cluster of objects in terms of severity distribution. More than half of their misconfigurations were classified as Medium or High, concentrated in *Access and Privileges* and *Encryption and Permission Management*. These reflect deep structural vulnerabilities, such as over-permissive role definitions, missing namespace scoping, or unbounded credential exposure misconfigurations capable of compromising the cluster's security perimeter.

ConfigMaps and Secrets: showed a predominance of Low and Medium severities under *Encryption and Permission Management*. Frequent issues included plaintext storage of sensitive credentials and incorrect use of ConfigMaps to hold confidential data. Although High-severity cases were rare, they correspond to direct violations of encryption policies and thus carry significant potential impact.

Jobs, CronJobs, and HorizontalPodAutoscalers : were primarily affected by *Resource Management and Probes* issues, almost entirely Low or Medium in severity. These misconfigurations typically involved missing readiness or liveness probes, suboptimal scaling parameters, or loosely defined restart policies. While not directly security-critical, their recurrence indicates systemic weaknesses in reliability-oriented configuration practices.

The two-dimensional analysis reveals that misconfiguration severity in Kubernetes is inherently contextual, relational, and non-uniform. Severity does not arise solely from the intrinsic nature of a misconfiguration, but rather from its interaction with the operational semantics of the affected object, such as responsibilities, privilege boundaries, and exposure level. A privilege escalation that remains benign within a confined namespace may become catastrophic when applied to a cluster-wide role. Likewise, an omitted

Table 1: Empirical Assessment of Kubernetes Misconfigurations Across Object Types and Taxonomy Categories

Kubernetes Object Type	FileSystem Configuration	Access & Privileges	Resource Mgmt. & Probes	Image & Network Security	Encryption & Permission Mgmt
Pod	Low / Medium	Medium / High	Medium	Medium / High	Low
Deployment	Low / Medium	Medium / High	Medium	Medium / High	Low
Service	Low	Medium	Medium	Medium	Low
Ingress	–	Medium	Medium	Medium	Low
ConfigMap	Low / Medium	–	–	Medium	Medium
Secret	–	–	–	–	Medium / High
PersistentVolumeClaim (PVC)	Medium	Medium	Low / Medium	Low	Low
PersistentVolume (PV)	Medium	Medium	Low / Medium	Low	Low
ServiceAccount	Medium / High	Medium / High	Medium	Medium	Medium / High
Role / ClusterRole	Medium	Medium / High	Medium	Medium	Medium / High
RoleBinding / ClusterRoleBinding	Medium	Medium / High	Medium	Medium	Medium / High
Job	Low	–	Medium	Low	Low
CronJob	Low	–	Medium	Low	Low
HorizontalPodAutoscaler (HPA)	Low	–	Medium	Low	Low

probe that merely reduces scaling efficiency in a HorizontalPodAutoscaler (HPA) can trigger service unavailability when it affects a Pod. These observations empirically validate the necessity of an intersectional severity analysis, in which risk is characterized through the joint consideration of object type and misconfiguration category, rather than through a flat or category-only assessment.

4.3 RQ3 – Misconfiguration Evolution Across Maturity Levels

To examine how Kubernetes misconfigurations evolve with project maturity, we analyzed the transition from incubator to stable Helm chart versions. The results reveal heterogeneous correction patterns across taxonomy categories.

Operational issues received the greatest attention. *Resource Management and Probes* exhibited the highest correction rate (55%), followed by *FileSystem Configuration* (42.86%), indicating that availability and performance concerns are prioritized as projects progress toward stability.

In contrast, security-related categories showed significantly lower remediation rates. *Access and Privileges* achieved a correction rate of only 16.28%, while *Encryption and Permission Management* reached 17.22%. Similarly, *Image and Network Security* recorded a correction rate of 24.14%.

Overall, out of 336 misconfigurations detected in incubator versions, only 72 were corrected in stable releases, corresponding to a global correction rate of 21.43%. Additionally, 21 new misconfigurations (6.25%) emerged in stable versions, demonstrating that configuration evolution is neither linear nor purely corrective.

A closer inspection of unresolved cases reveals that the majority of persistent issues lie within security-critical domains. More than 80% of misconfigurations in *Encryption and Permission Management* and *Access and Privileges* remained uncorrected. These findings suggest that while developers actively prioritize operational stability and performance, security refinements are often deprioritized.

Overall, Kubernetes configuration evolution appears dynamic and non-linear, marked by both progress and regression. The persistence of security weaknesses highlights the limitations of manual remediation and motivates the investigation of automated correction mechanisms based on Large Language Models and schema-guided validation.

[!t]

4.4 RQ4 – LLMs Capability in Correcting Misconfigurations

This section investigates the corrective capabilities of five LLMs in repairing Kubernetes misconfigurations. The models Mistral-7B [33], GPT-3.5-Turbo [21], LLaMA 3.1-70B [35], DeepSeek R1 [32], and Qwen 2.5-7B [36] were chosen to capture architectural, scale, and reasoning diversity across both proprietary and open-weight paradigms. These models were selected to represent diverse architectural paradigms and training strategies. GPT-3.5-Turbo served as a strong baseline due to its consistent performance in structured reasoning and prompt adherence [3]. LLaMA 3.1-70B, an open-weight model, enabled the assessment of transparent architectures and their capacity to leverage contextual cues in configuration repair [37]. Mistral-7B and Qwen 2.5-7B, both lightweight yet high-performing models, provided insights into the trade-off between parameter efficiency and corrective accuracy [34][6]. Finally, DeepSeek R1 introduced an explicit focus on logic-driven reasoning and structured task understanding [31], particularly well aligned with the declarative nature of Kubernetes configurations.

The evaluation was performed on a corpus of 10,000 Kubernetes configuration files extracted from Helm charts and deployment domains. After normalization and deduplication, 13,495 distinct misconfigurations were identified across five taxonomy categories.

4.4.1 ***Prompt-Based Correction*** The experiments assessed how progressively richer contextual information influences LLMs' ability to correct misconfigurations. In Experiment 1, models received only the faulty configuration file. Experiment 2 added the misconfiguration type to provide structural guidance. Experiment 3 further

Table 2: Performance Comparison Across Experiments

Model	Experiment 01				Experiment 02				Experiment 03				Experiment 04			
	Not Parsed	Not Corrected	Corrected	Emerged	Not Parsed	Not Corrected	Corrected	Emerged	Not Parsed	Not Corrected	Corrected	Emerged	Not Parsed	Not Corrected	Corrected	Emerged
Mistral-7B	59.24%	25.32%	15.71%	359	1.21%	12.82%	86.24%	507	1.15%	12.00%	87.12%	359	1.13%	2.10%	97.04%	46
GPT-3.5-Turbo	61.01%	24.86%	14.40%	217	1.46%	11.21%	**87.60%**	306	1.32%	9.89%	**89.06%**	217	0.80%	1.94%	97.53%	21
LLaMA 3.1-70B	51.52%	41.10%	**7.65%**	310	5.04%	11.56%	83.66%	224	4.56%	9.56%	86.14%	310	0.48%	1.77%	98.02%	43
DeepSeek R1	33.27%	28.62%	**38.38%**	409	2.11%	10.62%	87.54%	514	2.62%	8.91%	88.74%	409	0.32%	1.45%	**98.50%**	91
Qwen 2.5-7B	53.95%	31.56%	14.76%	286	4.47%	14.44%	**81.37%**	308	3.76%	14.27%	**82.24%**	286	1.44%	4.53%	**93.98%**	127

included a brief natural-language explanation of the issue, aligning with Kubernetes best practices. This stepwise design enabled a systematic analysis of how increasing context affects reasoning quality, correction accuracy, and the introduction of new misconfigurations. The comparative results are presented in Table 2.

The results summarized in Table 2 shows a clear trend: increasing contextual information consistently improves correction accuracy and reasoning stability across all models. In Experiment 1, where only the faulty configuration was provided, performance was limited due to the absence of structural and semantic cues, leading to high parsing failure rates (over 50%) and structurally invalid fixes. DeepSeek R1 achieved the highest correction rate (38.38%), while LLaMA 3.1-70B performed poorly (7.65%), indicating that model size alone does not ensure structural understanding.

In Experiment 2, adding the misconfiguration type significantly improved results, with all models surpassing 80% accuracy and parsing errors nearly disappearing. GPT-3.5-Turbo (87.6%) and DeepSeek R1 (87.54%) achieved comparable performance, demonstrating that even minimal semantic guidance stabilizes reasoning and reduces irrelevant or hallucinated corrections.

Experiment 3 further enhanced performance by including a descriptive explanation of the issue and its risks. Correction rates exceeded 86%, with GPT-3.5-Turbo and DeepSeek R1 approaching 89%. The richer context enabled more precise, risk-aware fixes and significantly reduced the introduction of new misconfigurations, confirming that descriptive context mitigates overcorrection and unnecessary edits.

4.4.2 ***Schema-Guided Rule-Based Correction with Kubecurity*** Experiment 4 assessed the impact of integrating Kubecurity, a schema-guided validation mechanism, into the correction pipeline. Compared to Experiment 3, this integration significantly improved performance across all models: average correction accuracy rose from 87% to over 97%, parsing failures fell below 1.5%, and newly introduced misconfigurations decreased by more than 70%.

The improvement stemmed from Kubecurity's ability to systematically resolve structural inconsistencies such as indentation errors, missing required fields, and type mismatches by enforcing official Kubernetes JSON Schemas. This deterministic validation layer converted near-correct but invalid outputs into fully compliant configurations. The effect was particularly notable for smaller models like Mistral-7B and Qwen 2.5-7B, which, once augmented with Kubecurity, achieved performance comparable to larger models such as DeepSeek R1 and GPT-3.5-Turbo, all exceeding 97.5% correction accuracy.

Overall, combining LLM contextual reasoning with schema-based validation proved highly effective, yielding configurations that were syntactically valid, semantically consistent, and fully compliant with Kubernetes standards.

5 Implications

This study highlights important implications for Kubernetes configuration management. For practitioners, the persistence of misconfigurations demonstrates the limits of manual correction and underscores the need for automated pipelines that combine LLM-based reasoning with schema validation, such as Kubecurity, to enhance reliability and security. Continuous validation and automated remediation should be embedded into deployment workflows.

For tool developers, the findings emphasize the importance of standardizing severity definitions and integrating schema-guided correction mechanisms to enable consistent, interoperable, and self-healing systems.

For researchers, the results call for shared benchmarks, labeled datasets, and systematic evaluations to advance automated correction methods. Hybrid approaches that combine LLM reasoning with formal validation appear particularly promising for strengthening cloud-native system reliability.

Overall, the study advocates a continuous, validation-driven approach to configuration management to reduce human error and improve Kubernetes stability.

6 Threats to Validity

The evaluation was conducted on a finite dataset of Kubernetes configurations and a selected set of detection tools and language models. Although the dataset contains diverse real-world cases, it may not fully represent the entire spectrum of possible misconfigurations encountered in practice. Similarly, the study considers a limited number of models and analysis tools, which may influence the observed performance trends. Future work may extend the experiments to larger datasets, additional models, and a broader set of detection tools to further assess generalizability.

7 conclusion

This research paper introduces a comprehensive taxonomy of Kubernetes misconfigurations and analyzes their evolution across Helm chart development phases, showing that while operational issues often decrease, critical security misconfigurations may persist. The study further evaluates LLM-based automated correction and proposes Kubecurity, a schema-guided validation framework that reinforces structural and semantic accuracy. The hybrid approach achieves high correction performance and full Kubernetes compliance, significantly reducing persistent and newly introduced misconfigurations. Overall, the findings provide an empirical foundation for understanding configuration reliability in cloud-native systems and support future research on automated correction and resilience mechanisms in Kubernetes environments.

References

[1] 2024. kube-score: Kubernetes object analysis with recommendations for improved reliability and security. https://kube-score.com/. Accessed: 2024-10-07.

[2] 2024. KubeLinter Documentation. https://docs.kubelinter.io/#/. Accessed: 2024-10-07.

[3] A. Al Zubaer et al. 2023. Performance analysis of large language models in the legal domain: argument mining with GPT-3.5 and GPT-4. *Frontiers in Artificial Intelligence* 6 (2023), 1278796. doi:10.3389/frai.2023.1278796 This study shows GPT-3.5's ability to follow prompts and perform structured classification tasks, albeit with limitations..

[4] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016), 50–57. doi:10.1145/2890784

[5] D. Choi, H. Seo, K. Kim, M. You, S. Shin, and J. Kim. 2024. Uncovering Threats in Container Systems: A Study on Misconfigured Container Components in the Wild. *IEEE Transactions on Dependable and Secure Computing* (2024). doi:10.1109/TDSC.2024.10788674 Early Access.

[6] Qwen LM Team / Alibaba Cloud. 2024. *Qwen 2.5: A Comprehensive Series of Large Language Models (including the 7B variant).* Technical Report arXiv:2412.15115. Qwen LM Team. https://arxiv.org/abs/2412.15115 Describes the 7B model variant in the Qwen 2.5 series and emphasises efficiency improvements and broad capability scaling..

[7] Cloud Native Computing Foundation (CNCF) and Fairwinds. 2022. *The Cost of a Kubernetes Repair in Development vs Production.* https://www.cncf.io/blog/2022/02/02/the-cost-of-a-kubernetes-repair-in-development-vs-production/ Accessed: 2025-10-24.

[8] Datree. 2024. Datree: Prevent Kubernetes Misconfigurations. https://datree.io/ Accessed: 2024-10-07.

[9] Fairwinds. 2024. Polaris: Open Source Policy Engine for Kubernetes. https://www.fairwinds.com/polaris Accessed: 2024-10-07.

[10] Forum of Incident Response and Security Teams (FIRST). 2019. *Common Vulnerability Scoring System v3.1: Specification Document.* Technical Report. FIRST.Org, Inc. https://www.first.org/cvss/v3-1/specification-document Version 3.1, available at https://www.first.org/cvss/v3-1/specification-document.

[11] Mostafa Anouar Ghorab and Mohamed Aymen Saied. 2025. Towards Secure Cloud-Native Computing: Unveiling Kubernetes Misconfigurations with Large Language Models. In *2025 IEEE 18th International Conference on Cloud Computing (CLOUD).* IEEE, 86–96.

[12] Maarten Grootendorst. 2022. BERTopic: Neural topic modeling with a class-based TF-IDF procedure. *arXiv preprint arXiv:2203.05794* (2022).

[13] M.A. Hearst, S.T. Dumais, E. Osuna, J. Platt, and B. Scholkopf. 1998. Support vector machines. *IEEE Intelligent Systems and their Applications* 13, 4 (1998), 18–28. doi:10.1109/5254.708428

[14] Run Huang and Souti Chattopadhyay. 2024. A Tale of Two Communities: Exploring Academic References on Stack Overflow. In *Companion Proceedings of the ACM Web Conference 2024 (WWW '24 Companion).* Association for Computing Machinery, Singapore, 1201–1210. doi:10.1145/3589335.3651464

[15] Joint Task Force Transformation Initiative. 2012. *Guide for Conducting Risk Assessments (NIST Special Publication 800-30 Rev. 1).* Technical Report NIST Special Publication 800-30 Rev. 1. National Institute of Standards and Technology, Gaithersburg, MD. doi:10.6028/NIST.SP.800-30r1

[16] Michael Kavis. 2014. *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS).* John Wiley & Sons.

[17] H. H. Khan, S. Zubair, F. Nasim, and S. Akhter. 2024. Role of Kubernetes in DevOps Technology for the Effective Software Product Management. *Journal of Computing and Business Informatics* (2024). https://jcbi.org/index.php/Main/article/view/471

[18] Nane Kratzke and Peer-Christian Quint. 2017. Understanding cloud-native applications after 10 years of cloud computing – A systematic mapping study. *Journal of Systems and Software* 126 (2017), 1–16. doi:10.1016/j.jss.2017.01.001

[19] E. Malul, Y. Meidan, D. Mimran, and Y. Elovici. 2024. GenKubeSec: LLM-Based Kubernetes Misconfiguration Detection, Localization, Reasoning, and Remediation. *arXiv preprint arXiv:2405.19954* (2024). https://arxiv.org/pdf/2405.19954

[20] Ehud Malul, Yair Meidan, Dudu Mimran, Yuval Elovici, and Asaf Shabtai. 2024. GenKubeSec: LLM-Based Kubernetes Misconfiguration Detection, Localization, Reasoning, and Remediation. *arXiv preprint arXiv:2405.19954* (2024).

[21] OpenAI. 2023. GPT-3.5 Turbo Model Documentation. https://platform.openai.com/docs/models/gpt-3.5-turbo. Proprietary model; no full academic paper publicly released..

[22] OWASP Foundation. 2021. OWASP Risk Rating Methodology. https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. Accessed: 2025-10-24.

[23] Akond Rahman. 2024. sli-kube. https://hub.docker.com/r/akondrahman/sli-kube. Accessed: 2024-10-08.

[24] A. Rahman, S. I. Shamim, and D. B. Bose. 2023. Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study. In *Proceedings of the ACM on Software Engineering.* doi:10.1145/3579639

[25] O. Sgan Cohen, E. Malul, Y. Meidan, D. Mimran, Y. Elovici, and A. Shabtai. 2025. KubeGuard: LLM-Assisted Kubernetes Hardening via Configuration Files and Runtime Logs Analysis. *arXiv preprint arXiv:2509.04191* (2025).

[26] Shoham Shoham. 2024. *The Cost of Kubernetes: Which Workloads Waste the Most Resources.* https://dev.to/shohams/the-cost-of-kubernetes-which-workloads-waste-the-most-resources-2514 Dev.to, Accessed: 2025-10-24.

[27] R. Shrestha and A. A. N. Ali. 2024. Configuration Management in Kubernetes Environments: A GitOps Approach. In *IEEE Xplore.* https://ieeexplore.ieee.org/abstract/document/10971761/

[28] A. Singh. 2025. *Configuration Changes in Kubernetes Configuration Scripts.* Master's thesis. Auburn University. https://etd.auburn.edu/bitstream/handle/10415/9682/Ayush_Singh_Masters_Thesis.pdf?sequence=8&isAllowed=y

[29] Snyk. 2024. Snyk: Find and fix vulnerabilities in your code, open source dependencies, containers, and infrastructure as code. https://snyk.io/fr/ Accessed: 2024-10-07.

[30] Y. Sun, D. Lyu, C. Cui, and H. Xu. 2025. KubeChecker: Detecting Configuration Bugs in Container Orchestration. In *IEEE International Conference on Cloud Computing (CLOUD).* https://ieeexplore.ieee.org/abstract/document/11068388/

[31] DeepSeek AI Team. 2025. *DeepSeek-R1: Incentivizing Reasoning Capability in Large Language Models via Reinforcement Learning.* Technical Report arXiv:2501.12948. DeepSeek AI. https://arxiv.org/abs/2501.12948 Introduces a reasoning-focused LLM trained via multi-stage and RL methods, achieving competitive reasoning performance on structured tasks..

[32] DeepSeek AI Team. 2025. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning.* Technical Report arXiv:2501.12948. DeepSeek AI. https://arxiv.org/abs/2501.12948 Open-source reasoning model trained via large-scale RL, includes zero stage and distilled variants..

[33] Mistral AI Team. 2023. Mistral 7B v0.1: A 7-Billion-Parameter Language Model Engineered for Superior Performance and Efficiency. *arXiv preprint* arXiv:2310.06825 (2023). https://arxiv.org/abs/2310.06825 Released by Mistral AI; demonstrates grouped-query attention (GQA) and sliding-window attention (SWA)..

[34] Mistral AI Team. 2023. Mistral 7B v0.1: A 7-Billion-Parameter Language Model Engineered for Superior Performance and Efficiency. *arXiv preprint* arXiv:2310.06825 (2023). https://arxiv.org/abs/2310.06825 Presented grouped-query attention (GQA) and sliding-window attention (SWA) to get high performance despite only 7B parameters..

[35] Meta AI Team. 2024. *The LLaMA 3 Herd of Models.* Technical Report arXiv:2407.21783. Meta Platforms, Inc. https://arxiv.org/abs/2407.21783 Release includes 8B, 70B and 405B sizes; here the 70B version is cited..

[36] Qwen LM Team. 2025. *Qwen 2.5: A Comprehensive Series of Large Language Models.* Technical Report arXiv:2412.15115. Alibaba Cloud / Qwen LM. https://arXiv.org/abs/2412.15115 Includes the 7B model in the series; improved pre-training scale and downstream capabilities..

[37] Hugo Touvron, Louis Martin, et al. 2024. The LLaMA 3 Herd of Models. *arXiv preprint* arXiv:2407.21783 (2024). https://arxiv.org/abs/2407.21783 The paper presents the LLaMA 3 model family (8B, 70B, 405B) supporting multilinguality, reasoning, coding, and long-contexts..

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009