Configuration Changes in Kubernetes Configuration Scripts

by

Ayush Singh

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 10, 2025

Keywords: Kubernetes, Configuration Chnages

Copyright 2025 by Ayush Singh

Approved by

Dr. Akond Rahman, Assistant Professor of Computer Science & Software Engineering
Dr. Samuel Mulder, Associate Professor of Computer Science & Software Engineering
Dr. Ali Ghanbari, Assistant Professor of Computer Science & Software Engineering

Abstract

Kubernetes has emerged as the dominant container orchestration platform, widely adopted for managing cloud-native applications. However, its dynamic configuration introduces complexities that can lead to security vulnerabilities, system failures, and operational inefficiencies. Understanding the patterns and implications of these configuration changes is crucial for improving system reliability, maintainability, and security.

This thesis presents a large-scale empirical study of 28,675 Kubernetes-related commits to examine the nature of configuration changes. We categorize these changes into eight distinct types: Annotations, Corrective, Dependency Updates, Deployment Infrastructure, Documentation, Maintenance, Security and Compliance, and User Interface. We also analyze patterns in these categories by focusing on their frequency.

Our study highlights that dependency updates and deployment infrastructure related changes are among the most frequent modifications, reflecting the critical need for up-to-date configurations, while other categories such as corrective, security and compliance show meaningful patterns in their frequency.

Our classification sheds light on the nature of configuration-related changes in Kubernetes-based container orchestration by characterizing the evolution of Kubernetes configuration changes by understanding the common trends in Kubernetes configuration. These findings offer a stepping stone for future research and practical tooling around configuration changes and provides actionable insights for Kubernetes practitioners, DevOps engineers.

Acknowledgements

I express my gratitude to my father, Mr. Akhilesh, whose steadfast support has been the foundation of my academic career. I also want to express my sincere gratitude to my mother, Mrs. Nirmala Singh, for her unwavering belief in me and support. I am thankful to my siblings especially my brother, Shivam, who advised me to pursue career in computer science & engineeirng and my Sister, Aarju Singh for the support during the challenging time of my academic era, when I had just started my Masters degree. This accomplishment would not have been achieved without their affection and assistance. I also want to express my gratitude to my entire family for their support, encouragement, and prayers.

I am deeply grateful to Dr. Akond Rahman, my research advisor, for his constant encouragement and invaluable guidance throughout my time at Auburn University. His mentorship has been instrumental in shaping my research journey and ensuring the successful completion of my thesis. I would also like to extend my sincere thanks to my thesis committee members, Dr. Ghanbari and Dr. Mulder, for their insightful feedback and constructive advice. I am equally thankful to my academic advisor, Dr. Clint Lovelace, for his time, support, and thoughtful guidance on my research and academic pursuits. My heartfelt thanks go to my lab mates and colleagues, whose knowledge-sharing and collaborative spirit have enriched my learning experience. I am also immensely thankful to Dr. Narendra Kumar Govil, Professor Emeritus, Department of Mathematics & Statistics, whose wisdom and life experiences inspired me to dream big and pursue my academic goals with confidence and determination. His encouragement left a lasting impact on my outlook toward success and perseverance. Finally, I would like to express my deepest appreciation to my friends, Shiv Sahaya Shukla, Tithi Patel & Lalith Medury whose unwavering support has been a constant source of strength throughout my academic journey at Auburn University.

I sincerely thank the PASER research group at Auburn University for their collaboration and valuable feedback. Special appreciation goes to Yue Zhang, Anita Cheng, Arpan Srivastava, and Jahidul Arafat for their contributions and discussions. This research was partially supported by NSF Awards #2247141, #2310179, and #2312321.

I would also like to express my sincere gratitude to Dr. Xiao Qin and Dr. Hari Narayanan, the Chair of the Computer Science and Software Engineering Department at Auburn University, for their guidance and support. Additionally, I am deeply thankful to the Auburn University RFID Lab for their generous financial support throughout my Master's program. I would also like to extend my heartfelt thanks to the members of the Auburn University RFID Lab for providing me with a positive and collaborative environment to work in, as well as for giving me the opportunity to make lifelong friendships.

Furthermore, I am extremely grateful for my first internship at IIT Kanpur, where I had the privilege of working with employees who encouraged and boosted my confidence to pursue higher studies in the United States. We became good friends, and I cherish the experience. I would also like to thank Mr. Rahul Garg & Mrs Prachi Garg, the Director of Prutor, an organization affiliated with IIT Kanpur, for their valuable support and mentorship during my internship tenure.

Table of Contents

Abstract 2

List of Tables 7

List of Figures 8

1 Introduction **9**

1.1 Motivation 9

1.2 Problem & Goal statement 10

1.3 Overview of Methodology 10

1.4 Thesis Organization 11

1.5 Contributions of the Study 11

2 Background and Related Work **12**

2.1 Background 12

2.2 Motivation for Commit Analysis in Kubernetes Configuration 17

2.3 Prior Work 17

2.4 Summary 19

3 Methodology **20**

3.1 Why Frequency Matters 20

3.2 Mining OSS Repositories 20

3.3 Categorization of Commits (RQ1) 22

3.4 Frequency Analysis of Commit Categories (RQ2) 23

3.5 Summary 24

4 Results **25**

4.1 Categorization of Configuration Changes 25

4.2 Example Integration 27

4.3 Distribution of Configuration Change Per Categories 47

4.4 Distribution Of Contributor Counts Across Kubernetes Repositories 48

4.5 Summary of Key Findings 49

5 Discussion **50**

6 Threats to Validity **51**

6.1 Construct Validity 51

6.2 External Validity 51

6.3 Internal Validity 51

7 Conclusions **52**

Bibliography 53

List of Tables

1 Overview of Kubernetes Configuration Management Tools 19

2 Repository Filtering Process 21

3 Categories of Kubernetes Configuration Changes 26

List of Figures

1 Container orchestration workflow 13

2 Anatomy of a Kubernetes Configuration File 14

3 Overview of a Container 15

4 Kubernetes Architecture Diagram 16

5 Proportion of Commits per Category 47

6 Count of Kubernetes-related Repositories Based on Contributor Count 48

1 Introduction

1.1 Motivation

Kubernetes has emerged as the leading container orchestration platform, enabling organizations to deploy, manage, and scale containerized applications efficiently. Its ability to automate scaling, manage containerized workloads, and ensure high availability makes it a crucial component in modern infrastructure. Kubernetes is an open-source software for automating management of computerized services, such as containers [7]. With widespread adoption by enterprises, Kubernetes has become a key technology for managing microservices architectures, distributed computing environments, and hybrid cloud deployments. However, in practice, configuration-related changes are routinely made by developers, yet these changes are poorly understood and also introduces significant complexities in maintaining and updating configurations.

Despite Kubernetes widespread adoption, little empirical research exists on configuration changes. Prior research has extensively analyzed general software commits, but commits specifically related to Kubernetes configurations have not been systematically studied. A deeper understanding of the nature and frequency of configuration-related changes is essential to improving configuration practices and supporting tools for DevOps. To address this gap, our study conducts a large-scale empirical analysis of Kubernetes-related commits to categorize common configuration changes. Understanding the patterns of configuration changes is essential to improving automation and best practices in Kubernetes deployment management. This line of research opens new directions for future work in the area of automated repair of Kubernetes configurations, there is a compelling opportunity to collaborate on building intelligent repair mechanisms that can detect, classify, and autonomously fix erroneous Kubernetes configuration changes. Such tools would significantly enhance the resilience, security, and maintainability of modern cloud-native systems.

1.2 Problem & Goal statement

The goal of this research is to support Kubernetes practitioners and DevOps engineers in better understanding configuration-related changes by conducting a large-scale empirical study of Kubernetes-based open-source repositories. Through systematic categorization of configuration modifications and analysis of historical commit data, we aim to identify distinct types of configuration-related commits and uncover their frequency.

Accordingly, we answer the following research questions:

- ***RQ1:*** *What are the common types of configuration changes in Kubernetes repositories?*
- ***RQ2:*** *How frequently do these types of configuration-related commits occur?*

By answering these questions, this study provides actionable insights into how Kubernetes configurations evolve in practice, highlighting the most common types of changes and their frequency which is ultimately supporting the development of better tooling and helping practitioners ramp up more effectively in Kubernetes environments.

1.3 Overview of Methodology

To address these research questions, we conduct a large-scale empirical study analyzing 28,675 commits from 185 open source Kubernetes-related repositories. Our methodology consists of the following steps:

1. Repository Selection: We filter public repositories on GitHub using the GHTorrent dataset, selecting those that frequently update Kubernetes configuration files.

2. Commit Extraction and Categorization: We manually classify commits into eight configuration change categories, ensuring a structured taxonomy.

3. Multi-Phase Qualitative Coding: First, we applied a multi-phase qualitative coding process beginning with existing Swanson's taxonomy[12] and extending it through open coding to further identify meaningful categories of configuration-related commits.

4. Frequency Analysis of Configuration Commit Categories: We analyzed the overall frequency of each configuration related commit category to assess how often different types of changes occurs in practice.

1.4 Thesis Organization

The remainder of this thesis is structured as follows:

• Section 2 provides an overview of Kubernetes configurations and reviews related research in software configuration management.

• Section 3 presents the methodology, including data collection, commit categorization.

• Section 4 discusses the results of our empirical study, including findings from each research question.

• Section 5 explores the implications of our findings and their significance for Kubernetes practitioners and researchers.

• Section 6 outlines potential threats to validity and limitations of our study.

• Section 7 concludes the thesis with a summary of key takeaways and directions for future work.

1.5 Contributions of the Study

1. Commit Categorization via Qualitative Analysis: A categorization of configuration-related commits in Kubernetes-based container orchestration systems, derived through a multi-phase qualitative analysis combining Swanson's taxonomy with open coding.

2. Comprehensive Classification of Configuration Changes: We categorize Kubernetes configuration commits into eight types, providing a structured taxonomy.

3. Empirical Dataset for Future Research: Our dataset of classified Kubernetes commits can support studies on configuration automation and security enforcement.

Dataset Availability: Datasets and source code used for our paper are publicly-available on line [1].

2 Background and Related Work

2.1 Background

In this section, we provide relevant background and discuss about academic works. First, we provide a brief background on Kubernetes and its architecture. Then we provide background on Kubernetes configurations, which are files written in YAML or JSON format that declaratively define the desired state of various components in a Kubernetes cluster such as deployments, services, and configuration which is for enabling automated, consistent, and version-controlled application management. We further end this section by describing related academic research.

2.1.1 Containers and the Need for Orchestration

Containers have revolutionized modern software development by enabling developers to package applications and their dependencies into lightweight, portable units. These containers ensure that software behaves consistently across different environments, like in a developer's local machine, a testing server, or a production cloud instance. Technologies like Docker made containerization widely accessible, and now containers are at the core of microservice-based architectures and DevOps practices.

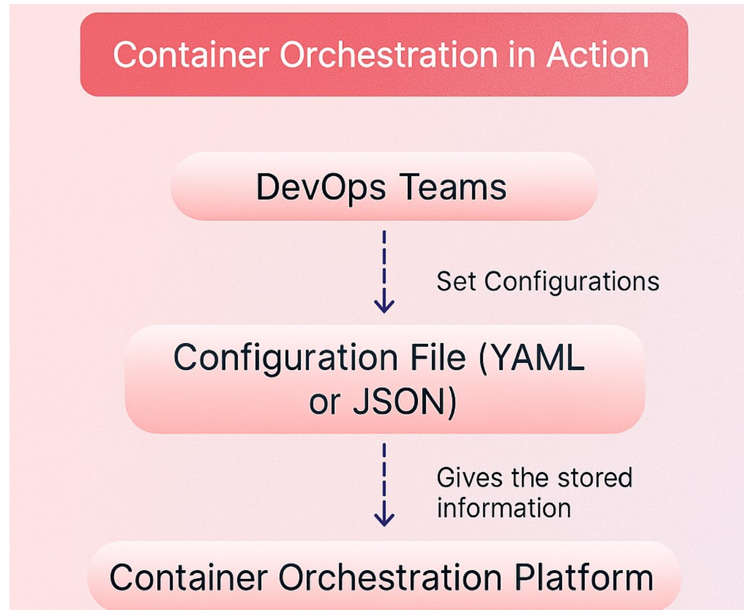

Figure 1: Container orchestration workflow

Figure 1 is the container orchestration workflow from DevOps configuration to lifecycle managed containers. It illustrates the container orchestration process in action. DevOps teams define application behavior through configuration files, typically written in YAML. These are submitted to a container orchestration platform, such as Kubernetes, which handles scheduling, deployment, and lifecycle management. The orchestration platform ensures containers are placed on optimal nodes, deployed reliably, and monitored continuously, thereby abstracting the complexity of infrastructure from developers.

On the other hand, A pod in worker node is the smallest deployable unit in Kubernetes and typically hosts one or more containers. These pods and all other Kubernetes objects are provisioned and managed through YAML-based configuration scripts, often referred to as Kubernetes manifests. A typical configuration file can include settings for resources (CPU/memory), networking, access control, image versions, and more.

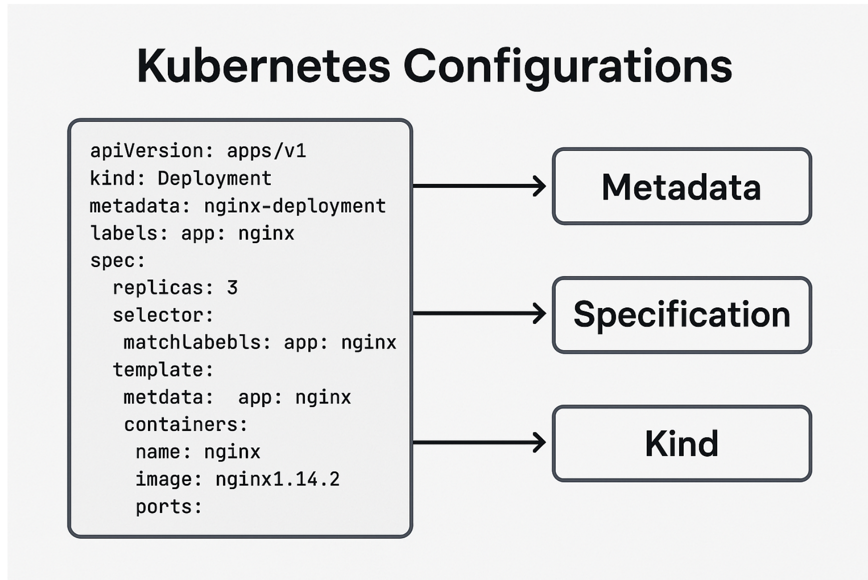

Figure 2: Anatomy of a Kubernetes Configuration File

Figure 2 explains the architecture of Kubernetes Configuration Files. Kubernetes configurations are declarative files which are commonly written in YAML that define the desired state of Kubernetes resources. This example shows a Deployment configuration which specifies the number of replicas, container image, ports, and metadata. Kubernetes uses these configurations to ensure the current state of the cluster matches the user-defined desired state.

However, as software systems scale and move to the cloud, managing hundreds or thousands of containers manually becomes a very challenging task. This challenge gave rise to container orchestration, which automates the deployment, scaling, and management of containers across clusters of machines.

2.1.2 Containers and the Need for Orchestration

A container is a lightweight, portable, and executable software package that includes everything needed to run an application — code, runtime, libraries, system tools, and environment

variables. Containers isolate applications at the process level while sharing the same underlying OS kernel, making them more efficient than traditional virtual machines. Technologies like Docker have made containerization mainstream by simplifying the build and deployment of containers across environments. Figure 3 explains the architecture of containers in detail.

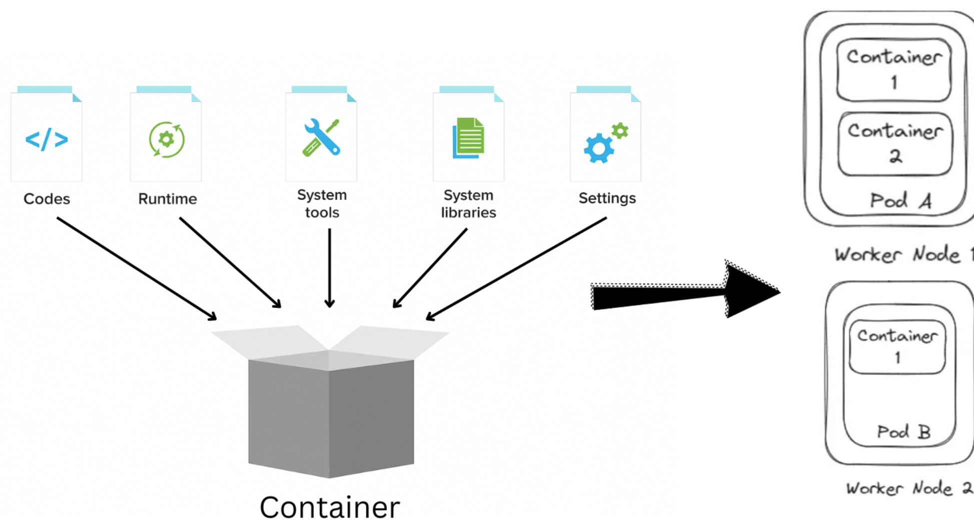

Figure 3: Overview of a Container

Containers solve a long-standing issue in software development by packaging the application and its dependencies together, containers ensure consistent behavior across local, staging, and production environments. This consistency, portability, and lightweight nature have led to widespread adoption of containers in microservices-based architectures and DevOps workflows.

However, managing containers at scale introduces a new set of challenges. In large systems, developers and operators often need to deploy, monitor, scale, and update hundreds or thousands of containers. Tasks such as load balancing, rolling updates, service discovery, and fault recovery become complex and error-prone if done manually. This operational complexity gave rise to the need for container orchestration platforms like Kubernetes.

2.1.3 Kubernetes and Its Role in Container Orchestration

Kubernetes has emerged as the industry-standard platform for container orchestration. Developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF) [3], Kubernetes provides a declarative approach to infrastructure management. It abstracts away the underlying infrastructure and allows practitioners to define how applications should run using high-level configuration files written in YAML. These configurations manage a wide variety of Kubernetes objects, such as pods, services, deployments, volumes, namespaces, and access controls.

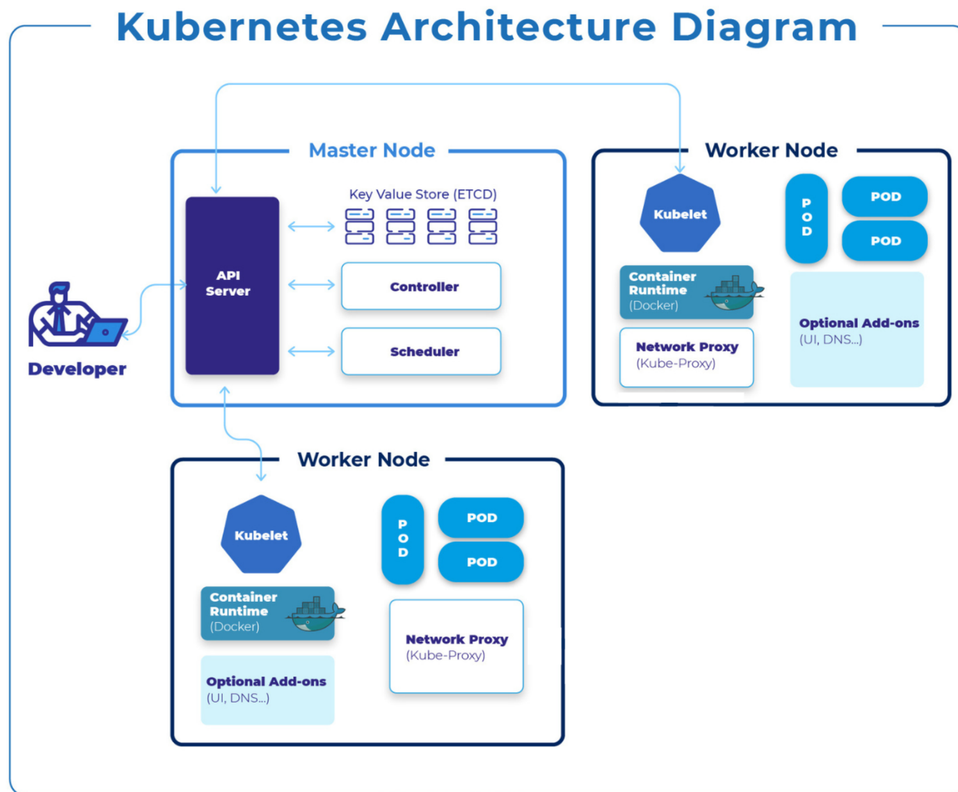

Figure 4: Kubernetes Architecture Diagram

Figure 4 showing interactions between master and worker nodes, configuration flow, and execution of containerized workloads. It illustrates the interaction between core components of Kubernetes. Developers submit configurations to the API Server, which coordinates with the Scheduler and Controller to determine workload placement. The etcd key-value store

maintains cluster state, while Kubelets on worker nodes ensure containers are running correctly. Network management and routing are handled via Kube-proxy, and the optional add-ons provide DNS resolution and user interfaces.

2.2 Motivation for Commit Analysis in Kubernetes Configuration

Configuration changes are primarily tracked through commits in version control systems like Git. Analyzing these commits helps us understand:

- What types of changes are being made?
- Why and how frequently these changes occur?

In the case of Kubernetes, commits involving configuration scripts can reveal patterns of evolution, maintenance, dependency updates, corrections provide insights that are crucial for both researchers and tool builders.

Despite the critical nature of configurations in Kubernetes, there is limited work focused on characterizing commits related specifically to configuration files. Most existing literature either focuses on application code changes or broader system-level changes, overlooking configuration-specific commit behaviors.

This gap motivates our work to categorize and analyze configuration-related commits in Kubernetes-based container orchestration systems. By doing so, we can build a foundational understanding of how configurations evolve and what categories of changes dominate in real-world projects.

2.3 Prior Work

2.3.1 Commit Analysis in Software Engineering

Commit analysis has a long-standing presence in software engineering research. Foundational work by Swanson [12] proposed a taxonomy of software changes, classifying commits into

corrective, adaptive, and perfective categories. Subsequent studies extended this taxonomy to understand bug fixes, refactorings, and performance-related changes[[5], [11], [2]]. Recent efforts have leveraged natural language processing (NLP) and machine learning to automatically classify commits or predict defects. However, most of these studies treat commits at the code level and rarely focus on YAML-based configuration files.

To the best of our knowledge, no prior work has comprehensively categorized and quantified configuration-related commits specifically within the context of Kubernetes.

2.3.2 Classification of Kubernetes Configuration Changes

Software maintenance research traditionally categorizes changes into corrective, adaptive, and preventive modifications [12](Swanson, 1976). Applying this framework to Kubernetes configurations, we define:

Corrective Changes: Fixes for misconfigurations, syntax errors, and failed deployments.
Adaptive Changes: Updates to accommodate Kubernetes API changes, external dependencies, or evolving infrastructure needs.
Preventive Changes: Refactoring configurations to improve maintainability, security compliance, or performance.

Our study refines this classification further by identifying eight key categories of Kubernetes configuration modifications, providing a more detailed understanding of real-world configuration evolution.

2.3.3 Existing Tools and Techniques for Kubernetes Configuration Management

Several tools have been developed to assist in Kubernetes configuration management 1, each addressing different aspects of validation, enforcement, and deployment:

Tool Name	Description	Primary Use Case
Helm	Package manager for Kubernetes that enables templated deployments.	Simplifies application deployment, though version control can become complex.
Kustomize	Tool that allows modification of Kubernetes configurations without duplication.	Improves maintainability by enabling reusable and environment-specific configuration overlays.
ArgoCD	GitOps-based continuous delivery tool that maintains declarative cluster state.	Ensures consistency between Git repositories and live Kubernetes clusters.
Open Policy Agent (OPA)	Policy engine for enforcing compliance and governance policies on Kubernetes configurations.	Enables organizations to apply and manage security and operational policies at scale.
KubeLinter	Static analysis tool for validating Kubernetes YAML files before deployment.	Identifies potential misconfigurations early in the development pipeline.

Table 1: Overview of Kubernetes Configuration Management Tools

Despite these advancements, manual intervention is still required in many cases, increasing the risk of errors and inconsistencies.

2.4 Summary

In this study, we focus on categorizing configuration-related commits by analyzing 28,675 commits from 185 open-source Kubernetes-based repositories. Using a multi-phase qualitative coding approach, we derive eight distinct categories of configuration changes including Annotations, Corrective, Dependency Updates, Deployment Infrastructure, Documentation, Maintenance, Security & Compliance, and UI-related changes.

Understanding these categories is essential because:

- They reveal real-world practices and common change patterns.
- They serve as a stepping stone for tool support and automation.

They allow researchers to identify gaps and focus areas for future studies in configuration evolution.

By providing this categorization, we fill a critical gap in the field and offer a foundational dataset and framework for subsequent research in Kubernetes configuration engineering.

3 Methodology

3.1 Why Frequency Matters

Understanding the frequency of configuration-related commit categories is important for several reasons:

- **Prioritization:** Frequent categories such as dependency updates or deployment infrastructure changes can guide researchers and tool developers to focus their efforts where automation or support is most needed.
- **Risk Identification:** Categories like corrective or security-related changes may indicate instability or potential vulnerabilities that need to be addressed more proactively.
- **Operational Insights:** DevOps teams can better anticipate configuration workloads and align their CI/CD practices accordingly.

3.2 Mining OSS Repositories

To conduct our empirical study, we mine open-source software (OSS) repositories hosted on GitHub which is the most popular code hosting platform [8]. GitHub is selected due to most widely used platform for collaborative software development. We use **GHTorrent** archieve to extract repositories relevant to Kubernetes configurations [4].

3.2.1 Filtering Criteria

However, as publicly-available GitHub repositories are susceptible to quality issues [8], we start with 14,747,836 repositories and apply the following filtering criteria to ensure dataset quality as summarized in Table 2.

1. Public Availability: The repository must be publicly accessible [10].

2. Kubernetes Relevance: The repository must be tagged with 'kubernetes' to ensure its relevance [10].

3. Kubernetes Configuration Presence: To ensure the relevance of the selected repositories for our analysis, we require that at least 10% of the files within a repository be Kubernetes manifests. This threshold is informed by prior research [6], which demonstrates that configuration files are often co-located with source code and test files. By applying this 10% cutoff, we aim to identify repositories that contain a sufficient number of Kubernetes configuration files (i.e., manifests) to support meaningful analysis.

4. Non-Cloned Repository: The repository must not be a duplicate or fork of another repository.

5. Active Contribution: The repository must have at least 10 contributors to ensure active maintenance. Prior research [9] has also used the threshold of at least 10 contributors.

After applying these filters, we collect 185 OSS Kubernetes repositories from GitHub repositories. We identify 185 repositories containing 28,675 commits that modify Kubernetes configuration files.

Filtering Criterion	**Repository Count**
Initial Repository Count	14,747,836
Criterion-1 (Available)	10,947,836
Criterion-2 ('kubernetes' tag)	1,410
Criterion-3 ($\geq$10% Kubernetes scripts)	1,087
Criterion-4 (Not a clone)	1,079
Criterion-5 (Contributors $\geq$ 10)	185
Final Repository Count	**185**

Table 2: Repository Filtering Process

3.3 Categorization of Commits (RQ1)

To classify configuration-related commits, we perform a multi-phase qualitative coding process:

1. Closed Coding Phase: We apply Swanson's Taxonomy[12], categorizing commits as:

o **Adaptive:** Changes due to environmental updates.

o **Corrective:** Fixes for reported defects.

o **Maintenance:** Codebase improvements unrelated to defects.

o **Unclassified:** Changes that do not fit predefined categories.

2. Open Coding Phase: For unclassified commits, we apply manual open coding to derive new categories.

3.3.1 Identified Commit Categories

Our analysis results in the following eight categories, out of which, Corrective & Maintenance category are already represented in Swanson's Taxonomy[12].

- **Annotations:** Configuration changes involving Kubernetes metadata annotations.
- **Corrective:** Fixes for defects in Kubernetes configurations.
- **Dependency Updates:** Modifications to software dependencies.
- **Documentation:** Updates to configuration-related documentation.
- **Security and Compliance:** Changes that enhance security policies.
- **Maintenance:** Refactoring and cleanup of Kubernetes configuration files.
- **UI Changes:** Modifications to Kubernetes dashboards and user interfaces.
- **Deployment Infrastructure:** Changes affecting Kubernetes deployment specifications.

3.3.2 Detailed Manual Categorization Process

Since all categorization in RQ1 was performed manually, the process involved carefully analyzing each commit message, the associated file changes. The manual classification followed these steps:

1. Commit Message Analysis: Each commit message was read thoroughly to understand the intent of the change. If the message was vague, further analysis was conducted using commit diffs.

2. File Type Inspection: YAML files containing Kubernetes configurations were examined to determine the change.

3. Iterative Refinement: Categorization was performed iteratively, ensuring consistency.

This detailed process resulted in an accurate and robust classification of Kubernetes configuration changes, providing valuable insights into their patterns and implications.

3.4 Frequency Analysis of Commit Categories (RQ2)

To answer RQ2, we quantify the frequency of each identified configuration-related commit category. The goal is to assess how often each category appears in real-world Kubernetes-based repositories, providing a practical understanding of which types of configuration changes are most prevalent. This analysis also highlights dominant trends in configuration evolution.

3.4.1 Frequency Calculation Approach

We calculate the frequency for each category by applying the following steps:

- **Step 1: Category Mapping** — Using the results from RQ1, we label all 28,675 configuration-related commits with their corresponding category.

- **Step 2: Aggregation** — For each category, we count the number of commits that belong to that category.
- **Step 3: Normalization** — We compute the percentage share of each category relative to the total number of categorized commits to facilitate comparison across categories.

This provides a static (non-temporal) snapshot of how frequently each type of configuration change is observed across all repositories.

3.4.2 Findings Overview

Our frequency analysis reveals that:

- **Dependency Updates** represent the highest number of configuration-related changes, accounting for a significant portion of total commits.
- **Deployment Infrastructure** and **Corrective** changes also appear frequently, indicating that infrastructure maintenance and bug fixing are common tasks in Kubernetes configurations.
- Less frequent categories like **UI Changes** or **Annotations** still play vital roles in specific operational contexts.

These findings illustrate the distribution of effort across different types of configuration tasks and provide a grounded understanding of real-world Kubernetes development practices.

3.5 Summary

The rigorous manual classification of 28,675 commits provided a comprehensive understanding of Kubernetes configuration changes. By systematically categorizing these commits, this study offers valuable insights for Kubernetes practitioners, DevOps engineers, and researchers aiming to improve configuration management and automation in cloud-native environments.

4 Results

The results of our empirical investigation, which address all two research questions are presented in this part. Our study involved a large-scale manual classification of 28,675 commits from 185 open-source repositories, focusing on identifying configuration change patterns.

4.1 Categorization of Configuration Changes

To better understand the nature of Kubernetes configuration modifications, we classified commits into eight distinct categories based on their purpose and impact (Table 3). The categorization process involved manual inspection of commit messages and code diffs, ensuring a manual review of an accurate mapping of each commit to a specific type of change. The categorization followed a multi-phase qualitative analysis approach combining both closed and open coding techniques.

Closed Coding Phase: Initially, we applied Swanson's Taxonomy[12](corrective, adaptive, and perfective changes) as a guiding framework to categorize commits. In this phase:

- Commits explicitly mentioning fixes or bug resolutions were tagged as Corrective.
- Commits indicating changes due to API version upgrades or support for new features were labeled as Adaptive.
- Refactoring or restructuring changes were marked as Perfective/Maintenance.

However, a large number of commits did not clearly fall into these pre-defined categories, necessitating a deeper examination through open coding.

Open Coding Phase: In the second phase, we manually reviewed commit messages and corresponding diffs of uncategorized commits. Through this process, we iteratively identified additional patterns. This grounded approach led us to expand the classification into eight refined categories: The in-depth explanation of all categories is explained in Section 4.2.

Category	Description	Example Changes
Annotations	Modifications related to Kubernetes annotations for improved metadata handling and observability.	Adding ingress configurations or updating Helm chart annotations for Prometheus.
Corrective	Fixes for misconfigurations, syntax errors, or performance issues, improving system stability.	Fixing a bug causing pod restart failures or optimizing memory usage in a Kubernetes controller.
Dependency Updates	Changes related to managing dependencies, including version upgrades and feature support extensions.	Upgrading Kubernetes support to v1.22.x or bumping Go version for compatibility.
Deployment Infrastructure	Enhancements to deployment configurations, templates, and infrastructure automation.	Allowing persistent volume configurations in Grafana Helm charts or updating pod specifications.
Documentation	Updates to documentation, user guides, and API references for better clarity and usability.	Updating the README to reflect API changes or adding deployment instructions.
Maintenance	Code restructuring and cleanup to improve readability, maintainability, and adherence to coding standards.	Removing deprecated APIs, refactoring configuration files, and optimizing deployment templates.
Security & Compliance	Updates addressing security vulnerabilities, enforcing policies, and ensuring regulatory compliance.	Patching a vulnerability in token management or adding admission control webhooks to validate configurations.
UI Changes	Modifications enhancing user interface (UI) and user experience (UX). Includes design, accessibility, and usability improvements.	Redesigning the Kubernetes dashboard to improve navigation and adding tooltips for better usability.

Table 3: Categories of Kubernetes Configuration Changes

4.2 Example Integration

4.2.1 UI Changes

This category includes changes that add new features, enhance existing functionality, or improve the user interface (UI) and user experience (UX). It focuses on improving visual and accessibility aspects of applications. It also includes updates to improve the user interface, labels, or overall user experience, including accessibility and design enhancements.

Example 1: Listing 1 focuses on AnalysisRun and Experiment Custom Resource Definition that are being updated with additionalPrinterColumns. These columns allow for custom information to be displayed when listing resources using kubectl get.

Github URL: UI Changes Example

```
-   replicaSets:
-               items:
-                type: string
-               type: array

+   additionalPrinterColumns:
+   JSONPath: .status.phase
+    description: AnalysisRun status
+    name: Status
```

Listing 1: Example of an UI Changes in Kubernetes configurations.

Example 2: Listing 2 focuses on the update on the visual aspect of the Helm chart and possibly aligns with the current branding of the project, improving the user interface and visual consistency.

Github URL: UI Changes Example

```diff
-   icon:
    https://raw.githubusercontent.com/cncf/artwork/master/
    projects/k8gb/icon/color/k8gb-icon-color.svg

+   icon: https://www.k8gb.io/assets/images/icon-192x192.png
```

Listing 2: Example of an UI Changes in Kubernetes configurations.

Example 3: Listing 3 focuses on changing the icon improves the visual representation of the chart, enhancing the user experience by potentially providing a more recognizable or branded image for the project.

Github URL: UI Changes Example

```diff
-   icon:
    https://raw.githubusercontent.com/cncf/artwork/master/
    projects/k8gb/icon/color/k8gb-icon-color.svg

+   icon: https://avatars.githubusercontent.com/u/84017757?s=200
```

Listing 3: Example of an UI Changes in Kubernetes configurations.

4.2.2 Security and Compliance

This category includes changes to address security vulnerabilities, ensuring adherence to the principles of integrity, confidentiality, and availability. It involves mitigating risks, preventing unauthorized access, and protecting sensitive data. It also focuses on validating annotations and adding webhooks for verification.

Example 1: Listing 4 focuses on using a distroless image for the admission policy engine, which is a security measure aimed at reducing the attack surface by removing unnecessary dependencies.

Github URL: Security and Compliance Example

```
-   {{- include
"helm_lib_module_pod_security_context_run_as_user_nobody"
. | nindent 6 }}
+   {{- include
"helm_lib_module_pod_security_context_run_as_user_deckhouse"
. | nindent 6 }}
```

Listing 4: Example of an Security and Compliance in Kubernetes configurations.

Example 2: Listing 5 focuses on reverts the bpf masquerading for all cilium installations except OpenStack, and it introduces a conditional check for the masquerading mode in the Cilium configuration. This ensures that the network traffic is handled securely and efficiently based on the underlying environment and configuration. It also ensures that masquerading for network traffic in cilium can be managed securely and appropriately based on the environment.

Github URL: Security and Compliance Example

```
-   cilium:
{{ b64enc "{\"mode\": \"DirectWithNodeRoutes\"}"
| quote }}
+   cilium:
{{ b64enc "{\"mode\": \"DirectWithNodeRoutes\", \"masqueradeMode\": \"Netfilter\"}"
| quote }}
```

Listing 5: Example of an Security and Compliance in Kubernetes configurations.

Example 3: Listing 6 focuses on enhance security by enabling TLS (transport layer security) encryption for Prometheus, ensuring that communication is secure, preventing unauthorized access, and following security best practices.

Github URL: Security and Compliance Example

```
-   {{- if .Values.prometheusOperator.tlsProxy.enabled }}
+   {{- if or .Values.prometheusOperator.tlsProxy.enabled
.Values.prometheusOperator.tls.enabled }}
```

Listing 6: Example of an Security and Compliance in Kubernetes configurations.

4.2.3 Corrective

This category focuses on performing corrective actions to repair defects in Kubernetes configuration scripts. This category has the following subcategories:

(i) Bug Fixes and Error Recovery: This subcategory consists of fixing bugs causing exceptions and enhancing fallback mechanisms to prevent system crashes.

Example 1: Listing 7 focuses on removing the need for anyuid permission on OpenShift by replacing hardcoded runAsUser and runAsGroup values in Helm charts with dynamic fields (.ProxyUID and .ProxyGID). It resolves a defect where deployments required anyuid permissions, which was problematic in OpenShift. It ensures fallback mechanisms by defaulting to 1337 when annotations are unavailable, thus preventing deployment errors and enhancing error recovery.

Github URL: Bug Fixes and Error Recovery Example

```
-   runAsUser: 1337
+   runAsUser: {{ .ProxyUID
| default "1337" }}

-   runAsGroup: 1337
+   runAsGroup: {{ .ProxyGID
| default "1337" }}
```

Listing 7: Example of a Bug Fix and Error Recovery in Kubernetes configurations.

Example 2: Listing 8 focuses on modifying the behavior when an installation fails. Instead of generating new secrets or causing failures, it ensures that secrets are preserved, and the system can recover from a failure without generating new secrets or interfering with the previously existing configuration. This ensures better error recovery when an installation fails.

Github URL: Bug Fixes and Error Recovery Example

```
    args:
            - |
              set -e
-              kubectl -n otomi get cm otomi-status &&
                echo 'Already installed. Installation of Otomi with chart
                can happen only once! To destroy or modify the configuration
                please use the Otomi CLI.' && exit 1
-             binzx/otomi bootstrap
              binzx/otomi apply
              binzx/otomi commit

    args:
            - |
              set -e
+             [ $(kubectl -n otomi get cm otomi-status) ] &&
              echo 'The initial Otomi deployment hes been performed succesfully!
              To modify the configuration please use either the Otomi CLI
              or Otomi console.' && exit 0
+             binzx/otomi bootstrap
              binzx/otomi apply
              binzx/otomi commit
```

Listing 8: Example of a Bug Fix and Error Recovery in Kubernetes configurations.

Example 3: Listing 9 focuses on the changes in container image versions from one SHA to another, which suggests that the new image versions may contain bug fixes or updates. This is typical of a corrective action taken to repair defects in the Kubernetes configuration scripts.

Github URL: Bug Fixes and Error Recovery Example

```
    - name: mixer
-     image: gcr.io/istio-testing/mixer:3561e297ceb365ba990cbf376d99713b0a0add69
      imagePullPolicy: IfNotPresent

    - name: mixer
+     image: gcr.io/istio-testing/mixer:49e721e15d481cd5d92d9a2b30b5e8fcdcafdb63
      imagePullPolicy: IfNotPresent
```

Listing 9: Example of a Bug Fix and Error Recovery in Kubernetes configurations.

(ii) Performance Improvements: This subcategory includes improving efficiency or resource handling through optimizations, such as reducing latency or improving caching.

Example 1: Listing 10 improves the efficiency of Prometheus discovery in Kubernetes by limiting its scope to only relevant namespaces, specifically the namespace where the release is deployed (e.g., .Release.Namespace).

Github URL: Performance Improvements Example

```
   relabel_configs:
-      - source_labels: [__meta_kubernetes_namespace,
__meta_kubernetes_service_name, __meta_kubernetes_endpoint_port_name]

+   namespaces:
+          names:
+          - {{ .Release.Namespace }}
   relabel_configs:
+      - source_labels: [__meta_kubernetes_service_name,
__meta_kubernetes_endpoint_port_name]
```

Listing 10: Example of Performance Improvements in Kubernetes configurations.

Example 2: Listing 11 includes corrective changes to refine miscellaneous configurations, improve performance, and address minor issues. Key fixes include allowing an additional 2 MB of resources, likely optimizing memory usage, and dropping a security owner requirement, simplifying workflows while maintaining functionality.

Github URL: Performance Improvements Example

```
    env:
-   PILOT_ENABLE_INBOUND_PASSTHROUGH: "false"
    PILOT_ENABLE_HBONE: "true"
    CA_TRUSTED_NODE_ACCOUNTS: "istio-system/ztunnel,kube-system/ztunnel"

    env:
+   VERIFY_CERTIFICATE_AT_CLIENT: "true"
+   ENABLE_AUTO_SNI: "true"
    PILOT_ENABLE_HBONE: "true"
    CA_TRUSTED_NODE_ACCOUNTS: "istio-system/ztunnel,kube-system/ztunnel"
```

Listing 11: Example of Performance Improvements in Kubernetes configurations.

Example 3: Listing 12 focusses on scoping of the destination service query to Istio metrics reduces unnecessary data processing and ensures accurate results. It's also replacing the

unsupported dynamic query label values(reporter) with a static custom value eliminates redundant operations, which can improve performance by avoiding unsupported and inefficient operations.

Github URL: Performance Improvements Example

```
- {\"selected\":true,\"text\":\"destination\",\"value\":\"destination\"},
\"datasource\":\"Prometheus\",\"definition\":\"\",\"hide\":0,\"includeAll\":false,
\"label\":\"Reporter\",\"multi\":true,\"name\":\"qrep\",\"options\":
[],\"query\":\"label_values(reporter)\",\"refresh\":1,\"regex\":\"\",
\"skipUrlSync\":false,\"sort\":2,\"tagValuesQuery\"
:\"\",\"tags\":[],\"tagsQuery\":\"\",\"type\":\"query\",
\"useTags\":false},{\"allValue\":null,\"current\":

+   {\"selected\":true,\"text\":\"destination\",\"value\":\"destination\"},
\"datasource\":\"Prometheus\",\"definition\":\"\",
\"hide\":0,\"includeAll\":false,\"label\":\"Reporter\",\"multi\":true,
\"name\":\"qrep\",\"query\":\"source,destination\",
\"refresh\":1,\"regex\":\"\",\"skipUrlSync\":false,\"sort\":2,
\"tagValuesQuery\":\"\",\"tags\":[],\"tagsQuery\":\"\",
\"type\":\"custom\",\"useTags\":false},{\"allValue\":null,\"current\":
```

Listing 12: Example of Performance Improvements in Kubernetes configurations.

4.2.4 Maintenance

This category focuses on restructuring and cleaning up code to improve its readability, maintainability, and adherence to coding standards. It includes changes aimed at improving the overall quality of the codebase without affecting its functionality.

Example 1: Listing 13 indicates an update to the kube-prometheus-stack in Kubernetes, specifically for the nodeexporter configuration. The change makes the rules for the config-reloaders configurable, allowing users to adjust them as needed. Additionally, a runbook URL was added for better documentation or troubleshooting guidance. This enhancement improves the maintainability of the monitoring stack by allowing easier customization and providing additional resources for users to troubleshoot configuration issues.

Github URL: Maintenance Example

```
    annotations:
        description: The API server is burning too much error budget.
-       runbook_url: https://runbooks.prometheus-operator.dev/
        runbooks/kubernetes/kubeapierrorbudgetburn
        summary: The API server is burning too much error budget.
```

```
    annotations:
        description: The API server is burning too much error budget.
+       runbook_url: {{ .Values.defaultRules.runbookUrl }}/kubernetes/
        kubeapierrorbudgetburn
        summary: The API server is burning too much error budget.
```

Listing 13: Example of Maintenance in Kubernetes configurations.

Example 2: Listing 14 focusses on the configPatches field which is more flexible and adheres to the modern EnvoyFilter specification syntax which technically improves clarity and maintainability of the configuration file.

Github URL: Maintenance Example

```
-   filters:
-  - listenerMatch:
-     portNumber: 15443
-    listenerType: GATEWAY
-   insertPosition:
-     index: AFTER
-    relativeTo: envoy.filters.network.sni_cluster
-   filterName: envoy.filters.network.tcp_cluster_rewrite
-   filterType: NETWORK
-   filterConfig:
-     cluster_pattern: "\\.global$"
-     cluster_replacement: ".svc.{{ .Values.global.proxy.clusterDomain }}"

+   configPatches:
+   +  - applyTo: NETWORK_FILTER
+       match:
+         context: GATEWAY
+         listener:
+           portNumber: 15443
+           filterChain:
+             filter:
+               name: "envoy.filters.network.sni_cluster"
+       patch:
+         operation: INSERT_AFTER
+         value:
+           name: "envoy.filters.network.tcp_cluster_rewrite"
+           config:
+             cluster_pattern: "\\.global$"
+             cluster_replacement: ".svc.{{ .Values.global.proxy.clusterDomain }}"
```

Listing 14: Example of Maintenance in Kubernetes configurations.

Example 3: Listing 15 refers to a change that allows users to configure the Azure authentication method when initializing a cluster. Overall, this change simplifies the configuration of Azure authentication during cluster setup, improving cluster management and maintenance by providing more customizable options.

Github URL: Maintenance Example

```
-   env:
-       - name: ARM_SUBSCRIPTION_ID
-         valueFrom:
-           secretKeyRef:
-             key: SubscriptionID
-             name: cluster-autoscaler-azure
-       - name: ARM_RESOURCE_GROUP
-         valueFrom:
-          secretKeyRef:
-              key: ResourceGroup
-             name: cluster-autoscaler-azure

+   command:
+         - ./cluster-autoscaler
+         args:
+         - --cloud-provider=azure
+        - --cloud-config=/etc/azure/azure.json
+          - --logtostderr=true
+         - --namespace=kube-system
+         - --stderrthreshold=info
+         - --v=2
+         volumeMounts:
+         - name: azureconfig
+           mountPath: /etc/azure
+           readOnly: true
```

Listing 15: Example of Maintenance in Kubernetes configurations.

4.2.5 Documentation

This category focuses on updates related to user guides, API documentation, or any other descriptive materials aimed at improving understanding and usage.

Example 1: Listing 16 focus on providing a preview walkthrough for AWS App Mesh ingress enhancements, which likely updates documentation to guide users through new features or functionality.

Github URL: Documentation Example

```
+   Parameters:
+     ColorTellerImageName:
+       Description: The name for the color teller image
+       Type: String
+   Resources:
+     ColorTellerRepository:
+       Type: AWS::ECR::Repository
+       Properties:
+         RepositoryName: !Ref ColorTellerImageName
```

Listing 16: Example of Documentation in Kubernetes configurations.

Example 2: Listing 17 clarifies documentation related to the instance class in the cloud-provider-vsphere setup, which directly improves the readability and understanding of the documentation.

Github URL: Documentation Example

```
- description: Namespace overrides the environment
  namespace value in the ksonnet app.yaml

+ description: Namespace specifies the target
  namespace for the application's resources.
  The namespace will only be set for namespace-scoped
  resources that have not set a value for .metadata.namespace
```

Listing 17: Example of Documentation in Kubernetes configurations.

Example 3: Listing 18 focusses on detailed explanations of resource fields to help users better understand and use the CRD definitions. These updates make the CRD more user-friendly and accessible, especially for those new to the Kubernetes API or working with the custom resource.

Github URL: Documentation Example

```
    example: 20
+         x-doc-default: 20
          type: integer
        template:
          description: |
            Path to the template to be cloned. Relative to the datacenter.
+           By default, a template from the `master` InstanceClass is used.
          example: dev/golden_image
          type: string
```

Listing 18: Example of Documentation in Kubernetes configurations.

4.2.6 Dependency Updates

This category includes changes related to updating or managing dependencies, including version upgrades and downgrades. This category has the following subcategories:

(i) Version Updates: This subcategory includes updates to dependencies or tool versions, focusing on maintaining compatibility (e.g., ”bump Go version”).

Example 1: Listing 19 updates the Grafana dependency version for the kube-prometheus-stack, aligning with the need to maintain compatibility with new versions.

Github URL: Version Updates Example

```
    version: 1.17.0
    - name: grafana
    repository: https://grafana.github.io/helm-charts
-     version: 6.7.4
-    digest: sha256:3ad6db588084a1f225ee1a3bb8e5d47b30f1fab9dbba701e039d0919aaf86aef
-    generated: "2021-04-14T12:41:24.546213-04:00"

    version: 1.17.0
    - name: grafana
    repository: https://grafana.github.io/helm-charts
+     version: 6.8.0
+   digest: sha256:54bd36030ef0f5cac21d8c06b2a98b108cbfccdbba112770f8b8158fe479a00e
+   generated: "2021-04-22T14:35:03.234101561+02:00"
```

Listing 19: Example of Version Updates in Kubernetes configurations.

Example 2: Listing 20 focuses on to keep the chart and application aligned with the latest version, ensuring compatibility and improved functionality.

Github URL: Version Updates Example

```
-   version: 0.5.0
-   appVersion: v0.10.0

+   version: 0.6.0
+   appVersion: v0.11.0
```

Listing 20: Example of Version Updates in Kubernetes configurations.

Example 3: Listing 21 focusses on updating various dependencies, including Kube dependencies, controller-tools, cert-manager, Gatekeeper, and Golang modules, ensuring compatibility with newer versions. It also updates CRD generation by bumping controller-gen.kubebuilder.io/version from v0.11.3 to v0.13.0.

Github URL: Version Updates Example

```
    annotations:
-       controller-gen.kubebuilder.io/version: v0.11.3
        kubermatic.k8c.io/location: master,seed
-     creationTimestamp: null

    annotations:
+       controller-gen.kubebuilder.io/version: v0.13.0
        kubermatic.k8c.io/location: master,seed
```

Listing 21: Example of Version Updates in Kubernetes configurations.

(ii) Feature Support Extensions: This subcategory includes support for new features, versions, or external integrations (e.g., ”support for k8s v1.22.x”).

Example 1: Listing 22 focusses on support for integrating the prometheus-adapter with cert-manager, an external dependency. It extends the adapter’s features by allowing users to leverage cert-manager for automatic certificate management.

Github URL: Feature Support Extensions Example

```
        service:
            name: {{ template "k8s-prometheus-adapter.fullname" . }}
            namespace: {{ .Release.Namespace | quote }}
-         {{ if .Values.tls.enable -}}
          caBundle: {{ b64enc .Values.tls.ca }}
          {{- end }}
          group: custom.metrics.k8s.io
          version: v1beta1
-         insecureSkipTLSVerify: {{ if .Values.tls.enable }}
          false{{ else }}true{{ end }}
          groupPriorityMinimum: 100
          versionPriority: 100

        service:
            name: {{ template "k8s-prometheus-adapter.fullname" . }}
            namespace: {{ .Release.Namespace | quote }}
+         {{- if .Values.tls.enable -}}
          caBundle: {{ b64enc .Values.tls.ca }}
          {{- end }}
          group: custom.metrics.k8s.io
          version: v1beta1
+         insecureSkipTLSVerify: {{ if or .Values.tls.enable .Values.
          certManager.enabled }}false{{ else }}true{{ end }}
          groupPriorityMinimum: 100
          versionPriority: 100
```

Listing 22: Example of Feature Support Extensions in Kubernetes configurations.

Example 2: Listing 23 focusses on support for integrating the changes and extend the functionality of the platform by improving support for vSphere-specific features (e.g., tag categories, tagging, and cluster reconciliation).

Github URL: Feature Support Extensions Example

```
-   tagCategoryID:
-       description: This is category for the machine deployment tags
-       type: string

+   tagCategory:
+       description: TagCategory is the vSphere tag category that is
        owned by KKP. It is really important to note that, if the user
        set this field manually, KKP will claim this category as it's
        own, thus if the cluster has been deleted, the category and it
        is all underlying tags will be deleted as well.
+   properties:
+       id:
+           description: ID represents the category id for the machine
            deployment tags.
+           type: string
```

Listing 23: Example of Feature Support Extensions in Kubernetes configurations.

Example 3: Listing 24 focusses on introducing a validator for performing offline CRD (Custom Resource Definition) validation in tests. This ensures that CRDs can be validated without requiring a running Kubernetes cluster. Additionally, it includes updates to test data, specifically modifying mismatch.yaml.

Github URL: Feature Support Extensions Example

```
-   listeners: []

+   listeners:
      - port: 80
        protocol: HTTP
        routes:
          namespaces: {}
          kind: HTTPRoute
```

Listing 24: Example of Feature Support Extensions in Kubernetes configurations.

4.2.7 Annotations

This category focuses on adding, changing, or managing annotations in Kubernetes resources. Annotations in Kubernetes are key-value pairs used to store non-identifying metadata for objects like pods and services. They provide additional context or configuration for

tools and processes interacting with these resources.

Example 1: Listing 25 focus on providing a preview walkthrough for AWS App Mesh ingress enhancements, which likely updates documentation to guide users through new features or functionality.

Github URL: Annotations Example

```
labels:
       {{- include "otel-demo.labels" . | nindent 4 }}
-      {{- with .Values.serviceAccount.annotations }}
      annotations:
-       {{- toYaml . | nindent 4 }}
      {{- end }}


    labels:
      {{- include "otel-demo.labels" . | nindent 4 }}
+     {{- if .Values.serviceAccount.annotations }}
      annotations:
+           {{- range $key, $value := .Values.serviceAccount.annotations }}
+           {{- printf "%s: %s" $key (tpl $value $ | quote) | nindent 4 }}
+           {{- end }}
          {{- end }}
        {{- end }}
```

Listing 25: Example of Annotations in Kubernetes configurations.

Example 2: Listing 26 adds the ability to include annotations on a pod, which directly impacts the way annotations are used and managed.

Github URL: Annotations Example

```
+   {{- if .Values.annotations }}
+         annotations:
+           {{- toYaml .Values.annotations | nindent 8 }}
+         {{- end }}
```

Listing 26: Example of Annotations in Kubernetes configurations.

Example 3: Listing 27 provides users a way to manage metadata and customize the deployment behavior for components, aligning with the broader goal of making Kubernetes deployments more flexible and user-centric.

Github URL: Annotations Example

```
    annotations:
            sidecar.istio.io/inject: "false"
            checksum/config-volume: {{ .Files.Get "files/injection-template.yaml"
            | sha256sum }}
+           {{- if .Values.sidecarInjectorWebhook.podAnnotations }}
+    {{ toYaml .Values.sidecarInjectorWebhook.podAnnotations | indent 8 }}
+            {{- end }}
```

Listing 27: Example of Annotations in Kubernetes configurations.

4.2.8 Deployment Infrastructure

This category includes updates that improve the configuration, setup, management, and automation of Kubernetes deployments and infrastructure. It focuses on enhancing deployment behavior, enabling more flexible configurations for specific use cases, and optimizing deployment templates and runtime settings.

Example: Allowing persistent volume configurations in Grafana Helm charts (Helm, configuration) or enabling more dynamic and flexible deployment behavior to address specific use cases. This category has the following subcategories:

(i) Deployment Units: This subcategory consists changes to deployment templates, pod specifications, and runtime configurations.

Example 1: Listing 28 focusses on enhancing Kubernetes workload configurations to allow more dynamic and flexible deployment behavior with a unified affinity trait. It introduces a new trait of allowing users to define affinity rules and tolerations at the pod level.

Github URL: Deployment Units Example

```
    labels:
+       custom.definition.oam.dev/deprecated: "true"
        custom.definition.oam.dev/ui-hidden: "true"
```

Listing 28: Example of Deployment Units in Kubernetes configurations.

Example 2: Listing 29 focusses on update pod specifications to enhance runtime configurations (e.g., resource limits, security context, image definitions), enforcing specific node affinity for pod placement.

Github URL: Deployment Units Example

```
-       env:
-       - name: VAR1
-         valueFrom:
-           secretKeyRef:
-             key: VAR1
-             name: {{ include "app.fullname" . }}-my-secret-vars

+         image:
+           repository: controller
+           tag: latest
+         resources:
+           limits:
+             cpu: 100m
+             memory: 30Mi
```

Listing 29: Example of Deployment Units in Kubernetes configurations.

Example 3: Listing 30 focusses on modifying the memory management of the OpenTelemetry Collector using memory limits defined as percentages (limit-percentage, spike-limit-percentage). These changes directly affect how memory is managed at the pod level

Github URL: Deployment Units Example

```
    memory_limiter:
            check_interval: 5s
-           limit_mib: 156
-           spike_limit_mib: 48


    memory_limiter:
            check_interval: 5s
+           limit_percentage: 80
+           spike_limit_percentage: 25
```

Listing 30: Example of Deployment Units in Kubernetes configurations.

(ii) Deployment Configurations: Updates to service-related YAML files, storage configurations, infrastructure parameters, and overall deployment flexibility.

Example 1: Listing 31 focusses on enhancing Kubernetes workload configurations to allow more dynamic and flexible deployment behavior with a unified affinity trait. It introduces a new trait of allowing users to define affinity rules and tolerations at the pod level.

Github URL: Deployment Configurations Example

```
+    ---
+   apiVersion: v1
+   kind: ConfigMap
+   metadata:
+     name: {{ .Chart.Name }}
+     namespace: d8-monitoring
+     {{- include "helm_lib_module_labels" (list . (dict "app" .Chart.Name))
      | nindent 2 }}
+   data:
+     config.yaml: |-
+       auth_enabled: false
```

Listing 31: Example of Deployment Configurations in Kubernetes configurations.

Example 2: Listing 32 adds a configuration change (x509ignorecn 0), which is related to handling certificates in deployment scenarios, improving infrastructure configuration for secure communication.

Github URL: Deployment Configurations Example

```
name: prepare-test-index
+   env:
+     GODEBUG: x509ignoreCN=0
+
on:
  repository_dispatch:
    types:
```

Listing 32: Example of Deployment Configurations in Kubernetes configurations.

Example 3: Listing 33 focusses on updating YAML files, which are critical for deployment configurations in Kubernetes. It ensures compatibility and eliminates deprecated APIs, aligning with the category's definition. It also deals with improving and modernizing deployment templates and configurations rather than purely enhancing documentation.

Github URL: Deployment Configurations Example

```
-   apiVersion: rbac.authorization.k8s.io/v1beta1
        kind: ClusterRole
    metadata:
      name: skipper-ingress
    rules:

+   apiVersion: rbac.authorization.k8s.io/v1
        kind: ClusterRole
    metadata:
      name: skipper-ingress
    rules:
+    - apiGroups:
+     - networking.k8s.io
+     resources:
+     - ingresses
+     verbs:
+     - get
+     - list
```

Listing 33: Example of Deployment Configurations in Kubernetes configurations.

4.3 Distribution of Configuration Change Per Categories

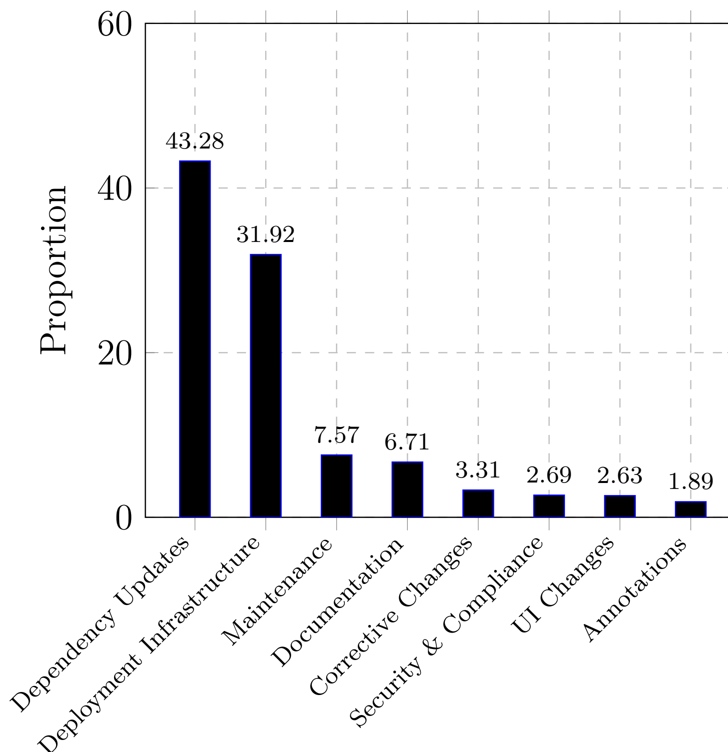

Figure 5: Proportion of Commits per Category

Figure 5 illustrates the distribution of Kubernetes configuration change categories across the analyzed repositories. Dependency updates were the most frequent type of change, accounting for approximately 43.28% of all commits. This indicates the dynamic nature of Kubernetes deployments, where frequent updates are required to maintain compatibility with new software versions. This high proportion can be attributed to the rapid evolution of Kubernetes and its ecosystem, where frequent version releases, API deprecations, and library updates necessitate regular modification of configuration files to ensure system compatibility and stability. Deployment Infrastructure updates accounted for 31.9%, highlighting the enhancements to deployment configurations.

4.4 Distribution Of Contributor Counts Across Kubernetes Repositories

To evaluate the extent of active development across the selected repositories, we analyzed the distribution of contributors. Figure 6 displays a histogram representing the number of repositories (y-axis) grouped by the number of contributors in intervals of 10 (x-axis). The figure offers a more fine-grained view of contributor distribution, aiding the interpretation of repository activity and team dynamics.

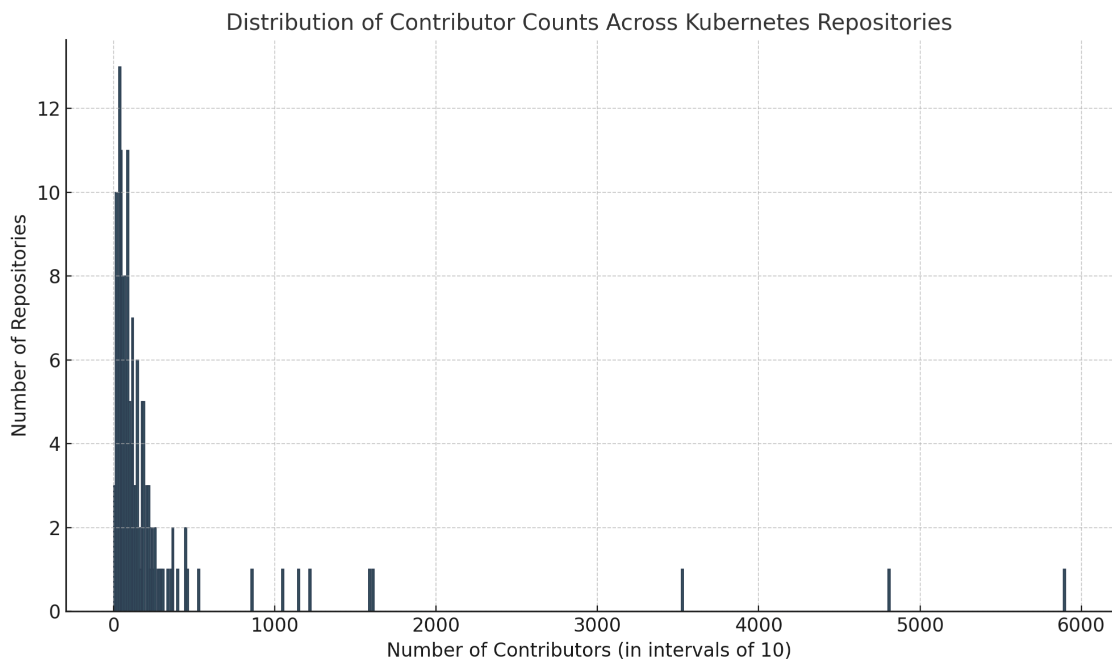

Figure 6: Count of Kubernetes-related Repositories Based on Contributor Count

As shown in the figure, a large proportion of repositories have between 10 and 30 contributors, reflecting the predominance of small to mid-sized development teams within the Kubernetes ecosystem. The distribution gradually tapers off as the number of contributors increases, with only a few repositories exceeding 100 contributors, which typically correspond to high-profile or widely adopted community-driven projects.

4.5 Summary of Key Findings

1. Dependency updates are the most common Kubernetes configuration change (43.30%), emphasizing the importance of managing dependencies efficiently.

2. Deployment Infrastructure updates account for 31.9% of changes, reflecting the frequent need to refine deployment pipelines and cluster behavior.

3. Security and Compliance-related changes, while less frequent, are critical for maintaining secure Kubernetes clusters, highlighting an area where proactive tooling can be valuable.

4. Documentation changes indicate a consistent effort by contributors to improve the usability and understanding of Kubernetes configurations.

5. Annotations and Metadata edits, though smaller in proportion, show that developers often refine how configurations are interpreted or tracked by Kubernetes.

5 Discussion

This study provides the first in-depth characterization of configuration-related commits within Kubernetes-based container orchestration systems. By categorizing 28,675 commits from 185 open-source repositories, we identify eight distinct types of configuration changes and analyze their frequency. These findings offer practical implications and guidance for researchers, developers, and DevOps practitioners helping them design better automation strategies, enforce best practices, and enhance the overall reliability of Kubernetes deployments.

Implications for Practitioners For Kubernetes practitioners and DevOps engineers, understanding which types of configuration changes are most common can help prioritize automation, testing, and monitoring efforts. For example, knowing that Dependency Updates and Deployment Infrastructure changes dominate configuration commits encourages teams to:

- Automate dependency management and validation pipelines.
- Invest in deployment configuration templates and reusable infrastructure patterns.

Implications for Researchers The taxonomy of commit categories created through qualitative analysis can serve as a foundation for future research in:

- Configuration evolution and refactoring in Kubernetes-based systems.
- Automated classification of configuration commits using NLP or machine learning.
- Configuration-related defect prediction and recovery mechanisms.

Enabling Tooling Support Our findings suggest that tool support can be tailored toward high-frequency change categories. For instance, tools that automatically detect, summarize, or visualize Dependency Updates and Corrective changes could greatly reduce manual workload and configuration drift.

6 Threats to Validity

6.1 Construct Validity

Our categorization is based on manual qualitative analysis of commit messages and file changes. While we applied well-established methods like Swanson's taxonomy[12] and iterative open coding, there remains a risk of subjective bias in classification. To reduce this threat, multiple raters were used in both closed and open coding phases. Additionally, some commit messages were ambiguous or lacked sufficient context, which may have introduced subjectivity in interpretation, despite thorough inspection of the corresponding file diffs. Another source of potential construct validity threat arises from commits that exhibited characteristics of multiple configuration change categories. In such cases, to maintain consistency and avoid duplicate classification, we assigned the commit to the first category that it most prominently represented based on its content and message. This one-label-per-commit strategy was chosen to ensure clear categorization while acknowledging that some commits naturally span multiple concerns.

6.2 External Validity

We studied 185 open-source repositories tagged with "Kubernetes" on GitHub, which may not fully represent private or enterprise-level Kubernetes configurations. However, the large sample size and public availability of projects improve generalizability.

6.3 Internal Validity

There's a chance that some commits involved multiple types of changes, which may blur category boundaries. We mitigated this by carefully inspecting file diffs.

7 Conclusions

This study offers the first large-scale empirical investigation of Kubernetes configuration-related commits by examining 28,675 commits from 185 GitHub repositories into eight meaningful types. Our work makes the following key contributions:

- We identify eight distinct categories of configuration-related changes, including commonly occurring categories like Dependency Updates and Deployment Infrastructure, and less frequent yet significant ones like Corrective, Security and Compliance.
- We quantify the frequency of each category, revealing real-world patterns.
- We provide a foundational taxonomy that can guide future research, tool development, and automation practices in Kubernetes environments.

Understanding the nature and frequency of configuration changes is vital for reducing misconfigurations, improving deployment reliability, and enabling better tooling. This work paves the way for future efforts in automated configuration analysis, intelligent DevOps tooling, and configuration-focused software engineering research.

Bibliography

[1] Anonymous Authors 2025. Replication package for paper. `https://figshare.com/s/7c19cc78e0c5b1265031?file=53151494`, 2025. Online;accessed22-Feb-2025.

[2] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. What's a typical commit? a characterization of open source software repositories. In 2008 16th IEEE International Conference on Program Comprehension, pages 182–191, 2008.

[3] Sunil Kumar Reddy Anumandla. Automating Container Orchestration: Innovations and Challenges in Kubernetes Implementation. Robotics Xplore: USA Tech Digest, 1(1):29–43, April 2024.

[4] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), pages 12–21, 2012.

[5] Lile P. Hattori and Michele Lanza. On the nature of commits. In Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE'08, page III–63–III–71. IEEE Press, 2008.

[6] Yujuan Jiang and Bram Adams. Co-evolution of infrastructure and source code - an empirical study. In 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pages 45–55, 2015.

[7] S. Miles. Kubernetes: A Step-By-Step Guide for Beginners to Build, Manage, Develop, and Intelligently Deploy Applications by Using Kubernetes (2020 Edition). Independently Published, 2020.

[8] Craig Cabrey Meiyappan Nagappan Nuthan Munaiah, Steven Kroh. "curating github for engineered software projects". In 2017.

[9] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. Gang of eight: a defect taxonomy for infrastructure as code scripts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE '20, page 752–764, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. Security misconfigurations in open source kubernetes manifests: An empirical study. ACM Trans. Softw. Eng. Methodol., 32(4), May 2023.

[11] Joe F. Shobe, Md Yasser Karim, Motahareh Bahrami Zanjani, and Huzefa Kagdi. On mapping releases to commits in open source systems. In Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014, page 68–71, New York, NY, USA, 2014. Association for Computing Machinery.

[12] E. Burton Swanson. The dimensions of maintenance. In Proceedings of the 2nd International Conference on Software Engineering, ICSE '76, page 492–497, Washington, DC, USA, 1976. IEEE Computer Society Press.