Configuration Defects in Kubernetes

Yue Zhang, Uchswas Paul, Marcelo d'Amorim, and Akond Rahman, *Member, IEEE*

Abstract—Kubernetes is a tool that facilitates rapid deployment of software. Unfortunately, configuring Kubernetes is prone to errors. Configuration defects are not uncommon and can result in serious consequences. This paper reports an empirical study about configuration defects in Kubernetes with the goal of helping practitioners detect and prevent these defects. We study 719 defects that we extract from 2,260 Kubernetes configuration scripts using open source repositories. Using qualitative analysis, we identify 15 categories of defects, of which 7 have not been reported in previously-studied software systems. We find 8 publicly available static analysis tools to be capable of detecting 8 of the 15 defect categories. We find that the highest precision and recall of those tools are for defects related to data fields. We develop a linter to detect two categories of defects that cause serious consequences, which none of the studied tools are able to detect. Our linter revealed 26 previously-unknown defects that have been confirmed by practitioners, 19 of which have already been fixed. We conclude our paper by providing recommendations on how defect detection and repair techniques can be used for Kubernetes configuration scripts. The datasets and source code used for the paper are publicly available online.

Index Terms—configuration, container orchestration, defect, devops, empirical study, Kubernetes

1 Introduction

The use of multiple containers to deploy software projects is a common practice today [102], e.g., Paypal uses 200,000 containers to speed up financial transactions [70]. Setting up and managing multiple containers manually is considered impractical and prone to errors [22], [78], [91]. For that reason, the practice of container orchestration advocates for *automated management of containers* with tools, such as Kubernetes [77] that has yielded benefits for organizations [58]. OpenAI reported that Kubernetes enabled a reduction of deployment time from "a couple of months" to "two or three days." [58] Kubernetes usage aided Adidas to reduce the load time for their e-commerce website by half, and increase the release frequency from once every 4∼6 weeks to 3∼4 times a day [58].

Unfortunately, Kubernetes configuration scripts are not immune to defects. In March 2023, the social media platform Reddit experienced a 5 hour-long outage that impacted millions of its users [51], [83]. The outage occurred because of a defect in a configuration script affecting the network traffic between containers [50], [51]. Figure 1 presents an YAML code snippet showcasing how certain Kubernetes-related configurations were specified when the outage occurred. The defect is due to the incorrect definition of configuration options `nodeSelector` and `peerSelector`, which used the value `node-role.kubernetes.io/master` instead of `node-role.kubernetes.io/control-plane`. The string `master` in the configuration value became obsolete with the release of Kubernetes 1.24 [57]. The entities `nodeSelector` and `peerSelector` are responsible to route the network traffic across containers. As a result of this defect, traffic was routed to a destination that does not exist, resulting in the outage. This defect illustrates the importance of understanding configuration defects in Kubernetes-related computing infrastructure.

```
metadata:
  annotations:
...
spec:
  asNumber: 0
- nodeSelector: has(node-role.kubernetes.io/master)
+ nodeSelector: has(node-role.kubernetes.io/control-plane)
  peerIP: ","
- peerSelector: has(node-role.kubernetes.io/master)
+ peerSelector: has(node-role.kubernetes.io/control-plane)
```

Fig. 1: Excerpt of the configuration defect that caused the Reddit outage [51].

Our paper presents an empirical study about configuration defects in Kubernetes *with the goals of assisting practitioners in preventing defects and guiding researchers in developing automated tools to detect those defects*. The results of the study enable researchers and practitioners (i) to gain insights about the defects in Kubernetes-based computing infrastructure; (ii) to assess the capabilities of existing tools in identifying defects; and (iii) to develop techniques to identify latent defects that occur during Kubernetes-based configuration management.

While the importance of defect categorization has been well-acknowledged in software engineering research [21], [45], [82], a systematic characterization of defects related to Kubernetes configuration management remains under explored. The paper addresses the two following key aspects: (i) an in-depth study on root causes of defects, their consequences, and fix patterns for Kubernetes configuration scripts; and (ii) an exploratory study of the ability to detect static analysis tools. Although there are prior empirical studies on Kubernetes including prior publications from the authors of this paper [75], [77], [85], [86], [103], [113], these publications have not addressed how configurations of Kubernetes-based deployments are specified, and how these

Yue Zhang is with the Department of Computer Science and Software Engineering, Auburn University, Auburn, Alabama, USA
Uchswas Paul is with the Department of Computer Science, NC State University, Raleigh, NC, USA
Marcelo d'Amorim is with the Department of Computer Science, NC State University, Raleigh, NC, USA
Akond Rahman is with the Department of Computer Science and Software Engineering, Auburn University, Auburn, Alabama, USA

specifications can result in defects. The novelty of this paper stems from expanding and detailing the understanding of defects in this domain. We systematically characterize how configuration-related defects occur using code constructs in configuration scripts. This characterization resulted in 15 defect categories, 7 of which have not been reported for previously-studied software systems [42], [76], [88], [104] as well as for Kubernetes-related research investigations [12], [15], [75], [77], [84]–[86], [103], [113].

We answer the following research questions:

- **RQ1 [Categories]**: What are the categories of defects in Kubernetes configuration management?
- **RQ2 [Consequences and Fix Patterns]**: What categories of consequences and fix patterns map to defects that occur during Kubernetes configuration management?
- **RQ3 [Tool Support]**: How frequently do static analysis tools support the detection of defects that occur during Kubernetes configuration management?

We analyze 719 defects that occur in 2,260 configuration scripts mined from 185 open source software (OSS) repositories. We use a qualitative analysis technique called open coding [81] with the obtained data to derive defect categories, consequences, and fix patterns. Using the data, we systematically evaluate the defect detection capabilities of 8 publicly available static analysis tools for Kubernetes. Our empirical study provides insights on the nature of configuration defects and identifies opportunities for developing defect detection techniques for Kubernetes. For example, we find that 533 of the 719 defects are found to cause crashes, incorrect operations, or outages. We construct a linter that detects two categories of defects that cause serious consequences, such as crashes and outages. These two defect categories are not detected by any of the 8 studied tools. With the help of the linter, we have identified 26 previously-unknown defects that have been confirmed by practitioners, and 19 have already been fixed.

Contributions: We list our contributions as follows:

- An evaluation on the performance of static analysis tools to detect defects that occur during Kubernetes configuration management (Section 5.1.2);
- A categorization of consequences and fix patterns for defects that occur during Kubernetes configuration management (Section 4.2); and
- A list of derived defect categories for Kubernetes configuration management (Section 3.2).

Dataset Availability: Datasets and source code used in our paper are publicly available online [112]. The dataset contains data where each of the 719 defects is mapped to their corresponding defect category, consequence, and fix pattern. Source code to construct our linter is available. To further support reproducibility, we have made ConShifu available as a Docker image [111], along with instructions on how to install and run the tool.

2 Background

Kubernetes is the most popular tool to implement container orchestration. Any computing infrastructure managed by

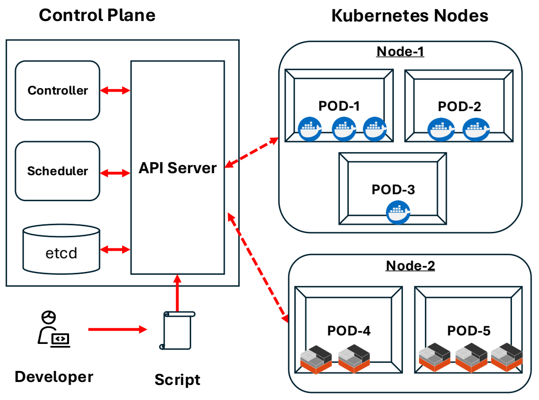

Fig. 2: An overview of the components in a Kubernetes cluster.

Kubernetes is referred to as a Kubernetes cluster [59]. Kubernetes uses objects to provision the cluster computing infrastructure. An *object* is a persistent entity representing the state of the cluster. A *pod* is a common kind of object; it is the most fundamental deployment unit that groups multiple containers together. Configurations for pods and other Kubernetes entities are specified using *configuration scripts* that are typically written in the YAML format. As Figure 2 shows, the API server stores configurations in a database called 'etcd.' With the provided configurations, the API server decides which pods can host the given containers. A controller and scheduler are automated agents that control the state of the Kubernetes to identify a suitable node for a pod. A configuration script can either be a *Kind script* or a *Helm script* [77].

Kind script: Kind scripts contain configurations for `kind`, which is a specific type of Kubernetes object. Kind scripts are executed using Kubernetes-provided utilities, such as 'kubectl' [59]. Listing 1 shows an example of a pod specified with a Kind script. This script defines a pod that runs a single container using the image 'myimage.'

Helm script: Helm is a package manager for Kubernetes that simplifies configuration management for Kubernetes [13]. A Helm script is developed using YAML, and a group of Helm scripts is referred to as a Helm chart. In a Helm chart, variables and default configuration values are defined in a script labeled as 'values.yaml.' [13] These variables and configuration values are loaded dynamically into scripts called 'templates' through template directives [13]. Listing 2 shows an example of a template.

3 RQ1: Categories of Defects in Kubernetes Configuration Scripts

We provide the methodology and results for RQ1 respectively, in Sections 3.1 and 3.2.

3.1 Methodology

We use the following steps:

3.1.1 *Identify Defects from OSS projects*

We follow three steps to identify defects.

Step#1 - Mine OSS repositories from GitHub: We identify defects by mining OSS repositories hosted on GitHub,

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: myimage
    ports:
    - containerPort: 80
```

Listing 1: An example of a Kind script.

```yaml
# configuration values defined in a Helm script, "values.yaml"
replicaCount: 2 --> Configuration value for spec.replicas
service:
  portName: https --> Configuration value for spec.ports.name
  portHttps: 80 --> Configuration value for spec.ports.port
# configuration values used by a Helm template
spec:
  replicas: {{ .Values.replicaCount }} --> Template directive
  ports:
    - name: {{ .Values.service.portName }}
    - port: {{ .Values.service.portHttps }}
    - protocol: TCP
```

Listing 2: An example of a Helm script.

which is the most popular code hosting platform [71]. We mine repositories using the GHTorrent archive [37] that is hosted on Google Big Query. However, as publicly-available GitHub repositories are susceptible to quality issues [71], we apply the following filtering criteria: *Criterion-1:* Repository must be publicly available and contain the 'Kubernetes' label to ensure that the repositories are Kubernetes relevant [77]; *Criterion-2:* At least 10% of the files in the repository are YAML files and each file must use Kubernetes objects (e.g., Pod, Service, Deployment, etc) to collect repositories that contain sufficient amount of configuration scripts for analysis; *Criterion-3:* The repository is not a copy of another repository; and *Criterion-4:* The repository has at least ten contributors. We use a threshold of ten contributors to ensure a higher likelihood that the repository represents a more collaborative and active project, reducing the chances of including repositories used for academic or personal projects. Prior research [76] has also used the threshold of at least 10 contributors.

As shown in Table 1, we collect 185 OSS Kubernetes repositories from GitHub repositories. We clone the master branches of the 185 repositories. We provide attributes of the mined 185 repositories in Table 2. In all we collect 44,401 configuration scripts.

Step#2 - Mine Commits and Issue Reports from 185 OSS Kubernetes repositories: We download the 185 OSS Kubernetes repositories on March 2024 to conduct our analysis. From the downloaded repositories, we mine 417,598 commits and 140,872 issue reports. To identify commits and issue reports that are related to defects, we use the following steps: *Step-1:* We filter issue reports by checking if the issue is closed and has a pull request to ensure we have sufficient content to derive fix patterns; *Step-2:* We apply a keyword search similar to prior work [76]. We use following keywords: 'bug,' 'defect,' 'error,' 'fault,' 'fix,' 'flaw,' 'incorrect,' 'issue,' and 'mistake' to ensure commits and issue reports are related to a defect; *Step-3:* We inspect the files modified in each commit and issue report to ensure commits and issue reports are related to Kubernetes configuration management; and *Step-4:* We exclude commits that are duplicates of others. In all, we identify 66 commits and 1,941 issue reports that include defect-related keywords.

Step#3 - Detect Defects by Applying Qualitative Analysis: We conduct qualitative analysis to identify defects from defect-related commits and issue reports. The rationale is that relying solely on keyword search can result in false positives. To identify defects, we use the IEEE definition [47]: *"an imperfection or deficiency in the code that needs to be repaired."*

Criteria to Identify Defects - For defect identification, the rater applies the following criteria: (i) problematic code exists in the commit message or the issue report; (ii) problematic code leads to an incorrect or undesired consequence that is explicitly expressed by a practitioner; (iii) the commit message or issue content describes an immediate consequence of the defect; and (iv) the problematic code was repaired. By applying these criteria, we identify that 52 of the 66 commits and 681 of the 1,941 issue reports to be related with defects.

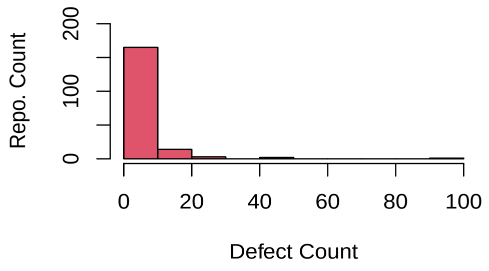

Fig. 3: Distribution of defects.

Criteria to Identify Configuration Defects - The rater inspects if any of the following criterion is satisfied: (i) the defect resides in a configuration script; (ii) the defect occurs when provisioning Kubernetes resources, or managing Kubernetes resources, or monitoring Kubernetes resources; and (iii) the defect is related to a Kubernetes configuration. Using this criteria, we identify 719 defects. Of these 719 defects 52 and 667 are respectively, obtained from 52 commits and 681 issue reports. Figure 3 shows the distribution of defects across 185 repositories. We observe 77.8% of the studied repositories include $=< 5$ defects.

3.1.2 Derive Defect Categories

We employ a qualitative analysis method known as open coding [81] to derive defect categories. Open coding involves recognizing patterns in unstructured text to establish categories [81]. The first and second author individually applies open coding with 52 defects from 52 defect-related commits and 667 defects from 667 defect-related issue reports. While applying open coding, each rater applies orthogonality, i.e., derive the categories so that do not overlap. Each rater examines messages and code changes for each commit, as well as title, description, comment, pull request, and code changes for each issue report.

Each rater creates a category with a short definition, which we use to identify and resolve differences in labeling. Disagreements occurred as the first and second author respec-

TABLE 1: Filtering of OSS Repositories

Initial Repository Count	14,747,836
Criterion-1 (Available and relevant)	1,410
Criterion-2 (>= 10% Configuration scripts)	1,087
Criterion-3 (Not a copy)	1,079
Criterion-4 (Contributors >=10)	185
Final Repository Count	185

TABLE 2: Dataset Attributes

Category	**Data**
Total Repositories	185
Total Commits	417,598
Total Developers	21,559
Kind Scripts	37,147
Helm Scripts	7,254
Total Kubernetes Scripts	44,401
Total Size (LOC)	51,282,124
Total Count of Issue Reports	140,872
Total Count of Stars	398,347
Time Span	06/2014 - 03/2024

tively, identified 20 and 12 categories. Here, 11 categories are identified by both raters, whereas 9 are identified by the first author, and 1 is identified by the second author. The Cohen's Kappa [23] is 0.67, suggesting 'substantial' agreement [63]. Amongst the 10 disagreements, 3 occurred because of naming issues, e.g., 'conditional operator' and 'conditionals,' and 7 occurred because of definition overlap, e.g., 'access control' is a sub-category of 'security.' The two raters discussed these disagreements but could not reach consensus on all cases. Therefore, the last author, with extensive experience in qualitative coding and defect classification, and who also is not involved in the initial coding, served as adjudicator to resolve the remaining disagreements. While resolving the disagreements, the last author examined the names, definitions, and code snippets for the defects of interest. Then, the last author mapped the defect to a category that has already been finalized.

Our defect categories is derived using open coding [81] where we merged sub-categories into categories. This merging process occurred because of similarities between derived subcategories. For example, the subcategory 'privileged ports' captures defects where containers are configured to bind to ports below 1024, while the subcategory 'access control' refers to defects that inadvertently grant excessive permissions. Although these defects occur in different configuration fields, they both represent security defects. Hence, we merged them into the broader 'security' category, which unifies defects that compromise confidentiality, integrity, or availability.

3.1.3 Scoping Review

We compare identified defect categories to those of previously studied software systems using a scoping review [9]. A scoping review is a variant of a systematic literature review conducted in a reduced scope [9]. Our review includes publications related to defect categorization published at the International Conference of Software Engineering (ICSE) and Foundations of Software Engineering (FSE) conferences from 2020 to 2024. We select ICSE and FSE due to their reputation in publishing software engineering research. Additionally, we include three defect categorization publications [21], [82], [95] and two Kubernetes-specific studies [53], [77]. In all, we use 21 publications, of which 9, 7, 1, 1, 1, 1 and 1 are respectively, from ICSE, FSE, TSE, EMSE, ESEM, TOSEM, and IWCMC.

3.2 Answer to RQ1

We answer RQ1 by reporting the defect categories and results related to scoping review (Section 3.2.2), results related to scoping review (Section 3.2.1), their frequency (Section 3.2.3).

3.2.1 Answer to RQ1: Defect Categories

We identify 15 defect categories, which we characterize with examples obtained from our OSS repositories.

```
-   {{- if and .Values.certificates.autoGenerated (
↪   not .Values.certificates.certManager.enabled ) }}
+   {{- if or (and .Values.certificates.autoGenerated
↪   (not .Values.certificates.certManager.enabled))
↪   (.Values.permissions.operator.restrict.secret) }}
    apiVersion: rbac.authorization.k8s.io/v1
```

Listing 3: Example of a conditional-related defect.

I **Conditional**: This kind of defect manifests when developers use incorrect operators or operands in conditional statements, such as if-else blocks. Listing 3 shows an example defect [33] where an improper operand, i.e., 'and' is used in an if-else block. Due to this defect, the application fails to startup.

II **Container Provisioning**: This defect category occurs when developers provision containers for pods. There are two sub-categories: (i) *Command Line Arguments (CLA)*: This defect category occurs due to specifying erroneous command line arguments. Arguments can be provided either from command line or using the `command` or `args` property. Listing 4 shows a CLA-related defect [67] where an erroneous argument is provided for `command`. Due to this defect, the image is unable to be recovered after being deleted. (ii) *Resources*: This defect occurs because of provisioning resources that are unspecified, under-specified, or over-specified. Listing 5 shows a resources-related defect [38] where resource limits are under-specified, i.e., 256 Mebibytes (Mi) is used instead of 550Mi. This defect causes a hang.

III **Custom Resource**: This defect category occurs when developers incorrectly manage custom resources (CRs) in Kubernetes. CRs are extensions of the Kubernetes API that allow developers to create and manage new kinds of resources beyond what Kubernetes offers by default [59]. Listing 6 shows an example defect [49] where an incorrect image tag is configured for the `ClusterServiceVersion` CR.

```
metadata:
  labels:
-   control-plane: controller-manager
+   control-plane: argocd-operator
```

Listing 7: A defect related to entity referencing.

IV **Data Fields**: This defect category occurs when the data fields are improperly handled in scripts. We identify five sub-categories: (i) *Base64 String and Encoding (BSE)*: This defect occurs due to the misuse of Base64 encoding. Figure 4a shows a BSE-related

```
-stringData:
-  username: "{{.vsphereUsername}}"
-  password: "{{.vspherePassword}}"
+data:
+  username: {{.vsphereUsername |
↪   b64enc}}
+  password: {{.vspherePassword |
↪   b64enc}}
```

a

```
-node: {{
↪   $sts.node }}
+node: {{
↪   $sts.node |
↪   quote }}
```

b

```
-path: {{ . }}
-pathType:
↪   ImplementationSpecific
+pathType: Prefix
```

c

```
-{{- toYaml
↪   .Values.volumes.keda.
↪   extraVolumeMounts |
↪   nindent 12 }}
+{{- toYaml
↪   .Values.volumes.keda.
↪   extraVolumeMounts |
↪   nindent 10 }}
```

d

```
volumeMounts:
-  - name: data-{{
↪   .Release.Namespace
↪   }}
+  - name: data-{{
↪   .Release.Namespace |
↪   trunc 58 |
↪   trimSuffix "-" }}
```

e

Fig. 4: Examples of defects related to data fields. Figures 4a, 4b, 4c, 4d, and 4e respectively, presents examples of defects related to BSE, IDT, IUPT, syntax, and VR.

```
command:
    ...
-   - longhornio/backing-image-manager:
↪   v2_20210820_patch2
+   - longhornio/backing-image-manager:v2_20221027
```

Listing 4: Example of a CLA-related defect.

```
 resources:
  limits:
-    memory: 256Mi
+    memory: 550Mi
```

Listing 5: Example of a resource-related defect.

```
kind: ClusterServiceVersion
...
    - name: OPERATOR_NAME
-     image: quay.io/jaegertracing/
↪   jaeger-operator:v1.29.0
+     image: quay.io/jaegertracing/
↪   jaeger-operator:1.29.0
```

Listing 6: Example of a CR-related defect.

defect [10] because of not using Base64 encoding with 'b64enc.' (ii) *Incorrect Data Types (IDT)*: This defect occurs because of using incorrect data types. Figure 4b shows an IDT-related defect [29] that causes a hang. The defect occurred because of missing `quote`, which causes `annotations` to be interpreted as numbers instead of strings. (iii) *Incorrect URL Path Types (IUPT)*: This defect occurs due to misuse of `pathType`, an attribute used to route incoming traffic to the backend services. Figure 4c shows an IUPT-related defect [108] which results in a dashboard failing to load. (iv) *Syntax*: This defect occurs due to syntax errors. Figure 4d shows a syntax-related defect [55] where incorrect indentation is used. (v) *Violation of Restrictions (VR)*: This defect occurs due to the failure to adhere to the specific technical rules and constraints enforced by Kubernetes on resource definitions and configurations. These restrictions include, but are not limited to, name length, allowed characters, and the correct format of values. Figure 4e shows a VR-related defect [41] where a dynamically generated `name` exceeds the maximum length of 63 characters.

V **Entity Referencing**: This defect category occurs when Kubernetes entities, such as names and labels are incorrectly referenced or entities that are referred to do not exist. Listing 7 shows an example defect [8] where the incorrect label 'controller-manager' is provided instead of 'argocd-operator.'

```
containers:
    - name: main
-     image: docker.io/aquasec/trivy:0.34.0
+     image: "{{ .Values.trivy.repository }}:{{
↪   .Values.trivy.tag }}"
```

Listing 8: A defect related to incorrect Helming.

VI **Incorrect Helming**: This defect category occurs when users hard-code configuration values in templates. Hard-coding configuration values in templates is considered as an anti-pattern in Kubernetes [17]. Listing 8 shows an example defect [4], where the configuration value is hard-coded in a template. Due to this defect, the user-provided image value is never applied.

```
  subjects:
-   namespace: {{ .Release.Namespace }}
+   namespace: {{ include
↪   "opentelemetry-collector.namespace" . }}
```

Listing 9: Example of a namespace-related defect.

VII **Namespaces**: This defect category occurs when an incorrect namespace is used. Namespaces provide a mechanism for isolating groups of resources within a single Kubernetes cluster by separating different environments [59]. If resources are placed in a namespace that is different from the objects that use them, then the referencing objects will not be able to access these resources, leading to application failures. Listing 9 shows an example defect where the namespace is incorrect due to an incorrect template directive. As a result, the deployed application of interest results in a crash.

```
-   kind: ClusterRole
-   name: argocd-server
-   ...
 ---
-   kind: ClusterRoleBinding
-   name: argocd-server
-   ...
 ---
 kustomization.yaml:
    resources:
-   - argocd-server-clusterrole.yaml
-   - argocd-server-clusterrolebinding.yaml
+   - ./application-controller
```

Listing 10: A defect related to orphanism.

VIII **Orphanism**: This defect category occurs when either resources in a pod are not properly de-allocated, or when resources are deployed but not referenced by any other resources. Listing 10 shows an example defect [7] where a ClusterRoleBinding object references a non-existent service account, i.e., 'argocd-server.' This defect leads to a resource leak, as the orphaned `ClusterRole` and `ClusterRoleBinding` continue to consume cluster resources unnecessarily.

```
-    affinity: {}
+    affinity:
+       nodeAffinity:
```

Listing 11: A defect related to pod scheduling.

IX **Pod Scheduling**: This defect category occurs when developers incorrectly use pod scheduling mechanisms, such as affinity. Affinity is a set of rules that assign pods to nodes based on certain criteria, such as node labels or the location of other pods [59]. Listing 11 shows an example defect [61]

```
rules:
  apiGroups:
-   - "*"
+   - ""
+   -
↪   events.k8s.io
```

a

```
args:
  - --cert-dir=/tmp
- - --secure-port=443
+ -
↪   --secure-port=4443
```

b

```
data:
- {{- if eq (typeOf .Values.alertmanager.config) "string" }}
+ {{- if .Values.alertmanager.stringConfig }}
+ alertmanager.yaml: {{ tpl (.Values.alertmanager.stringConfig) |
↪   b64enc | quote }}
+ {{- else if eq (typeOf .Values.alertmanager.config) "string" }}
+ alertmanager.yaml: {{ tpl (.Values.alertmanager.config) | b64enc |
↪   quote }}
```

c

```
securityContext:
- runAsNonRoot: true
+ runAsUser: 65534
```

d

Fig. 5: Examples of security defects. Figures 5a, 5c, 5b, and 5d respectively, presents examples of defects related to AC, PP, ESD, and SC.

where affinity is missing. This causes a pod to be unexpectedly scheduled on the 'fargate' node, leading to resource contention between pods.

X **Probing**: This defect category that occurs when probing is incorrectly handled in scripts. Kubernetes provides two health check probes namely, liveness probes and readiness probes to monitor the health status of the provisioned containers [59]. Listing 12 shows an example defect [6] where the configurations for a liveness probe is missing. Due to this defect, the pod could not automatically recover from an error status when a failure occurred, leading to an outage.

```
-     tcpSocket:
+   livenessProbe:
+     httpGet:
+       path: /healthz
```

Listing 12: Example of a probing-related defect.

XI **Property Annotation**: This defect category occurs when developers use user-defined annotations incorrectly. Annotations are used to attach arbitrary non-identifying metadata to objects [59]. Unlike labels, which are used to organize and select subsets of objects, annotations are not used to identify and select objects. Instead, they are used to store additional information that may be used by external libraries. Listing 13 shows an example defect [56] because of incorrectly using the `kserve/serving-cert` annotation.

```
annotations:
-
↪   "cert-manager.io/inject-ca-from":
↪   kserve/serving-cert
+
↪   "cert-manager.io/inject-ca-from":
↪   kubeflow/serving-cert
```

Listing 13: A defect related to property annotation.

XII **Security**: This category includes defects that violate the principle of confidentiality, integrity, or availability. The four sub-categories are: (i) *Access Control (AC)*: Access control is defined as the technique that regulates who or what can view or use resources in a computing environment. If access control is improperly configured, it can lead to creation of over-privileged and under-privileged entities. Over-privileged entities, such as users or processes may perform unauthorized actions, access sensitive data, or disrupt the operation of the system [31]. Under-privileged entities can lead to availability issues, such as not being able to access needed cluster data. Figure 5a shows an AC-related defect [62] that occurred because of using '*' that allows unauthorized users to gain access to sensitive data. (ii) *Privileged Ports (PP)*: This defect occurs due to the use of a privileged port number. Using privileged ports that are typically below 1024 requires higher privileges, which can increase security risks, such as privilege escalation, if not properly managed [5]. Privilege escalation can expose the system to attacks, as they may require running applications or containers with more access than necessary allowing a malicious user to gain unauthorized control [74]. Figure 5b shows a PP-related defect [60] where a privileged port number 433 is used. (iii) *Exposure of Sensitive Data (ESD)*: This defect occurs due to exposure of sensitive data in scripts. Figure 5c shows an ESD-related defect [73] where a plain string can be mistakenly passed to the `Secret` entity. (iv) *Security Context (SC)*: This defect that occurs due to privileged `securityContext` or a missing `securityContext`. A `securityContext` is a Kubernetes entity that determines the user IDs, group IDs, and whether the container runs as a privileged user. An improperly configured `securityContext` can result in containers running with unnecessary privileges, increasing the risk of privilege escalation, unauthorized access, and potential compromise of the system [66]. Figure 5d shows a SC-related defect [65] where `'runAsUser'` is missing for `securityContext`, which causes the container running with root privileges. Due to this defect, malicious users could gain unauthorized access.

XIII **Unsatisfied Dependency**: This defect category occurs when execution of scripts are dependent on one or multiple prerequisites, such as network-related dependencies and container images. Listing 14 shows an example defect [48] where scaling up pods on different nodes fails due to the missing precondition `ReadWriteMany`. The `ReadWriteMany` access mode in the persistent volume claim configuration allows multiple nodes to read and write simultaneously, which is a crucial precondition for scaling up pods across different nodes.

```
accessModes:
-   - ReadWriteOnce
+   - {{ .Values.accessMode }}
  ---
  values.yaml:
+ accessMode: ReadWriteMany
```

Listing 14: A defect related to unsatisfied dependency.

XIV **Version Incompatibility**: This defect category occurs when developers use APIs or Kubernetes objects that are no longer supported by Kubernetes and its API. Listing 15 shows an example defect [1] where a deprecated API version `'extensions/v1beta1'` is used. Due to this defect, the configu-

```
-   apiVersion:
↪   extensions/v1beta1}
+   apiVersion: apps/v1
    kind: Deployment
```

Listing 15: A defect related to version incompatibility.

1. https://github.com/SeldonIO/seldon-core/issues/3677

TABLE 3: A Comparison Between Identified Defect Categories and Defect Categories for Previously-Studied Software Systems

Previously-studied Software System	Conditio-nal	Container Provi-sioning	Custom Re-source	Data Fields	Entity Refer-encing	Incorrect Helm-ing	Namespa	Orphanis	Pod Schedul-ing	Probing	Property Annota-tion	Security	Unsatisfied Depen-dency	Version Incom-patibil-ity	Volume Mount-ing
Android Applications [98]	✓	–	–	✓	–	–	–	–	–	–	–	–	–	–	–
Autopilot Software [97]	✓	–	–	✓	–	–	–	–	–	–	–	–	–	–	–
Deep Learning Compiler [88]	✓	–	–	✓	–	–	–	–	–	–	–	–	–	✓	–
Deep Learning Deployment (Mobile) [19]	–	✓	–	✓	–	–	–	–	–	–	–	–	✓	–	–
Deep Learning Stack [44]	–	–	–	–	–	–	–	–	–	–	–	–	✓	✓	–
Federated Learning Systems [32]	✓	✓	–	✓	–	–	–	–	–	–	–	✓	✓	✓	–
IBM Proprietary Software [21]	✓	–	–	✓	–	–	–	–	–	–	–	–	–	–	–
IaC State [42]	✓	-	–	✓	–	–	–	–	–	–	–	✓	✓	–	–
IaC Defect [76]	✓	✓	–	✓	–	–	–	–	–	–	–	✓	✓	–	–
Linux Kernel [95]	✓	–	–	–	✓	–	–	–	–	–	–	✓	–	–	–
NASA Software Projects [82]	✓	–	–	✓	–	–	–	–	–	–	–	–	–	–	–
Machine Learning Model [40]	–	–	–	✓	–	–	–	–	–	–	–	–	–	–	–
IoT [68]	–	–	–	–	–	–	–	–	–	–	–	–	✓	✓	–
Database Systems [26]	–	–	–	–	-	–	–	–	–	–	–	-	✓	–	–
Web Server Systems [100]	–	–	–	–	✓	–	–	–	–	–	–	✓	✓	–	–
Multilingual Python Projects [104]	✓	–	–	–	–	–	–	–	–	–	–	✓	–	–	–
WeChat [99]	✓	–	–	✓	–	–	–	–	–	–	–	–	–	✓	–
Static Analyzers [43]	✓	–	–	–	–	–	–	–	–	–	–	–	–	–	–
SLIKUBE [77]	–	✓	–	–	–	–	–	–	–	–	–	✓	–	–	–
Kubernetes Security Deployments [53]	–	✓	–	–	–	–	–	–	–	✓	–	✓	–	–	–
Faults in Deep Learning [45]	-	–	–	-	–	–	–	–	–	–	–	–	–	✓	–

ration script fails to be executed and leads to a crash.

XV **Volume Mounting**: This defect category occurs when developers incorrectly mount storage for applications that are managed by Kubernetes. Listing 16 shows an example defect [2] where the 'zk-data' volume is incorrectly mounted instead of 'zk-datalog.' This defect resulted in a crash.

```
  - mountPath: /datalog
-     name: zk-data
+     name: zk-datalog
```

Listing 16: A defect related to volume mounting.

3.2.2 *Comparison with Defect Categories for Previously-studied Software Systems*

In Table 3 we report the defect categories that have appeared for other software systems. In the table, a '✓' indicates that the defect category is reported in previously-studied software systems, while a '–' denotes that the category is not reported. As highlighted in green, 7 are unique to Kubernetes: custom resource, incorrect Helming, namespaces, orphanism, pod scheduling, property annotation, and volume mounting. While the previous section introduces and defines each defect category, this comparison explains why these seven categories are specific to Kubernetes and absent from other software systems. In particular, these categories focus solely on defects in YAML-based configuration files, which do not overlap with prior work on Kubernetes operators and therefore are not renamed variants of operator-based defects. Unlike, operators that are custom controllers [59], configuration scripts are used to configure built-in Kubernetes entities, such as pods, custom resources, and namespaces. Custom resources in Kubernetes are used to extend the Kubernetes API by allowing developers to define and manage their own custom resource types through Custom Resource Definitions (CRDs). Incorrect Helming is unique to Kubernetes as it stems from Helm-specific template misuse, a defect type not applicable to other software systems. Namespaces in Kubernetes are used to isolating resources by providing logical separation within a cluster. Orphanism is unique to pods, where resources of pods, such as CPU and memory are left unused or unlinked due to improper cleanup or misconfiguration. Pod scheduling is unique to Kubernetes pods, which is conducted when the scheduler assigns pods to appropriate nodes based on resource availability, constraints, and policies. Property annotation is performed for Kubernetes resources to provide metadata or configuration details, such as specifying labels, or custom behaviors. Volume mounting is applicable for Kubernetes pods, where the `volume` tag is used to define the storage volumes and attach them to containers.

Our unique Kubernetes-specific categories, such as incorrect Helming and orphanism, are valuable because they capture defects unique to YAML-based configurations that prior publications have not addressed. The identified defect categories provide a foundation for tool builders to design targeted analysis techniques and for practitioners to more effectively detect and fix Kubernetes configuration defects.

3.2.3 *Results for RQ1: Frequency*

We present the count of defects for each defect category in Table 4, organized alphabetically by category names. The most frequently occurring category is entity referencing. '-' denotes categories without sub-categories. 'Category Total' represents the overall count of defects for categories with sub-categories. The frequency distribution shows that certain categories are more frequent than others. For example entity referencing and unsatisfied dependency are more frequent than orphanism or property annotation. These frequently occurring categories involve names, labels, and references to other resources, which are essential for identifying objects and linking related resources, such as pods to services. References that do not match expected names or labels can break these connections, explaining the high frequency of defects in entity referencing and unsatisfied dependency. In contrast, orphanism arises only when resources in a pod are not properly de-allocated or remain unreferenced. Because orphaned resources typically impact performance rather than core functionality, and a small number of orphaned resources often has limited observable

2. https://github.com/apache/openwhisk-deploy-kube/commit/720abadb5249eb96d5f27afd1cc21387ab85652d

TABLE 4: Answer to RQ1: Frequency of Defect Categories.

Category	Sub-category	Count
Conditional	-	40
Container Provisioning	Command Line Arguments	43
	Resources	9
	Category Total	52
Custom Resource	-	46
Data Fields	Base64 String and Encoding	2
	Incorrect Data Types	19
	Incorrect URL Path Types	1
	Syntax	35
	Violation of Restrictions	30
	Category Total	87
Entity Referencing	-	125
Incorrect Helming	-	13
Namespaces	-	15
Orphanism	-	10
Pod Scheduling	-	12
Probing	-	22
Property Annotation	-	12
Security	Access Control	76
	Exposure of Sensitive Data	4
	Privileged Ports	1
	Security Context	11
	Category Total	92
Unsatisfied Dependency	-	105
Version Incompatibility	-	58
Volume Mounting	-	30

impact, such defects are reported less frequently. Property annotations are optional metadata and are configured less frequently than names and labels, which leads to fewer opportunities for defects to occur. Overall, this distribution indicates that practitioners encounter difficulties with managing inter-resource references, rather than with individual resource definitions.

***Answer to RQ1**: We identify 15 defect categories, of which 7 have not been reported for previously-studied software systems: custom resource, incorrect Helming, namespaces, orphanism, pod scheduling, property annotation, and volume mounting. The most frequent category is entity referencing.*

4 RQ2: Consequences and Fix Patterns

We provide the methodology and results for RQ2 respectively, in Sections 4.1 and 4.2.

4.1 Methodology

In this section, we describe the methodology on how we derive the consequences and fix patterns.

4.1.1 Deriving Consequences

We analyze commit messages and content in issue reports for the identified 719 defects to determine the consequences using open coding [81]. The first and last author conduct open coding separately. Each rater applies the following steps: (i) separate commits/issues labeled as defects identified from Section 3.1.2; (ii) read text in commit messages and issue report titles/body; (iii) separate text that expresses consequences; and (iv) categorize consequences based on commonality, e.g., two issue reports [3] express an outage-related consequence.

Initially, the first and last author respectively, identify 17 and 12 consequences. The Cohen's Kappa [23] is 0.53, suggesting a 'moderate' agreement [63]. The disagreement results from the last author's opinion of finding 5 consequences that are synonymous with consequences identified by the first author. In order to resolve the disagreements, we use a rater who is not the author on the paper for rater verification. The voluntary rater uses his judgment to resolve the disagreements. This rater is a third-year PhD student in the department. Each identified defect category from Section 3.2 maps to one of the 12 consequences.

4.1.2 Deriving Fix Patterns

We apply a qualitative analysis technique called open coding [81] similar to prior research [52], [115]. The first and last author individually apply the following steps: (i) separate issues labeled as defects from Section 3.1 that have code changes; (ii) read the code that was changed for each defect; (iii) identify commonalities in the changes and create groups based on commonalities; and (iv) merge groups into fix pattern categories. Each rater uses messages and code changes from commits as well as from issue reports to apply the above-mentioned steps.

Upon applying open coding, the first and last author respectively, identify 12 and 8 fix pattern categories. The authors disagree on 3 categories. Upon discussion, the 8 categories identified by both raters and one category identified by the first author that was not identified by the last author were added. The Cohen's Kappa [23] is 0.81, suggesting a 'substantial' agreement [63].

4.2 Answer to RQ2

We provide answers to RQ2 in this section.

4.2.1 Answer to RQ2: Consequences

We identify 12 consequences, definitions of which are provided in Table 5. A mapping between the identified defect categories and the consequences is provided in Table 6. We observe the most frequently occurring consequence to be incorrect operations (InOp). We observe 52 defects to be related to configuration inexecutability that does not lead to crashes and hangs but keep the Kubernetes cluster running with incorrect configurations. These consequences show how serious Kubernetes configuration defects are and highlight the importance of our empirical study.

4.2.2 Results for RQ2: Fix Patterns

We identify 9 fix patterns, definitions of which are provided in Table 7. A mapping between the identified defect categories and the fix patterns is provided in Table 8. The most frequently occurring fix pattern to be configuration value changes (CVC).

3. https://github.com/argoproj/argo-cd/issues/10249, https://github.com/Azure/application-gateway-kubernetes-ingress/issues/67

TABLE 5: Results for RQ2: Consequences and their definitions.

Consequence	Definition
Compiler Warning (CW)	The consequence of obtaining warning messages from the compilation engine.
Configuration Inexecutability (CI)	The consequence of running the Kubernetes cluster with incorrect configurations. In this case, configurations specified in scripts are not executed or are overridden.
Crash	The consequence of a Kubernetes operation being terminated abruptly.
Diagnose Inability (DI)	The consequence of not being able to diagnose failures or crashes.
Exposure of Unauthorized Data (EUD)	The consequence when unauthorized users get access to data.
Hang	The consequence when an operation is unresponsive.
Incorrect Artifact Generation (IAG)	The consequence of generating an artifact incorrectly because of a defect.
Incorrect Operations (InOp)	The consequence when Kubernetes-related operations are executed incorrectly.
Incorrect Rendering (IR)	The consequence of generating an incorrect display for the Kubernetes dashboard.
Outage	The consequence when a Kubernetes object is unavailable when requested by users.
Performance	The consequence of incurring unexpected usage of CPU and memory.
Unpredictable Responses (UR)	The consequence of providing unpredictable responses to the user, such as conducting unpredictable routing of traffic and obtaining unpredictable responses from pods.

***Answer to RQ2**: We identify 12 consequences, with incorrect operations being the most common. We observe 52 defects mapped to configuration inexecutability, where clusters continue running with incorrect configurations but without crashing. We also identify 9 fix patterns, with configuration value changes being the most frequent.*

5 RQ3: Evaluation of Static Analysis Tools for Detecting Defects

We organize this section by answering two sub-questions:

- RQ3.a: What categories of Kubernetes-related defects are supported by static analysis tools?
- RQ3.b: How can we detect defects that are not supported by existing static analysis tools?

We evaluate static analysis tools in our paper. We do not evaluate dynamic analysis tools, such as 'Kube-hunter' [4] and 'BotKube' [5] as these tools rely on logs that are generated from a Kubernetes cluster at runtime. Therefore, evaluation of these dynamic analysis tools require execution of configuration scripts, which in turn is dependent on correct inference of computing environments [69]. Setting up these environments correctly require adequate installation of all artifacts specified as dependencies for each of the 185 repositories, which makes the evaluation of dynamic analysis tools unfeasible.

4. https://github.com/aquasecurity/kube-hunter
5. https://botkube.io/

In order to conduct evaluation, we use the curated dataset described in Section 3.1. This dataset is informed by: (i) real-worlds defects confirmed by practitioners; and (ii) manual verification by the raters. An alert reported by a tool that does not exist in the dataset is a false positive. Any defect included in the dataset but missed by the tool is a false negative. Our approach is consistent with prior research [64] that conducted tool evaluation using curated datasets.

5.1 RQ3.a: Defect Categories Supported by Static Analysis Tools

5.1.1 Methodology

We use the following steps to answer RQ3.a:

5.1.1.1 *Selection of Static Analysis Tools*: We start the selection process using the Google search engine in incognito mode with the search string 'defect detection tools for kubernetes.' From the collected top 100 search results, we identify 100 tools for Kubernetes. Next, we apply the following criteria: *Criterion-1:* The tool must be publicly available for use. *Criterion-2:* The tool must be able to detect defects using static analysis. The first author of the paper read the documentation of each tool to determine if the tool can detect defects in configuration scripts. *Criterion-3:* The tool must support execution through the command line interface, allowing for automated execution. *Criterion-4:* The tool must be capable of detecting at least one of the 15 identified defect categories. This ensures that each tool contributes to the overall coverage of defect detection. The first author reads the documentation of each tool to apply this criterion. By applying Criterion-1, 2, and 3, we respectively, identify 23, 20, and 8 tools. From our application of the four criteria we identify eight tools. Attributes of these tools are available in Table 9. Each of the 8 tools was applied on 2,260 scripts using the command line. For example, 'Kubeconform' was executed using 'kubeconform $< file\_path >$.' The process took 9.75 hours in total, averaging 1.2 hours per tool.

As this tool selection process is subjective, we allocate another rater during the revision of the paper. The other rater is the last author of the paper who apply the same steps as the second author where they read the documentation and source code of each tool. Upon completion of the process, we observe a Cohen's Kappa [23] of 1.0 indicating 'Perfect' agreement [63].

5.1.1.2 *Evaluation of Static Analysis Tools*: We use two evaluation activities:

Activity-1: Evaluation based on support: For this evaluation, we conduct a mapping between each identified defect category to a detection rule used by each of the eight tools. The first and second authors independently apply closed coding [81], where they read the documentation and source code of each tool to perform this mapping. A mapping exists if a rule matches the definition of a defect category. Upon completion of the closed coding process, the Cohen's Kappa is 0.95 [23], indicating an 'almost perfect' agreement [63]. Disagreements arose for 10 rules because one of the raters

TABLE 6: Answer to RQ2: Frequency of consequences. '-' means zero defects map to that consequence.

Defect Category	CW	CI	Crash	DI	EUD	Hang	IAG	InOp	IR	Outage	Performance	UR
Conditional	-	9	13	-	-	1	9	1	-	6	-	1
Container Provisioning	-	7	4	1	-	2	1	8	-	24	4	1
Custom Resource	-	3	13	6	-	1	2	6	-	13	2	-
Data Fields	1	3	57	-	-	3	-	6	-	15	2	-
Entity Referencing	-	19	34	5	-	1	2	25	2	26	7	4
Incorrect Helming	-	8	1	-	-	-	-	-	-	3	-	1
Namespaces	-	-	1	-	-	-	-	11	-	3	-	-
Orphanism	-	-	1	-	-	-	-	2	-	-	7	-
Pod Scheduling	-	1	1	-	-	-	-	3	-	4	3	-
Probing	-	1	3	-	-	-	-	-	-	12	2	4
Property Annotation	-	-	2	2	-	1	-	3	-	3	1	-
Security	1	-	2	4	9	1	-	63	-	12	-	-
Unsatisfied Dependency	-	1	6	7	-	6	-	46	-	29	7	3
Version Incompatibility	7	-	17	2	-	1	1	13	-	15	-	2
Volume Mounting	1	-	6	2	-	1	-	7	-	13	-	-
Total	10	52	161	29	9	18	15	194	2	178	35	16

TABLE 7: Answer to RQ2: Fix patterns, their definitions, and examples.

Fix Pattern	Definition	Example Code Snippet

Adding Conditional Statements (ACS)	Adding missing or incorrect conditional statements in templates.	`- {{- if .Values.extraVolumeTags }}`
		`+ {{- if or .Values.controller.extraVolumeTags`
		`↪   .Values.extraVolumeTags }}`

Configuration Value Changes (CVC)	Changing configuration values to correct or updated values.	`ports:`
		`-   - port: 443`
		`+   - port: 8443`

Directive Fix (DF)	Fixing a template directive to correctly populate configuration.	`- value: {{ .Values.checkReaper.maxPodsThreshold`
		`↪   }}`
		`+ value: {{ .Values.checkReaper.maxPodsThreshold`
		`↪   | toString }}`

Environment Variable Fix (EVF)	Changing environment variables used at container runtime.	`env:`
		`- - name: CONSUL_HTTP_TOKEN`
		`+ - name: CONSUL_ACL_TOKEN`

Object Modification (OM)	Adding or deleting Kubernetes objects.	`+ apiVersion: rbac.authorization.k8s.io/v1`
		`+ kind: ClusterRoleBinding`

Property Modification (PM)	Adding or deleting properties of a Kubernetes object.	`readinessProbe:`
		`+ httpGet:`
		`+   path: /healthz`
		`+   port: 8082`
		`- tcpSocket:`
		`-   port: 8082`

Relocation	Relocating objects, paths, or properties to correct places.	`- mountPath: /usr/bin`
		`+ mountPath: /usr/local/mount-from-host/bin`

Rule Fix (RF)	Fixing rules for access control policies.	`rules:`
		`- resources: ["replicasets"]`
		`+ resources: ["replicasets", "daemonsets",`
		`↪   "deployments", "statefulsets"]`

Syntax Fix (SF)	Correcting YAML syntax errors.	`labels:`
		`- cluster.x-k8s.io/aggregate-to-manager: true`
		`+ cluster.x-k8s.io/aggregate-to-manager: "true"`

was less familiar with specific Kubernetes configuration concepts. For example, a rule for unsafe `sysctls` should have been mapped to the 'security' category, but because of lack of familiarity it was not mapped by the rater. All disagreements are discussed and resolved collaboratively.

Activity-2: Evaluation based on detection accuracy: Using precision and recall, we compute the detection accuracy of the generated alerts, i.e., the detection results obtained from each tool. Precision is calculated as $\frac{\text{TP}}{\text{TP+FP}}$. Recall is calculated as $\frac{\text{TP}}{\text{TP+FN}}$. Here, TP corresponds to the number of alerts that are true positives, i.e., actual defects, FP corresponds to the number of false positive alerts, and FN corresponds to the number of missed actual defects. We determine an alert to be a TP if the alert correctly identifies a defect for the same category, same configuration script, same location, and same coding pattern for the defect of interest. We determine an alert to be a FP if the alert incorrectly identifies a defect belonging to an incorrect category, or incorrect script, or incorrect location, or for the incorrect coding pattern. We determine FN for a defect, if a tool does not report an alert for it. In order to determine TP, FP, and FN we use the dataset that we construct for answering RQ1. We do not include any defects that is not present in our dataset used for categorization. We repeat the process for calculating TP, FP, and FN for all defect categories.

Activity-3: Investigating Tools' Capabilities Beyond Our Defect Taxonomy: While the studied tools may not accurately detect our identified categories, they can still be useful for detecting coding patterns related to the validation of configuration scripts. We investigate this aspect as part of this activity using the following steps:

1) First, we apply the tools on a set of 2,260 configuration scripts and identify the categories of coding patterns that can be helpful for validation of scripts but not included in our taxonomy. Here, we examine the rules from each tool to determine whether they match the definitions of our taxonomy categories. Rules that do not match are

TABLE 8: Answer to RQ2: Frequency of fix patterns. '-' means no defects map to the fix pattern.

Defect Category	ACS	CVC	DF	EVF	OM	PM	Relocation	RF	SF
Conditional	13	-	27	-	-	-	-	-	-
Container Provisioning	1	32	3	9	1	4	1	1	-
Custom Resource	3	9	3	-	2	29	-	-	-
Data Fields	4	8	23	1	4	12	1	-	34
Entity Referencing	10	58	41	2	-	7	4	3	-
Incorrect Helming	-	-	9	1	-	3	-	-	-
Namespaces	-	2	3	1	1	6	-	2	-
Orphanism	-	3	1	-	5	-	-	1	-
Pod Scheduling	2	-	1	-	-	9	-	-	-
Probing	1	6	-	-	-	15	-	-	-
Property Annotation	2	6	1	-	-	3	-	-	-
Security	1	4	3	-	2	11	-	71	-
Unsatisfied Dependency	11	11	2	5	16	25	2	33	-
Version Incompatibility	14	32	5	-	1	4	2	-	-
Volume Mounting	4	3	1	-	-	20	2	-	-
Total	66	174	123	19	32	148	12	111	34

TABLE 9: Descriptions and Attributes of Selected Static Analysis Tools

Tool	Description	Source	Output Format	Size(LOC)	Technique
Checkov	A tool that can scan configurations used in cloud infrastructure. It supports over 1,000 checks related to security and compliance.	GitHub [18]	SARIF, Text, JSON, XML, CSV, Markdown	695,709	Policy-as-code [14]
Datree	A tool designed to secure Kubernetes workloads. It focuses on workload security, resource management, and best practices.	GitHub [28]	SARIF, JSON, XML, Text	31,193	Rule-based analysis [101]
Kube-Score	A tool designed to analyze Kubernetes object definitions. It investigates Kubernetes resources and provides recommendations to enhance the resilience of applications.	GitHub [109]	SARIF, JSON, JUnit, Text, CI	17,054	Rule-based analysis [101]
KubeLinter	A tool developed that identifies security defects and deviations from recommended practices. KubeLinter is perceived as the most popular static security analysis tool.	GitHub [92]	SARIF, JSON, Text	24,295	Rule-based analysis [101]
Kubesec	A security-focused static analysis tool that identifies potential security weaknesses in configuration scripts. It assigns a security score to Kubernetes resources based on their configuration.	GitHub [24]	JSON, YAML, Text	9,919	Rule-based analysis [101]
Kube-conform	A tool that validates scripts with OpenAPI and JSON schemas, ensuring they comply with expected standards.	GitHub [106]	JSON, XML, Text, TAP	639,910	Schema validation [30]
SLI-KUBE	A tool developed by researchers that identifies 11 categories of security weaknesses in scripts. It can be executed from the command line and is available as a Docker image.	TOSEM'23 [77]	SARIF, CSV	10,987	Rule-based analysis [101]
Yamlint	A tool that checks for syntax validity and adherence to best practices, including key repetition and syntax issues, such as trailing spaces.	GitHub [1]	Text	11,535	Pattern-based Analysis [34]

classified into new categories that are not included in our taxonomy. The classification process is performed independently by the first and last author using closed coding [81], where both raters read the documentation and source code of the tools to understand the intent of each unmapped rule. A rule is classified into a category if the rule matches the definition of one of the categories that have not been included in our taxonomy. Upon completion, the Cohen's Kappa is 0.91 [23], indicating an 'almost perfect' agreement [63]. The two raters disagree on 10 classifications, which are resolved by consensus.

2) Second, we run the eight tools on a random sample of 329 configuration scripts, a sample size that corresponds to a 95% confidence level. We selected the 95% confidence level because it is a commonly-used standard in empirical software engineering and statistical analysis [86], [114]. Finally, we count the frequency of alerts generated by the unmapped rules and report the number of detected defects in each category that have not been included in our taxonomy across the eight tools.

5.1.2 Results for RQ3.a

Our results are:

5.1.2.1 ***Results Related to Support***: We find eight defect categories to be supported by at least one tool. The defect categories for which we observe no support are: conditional, CR, incorrect Helming, orphanism, property annotation, unsatisfied dependency, and volume mounting. A full breakdown is available in Table 10, which is organized alphabetically by category names. In the table, a '✓' indicates that the tool can detect the category, while a '-' denotes that the tool cannot detect the category.

5.1.2.2 ***Results related to Detection Accuracy***: We observe the average precision and recall to be ≤ 0.28 for all eight tools. The highest precision is observed for Datree and Kubesec respectively, for syntax and incorrect data types (IDT), which are sub-categories of data fields. The highest

TABLE 10: Answer to RQ3.a: Support for Detecting Defects in Kubernetes Configuration Management.

Category	Sub-category	Checkov	Datree	Kube-conform	Kube-Linter	Kube-Score	Kubesec	SLI-KUBE	Yaml-lint
Conditional	N/A	-	-	-	-	-	-	-	-
Container Provisioning	CLA	-	-	-	-	-	-	-	-
	Resources	✓	✓	-	✓	✓	✓	✓	-
Custom Resource	N/A	-	-	-	-	-	-	-	-
Data Fields	BSE	-	-	-	-	-	-	-	-
	IDT	-	✓	✓	-	-	✓	-	-
	IUPT	-	-	-	-	-	-	-	-
	Syntax	-	-	-	-	-	-	-	✓
	VR	-	✓	✓	✓	✓	✓	-	-
Entity Referencing	N/A	✓	✓	-	✓	✓	✓	-	-
Incorrect Helming	N/A	-	-	-	-	-	-	-	-
Namespaces	N/A	✓	✓	-	✓	-	-	✓	-
Orphanism	N/A	-	-	-	-	-	-	-	-
Pod Scheduling	N/A	-	✓	-	✓	✓	-	-	-
Probing	N/A	✓	✓	-	✓	✓	-	-	-
Property Annotation	N/A	-	-	-	-	-	-	-	-
Security	AC	✓	✓	-	✓	-	✓	-	-
	ESD	✓	✓	-	✓	-	-	✓	-
	PP	-	-	-	-	-	-	-	-
	SC	✓	✓	-	✓	✓	✓	✓	-
Unsatisfied Dependency	N/A	-	-	-	-	-	-	-	-
Version Incompatibility	N/A	-	✓	-	✓	✓	-	-	-
Volume Mounting	N/A	-	-	-	-	-	-	-	-

recall is observed for Yamllint in the case of detecting defects-related to syntax. The worst performing tool is SLI-KUBE as its precision and recall are 0.0 for all categories. Data related to all tools and defect categories are available in Table 11. The '#' column represents the count of defects for each category and sub-category.

5.1.2.3 ***Results Related to Tools' Capabilities Beyond Our Defect Taxonomy***: We identify 6 categories of coding patterns that are related to validation identified by the studied tools but not included in our taxonomy: violations of best practice, built-in features, control plane configuration, broken isolation, missing availability safeguards, and tool setup. The results are summarized in Table 12. The '#' column reports the count of alerts that we classified into each category. A '-' indicates that the tool does not contain rules for that category.

The most frequent category that has not been included in our taxonomy is violations of best practice. The high frequency of alerts arises because of certain tools' emphasis on detecting violations of best practice. One such example is Checkov. For example, Checkov includes rule `CKV2_K8S_6`, 'minimize the admission of pods which lack an associated NetworkPolicy,' which represents a hardening recommendation rather than a configuration defect directly observed in our dataset. Missing availability safeguards and broken isolation are the next most frequent categories, with alerts reported by KubeScore, Checkov, and KubeLinter. Kubeconform and Yamllint do not provide rules that map to any categories that have been included in our taxonomy. In contrast, Checkov and Datree together provide coverage for every category not included in our taxonomy. This highlights that while some tools specialize in schema or syntax validation, others emphasize on best practice enforcement. Overall, these results show while the tools may not detect all defects in our dataset accurately, they can be for practitioners with respect to detecting violations of best practice, detecting missing safeguards for availability, and isolation issues that are important for Kubernetes deployments.

Answer to RQ3.a: *8 categories are supported by at least one tool, while 7 have no support: conditional, custom resource, incorrect Helming, orphanism, property annotation, unsatisfied dependency, and volume mounting. Average precision and recall are* ≤ 0.28 *across all tools.*

5.2 RQ3.b: Defect Detection with ConShifu

We provide the methodology and results for RQ3.b respectively, in Sections 5.2.1 and 5.2.2.

5.2.1 Methodology

Answers to RQ3.a show that there are seven categories of defects that are not covered by any tool. Of these seven categories, incorrect Helming and orphanism can be detected using static analysis. Detection of these two categories of defects is important as these defects can cause crashes and outages, as shown in Table 6. We hypothesize that by leveraging coding patterns from existing defects related to these two categories, we can develop a linter for defect detection. Accordingly, we construct 'ConShifu' [6] using the following steps:

Step#1 - Parsing: ConShifu takes one or multiple configuration scripts as input. Each script is parsed into key-value pairs where the hierarchies of keys are preserved. ConShifu is capable of analyzing Kind and Helm scripts. Upon completion of parsing, ConShifu stores the output in the forms of key-value pairs in JSON files.

Step#2 - Rule Matching: After parsing is complete, ConShifu applies rule matching to identify defects similar to existing static analysis tools [79]. The rules are listed in Table 13. String patterns needed to implement 'isKind' is shown in the 'String Pattern' column. For rule derivation, we identify commonalities amongst coding patterns that map to existing defects reported in Section 3.1. For example, the coding patterns

6. 'Shifu' (师傅) is a Chinese word, which means 'master'

TABLE 11: Detection accuracy of eight tools. '-' means a precision (P) or recall (R) of 0.0.

Category	Sub-category	#	Checkov		Datree		Kube-conform		KubeLinter		Kube-Score		Kubesec		SLIKUBE		Yamllint	
			P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
Container Provisioning	Resources	9	0.01	0.11	-	-	-	-	0.01	0.12	0.002	0.05	0.02	0.27	-	-	-	-
Data Fields	IDT	19	-	-	0.02	0.03	0.67	0.03	-	-	-	-	1.00	0.03	-	-	-	-
	Syntax	35	-	-	1.00	0.01	-	-	-	-	-	-	-	-	-	-	0.001	0.50
	VR	30	-	-	0.24	0.09	0.24	0.07	-	-	0.24	0.07	0.28	0.07	-	-	-	-
Entity Referencing	N/A	125	0.002	0.003	-	-	-	-	0.01	0.01	0.03	0.01	0.01	0.01	-	-	-	-
Namespaces	N/A	15	0.01	0.14	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Probing	N/A	22	0.06	0.20	0.03	0.06	-	-	-	-	0.06	0.16	-	-	-	-	-	-
Security	AC	76	0.02	0.02	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	ESD	4	0.002	0.17	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	SC	11	0.01	0.20	-	-	-	-	-	-	0.01	0.13	0.02	0.39	-	-	-	-
Version Incompatibility	N/A	58	-	-	-	-	-	-	0.04	0.02	0.08	0.01	-	-	-	-	-	-
Avg.		404	0.01	0.01	0.02	0.01	0.28	0.004	0.01	0.01	0.02	0.01	0.02	0.01	-	-	0.001	0.001

TABLE 12: Classification of unmapped rules and their counts across 8 tools.

Category	**Definition**	**#**	**Checkov**	**Datree**	**Kube-conform**	**Kube-Linter**	**Kube-Score**	**Kube-Sec**	**SLI-KUBE**	**Yaml-lint**
violations of Best Practice	This defect category occurs when coding patterns in configuration scripts violate coding-related best practices, such as not specifying image tags, or missing labels or annotations for resource organization.	948	439	15	-	106	160	171	57	-
Built-in Features	This defect category occurs when manifests deploy or fail to disable risky built-in components and legacy features, such as the Kubernetes Dashboard, Helm v2 Tiller, or insecure NGINX Ingress snippets.	2	2	-	-	-	-	-	-	-
Control Plane Configuration	This defect category occurs when Kubernetes control plane or kubelet settings are misconfigured through flags, admission plugins, or certificate parameters, such as anonymous-auth enabled, missing TLS certs, or disabled RBAC admission plugins.	0	0	-	-	-	-	-	-	-
Broken Isolation	This defect category occurs when Kubernetes resources are configured to break isolation boundaries by sharing host namespaces or binding sensitive host interfaces such as hostPID, hostIPC, or hostNetwork.	32	20	0	-	9	-	0	3	-
Missing Availability Safeguards	This defect category occurs when workloads are deployed without safeguards that improve availability. Safeguards include, for example, replica requirements, PodDisruptionBudgets, cronjob deadlines, or topology spread constraints.	50	-	0	-	0	45	-	5	-
Tool Setup	This defect category occurs when configuration scripts violate requirements imposed by higher-level Kubernetes management tools, such as GitOps platforms, CI/CD controllers, and operators. These tools extend Kubernetes with their own conventions for labels, namespaces, and resource configurations.	0	-	0	-	-	-	-	-	-
Total		1,032	461	15	-	115	205	171	65	-

`mountPath: /var/lib/kubelet` and `mountPath: /var/lib/kubelet/plugins/ebs.csi.aws.com` appear for two instances of incorrect Helming where a hard-coded value is used for a key called 'mountPath.' The commonality here is both coding patterns having a hard-coded value for a key that is used in a template. Thus, we can abstract these coding patterns into a rule 'isTemplate(x) $\wedge \exists$((x.key) $\wedge$ isHardCoded($x.key.value$)).' We repeat the same process for orphanism.

ConShifu is a static analysis tool that we execute using the command line for 2,260 scripts in 0.76 hours. Using ConShifu we identify 381 instances of defects. From these identified defects, we evaluate a random sample of 192 instances. On this sample, we obtain an average precision and recall of respectively, 0.83 and 0.92. The precision and recall of ConShifu for incorrect Helming is respectively, 0.85 and 0.96. The precision and recall of ConShifu for orphanism is respectively, 0.81 and 0.89. These results are obtained at a 95% confidence level, providing us the confidence that the detected instances of incorrect Helming and orphanism could be of relevance to practitioners.

Step#3 - Evaluation Using Practitioner Feedback: We submit issue reports to obtain feedback on the detected defects by ConShifu. We start with 185 repositories and exclude archived ones, resulting in 124 active repositories as of August 01, 2024. We apply ConShifu on these 124 repositories to detect defects and submit issue reports for developer feedback. ConShifu analyzes 8,576 scripts in 22 minutes and respectively, identifies 183 and 198 instances of incorrect Helming and orphanism. We take a random sample and submit 24 issue reports for 26 instances of incorrect Helming and 18 instances of orphanism. We take a random sample to comply with ethical recommendations by not spamming the practitioners [36]. Each issue report includes the defect's location, a brief description, the potential consequences, and a fix that is submitted as a GitHub pull request. The 24 submitted issue reports correspond to 21 distinct repositories. Table 14 lists the repositories for which we submit issue reports with URLs to the issue reports.

5.2.2 Results for RQ3.b

As of Jan 20 2025, we obtain 33 responses for 44 defects. Practitioners have confirmed 26 defects as valid. As shown in Figure 6, of the 26 valid defects, 21 are related to incorrect Helming and 5 are related to orphanism. Four defects of orphanism detected by ConShifu are rejected as they reside in an application where the configuration values are expected to be provided by users. Evidence of submitted defect reports are available online [112].

TABLE 13: Rules Used by ConShifu

Category	Rule	String Pattern
Incorrect Helming	isTemplate(x) ∧ ∃((x.key) ∧ isHardCoded($x.key.value$))	N/A
Orphanism	(isKind(x) ∧ ¬isReferenced($x.key.value$)) ∨ (isKind(x) ∧ ¬isReferenceExist($x.key.value$))	'ServiceAccount,' 'ClusterRole,' 'StorageClass,' 'PersistentVolumeClaim,' 'PersistentVolume,' 'Role'

***Answer to RQ3.b**: We submit 44 defect reports and receive 33 responses, of which 26 are confirmed valid by practitioners. Among these, 21 relate to incorrect Helming and 5 to orphanism. The agreement rate is 79% for the acknowledged defect reports.*

TABLE 14: Issue reports per repository.

Repository	Count	Issue(s)
kedacore/charts	1	[1]
aquasecurity/trivy-operator	1	[1]
clastix/kamaji	1	[1]
k8gb-io/k8gb	1	[1]
kubernetes-sigs/aws-ebs-csi-driver	1	[1]
zalando/postgres-operator	2	[1], [2]
carina-io/carina	2	[1], [2]
mspnp/microservices-reference-implementation	1	[1]
apache/openwhisk-deploy-kube	2	[1], [2]
apache/dubbo-admin	1	[1]
aws/amazon-vpc-cni-k8s	1	[1]
kube-logging/logging-operator	1	[1]
kube-green/kube-green	1	[1]
aws/eks-charts	1	[1]
senthilrch/kube-fledged	1	[1]
kubeshark/kubeshark	1	[1]
clusternet/clusternet	1	[1]
kadalu/kadalu	1	[1]
mongodb/mongodb-enterprise-kubernetes	1	[1]
kubernetes-sigs/prometheus-adapter	1	[1]
kserve/kserve	1	[1]

6 DISCUSSION

We discuss our findings as follows:

6.1 Significance of Our Empirical Study

The significance of our work can be summarized as follows:

- Our findings fundamentally advance the science of container orchestration by providing the first systematic investigation of defects in Kubernetes configuration scripts. Despite Kubernetes being the most popular tool to implement the practice of container orchestration [12], configuration defects has remained an under-explored area. Our taxonomy and empirical findings fill this gap, establishing a foundation for future research in configuration quality assurance and automated repair;
- The taxonomy of configuration defects contributes to the knowledge of software defect literature. Prior work [76] has demonstrated that defect taxonomies are valuable for validation and verification efforts. Our taxonomy reveals categories specific to container orchestration, such as orphanism, pod scheduling, and incorrect Helming, which have not been documented in previous defect studies. These categories enable targeted improvements in quality assurance for container orchestration. Understanding which defects occur frequently and their consequences helps prioritize detection and prevention efforts;
- The dataset presented in this paper can be used for tool evaluation. Our evaluation of eight static analysis tools reveals which defect categories are currently supported and which categories lack tool support. These insights can be used to further improve tools to detect defects for container orchestration;
- Automated program repair for container orchestration is an under-explored area with a lot of potential. Our dataset of fix patterns makes a foundational contribution to this area. The documented patterns show how practitioners resolve different types of configuration defects. This dataset can be used to evaluate existing program repair techniques for configuration defects and develop new techniques for configuration repair; and
- We have developed a new tool called ConShifu that identifies defects that have been confirmed by open source contributors. This shows that the identified defect categories and our tool has relevance for practitioners.

We further discuss the implications and limitations respectively, in Sections 6.2 and 6.3.

6.2 Implications of Our Findings

The implications of our findings are:

6.2.1 *Prioritizing Validation Efforts Based on Defect Frequency*

Our analysis of defect frequency highlights three implications:

1) practitioners should prioritize validation efforts for high-frequency categories, such as entity referencing and unsatisfied dependency, since these fields are foundational for Kubernetes manifests and account for the majority of observed defects;
2) tool builders should ensure that analyzers provide adequate coverage for categories, such as entity referencing and unsatisfied dependency; and
3) although categories, such as orphanism and property annotation are less frequent, they still require mitigation as they can lead to serious consequences, such as crashes and outages.

6.2.2 *'Shift Left' Approach Towards Defect Detection*

In software development, the 'shift left' approach advocates for pro-active integration of quality assurance activities,

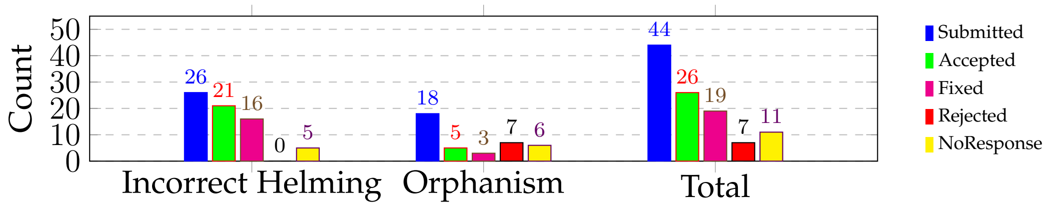

Fig. 6: Count of submitted, accepted, and fixed defects identified by ConShifu.

such as application of static analysis tools in the software development process [72]. We advocate for a 'shift left' approach for configuration management of Kubernetes as well. From our analysis, we observe 533 of the 719 defects result in a crash or an incorrect operation or outage. This finding shows defects in Kubernetes configuration scripts to be consequential, and therefore the community should take actions on how to facilitate defect detection for Kubernetes configuration scripts. Our findings and the dataset could be helpful in this regard as it could help the community understand the nature of defects. While static analysis tools suffer from low actionability due to false positives [80], these tools still provide value for practitioners [3], and therefore could be useful for detecting configuration defects.

Despite their limitations, practitioners can still benefit from using these tools, as they detect coding patterns that can aid in validation efforts. Moreover, prior work shows that practitioners still find static analysis tools valuable even when they produce false positives, since detecting critical defects is perceived to be better than missing them [3]. Our evaluation shows certain alerts correspond to violations of best practice, availability safeguards, and resource isolation issues. Therefore, despite low precision in our evaluation, these tools remain useful in practice for validating configuration scripts used in Kubernetes.

6.2.3 *The Need for Enhancing Static Analysis Tools for Kubernetes*

According to our analysis, none of the studied tools have support for 7 of the 15 categories. We also observe the second most frequently occurring defect category is unsatisfied dependency for which none of the eight studied tools provide any support. With a precision value of 0.28, Kubeconform has the highest average precision amongst all 8 tools. This is lower than what practitioners perceive 'acceptable,' i.e., a precision $<= 0.90$ [80]. Furthermore, while 5 of the 8 studied tools support the most frequently occurring category of entity referencing, the precision and recall is ≤ 0.03 for each tool.

The above-mentioned evidence highlights the need of enhancing static analysis tools for Kubernetes with respect to support and increasing detection accuracy. We provide three recommendations. **First**, detection rules used by existing tools need to be improved. Our curated dataset of defects can be used for improving the rules. **Second**, practitioner feedback can be collected to improve the detection accuracy of static analysis tools. These tools should allow for seamless integration into existing developer workspace in order to collect feedback for the detected defects. Prior research also advocated for obtaining practitioner feedback to improve detection accuracy of static analysis tools [79]. **Third**, runtime data from Kubernetes clusters can be collected to detect five categories of defects namely, conditional, CR, property annotation, unsatisfied dependency, and volume mounting. Detection for each of these categories is dependent on information that can be collected at runtime. An example utility is 'kubectl cluster-info' that can provide cluster information at runtime [59].

6.2.4 *The Need for Automated Configuration Defect Repair Tools for Kubernetes*

Our findings highlight the need of developing automated defect repair tools for Kubernetes configuration scripts. The top four most frequently occurring fix patterns are configuration value changes, directive fix, property modification, and rule fix that are applied manually to fix 553 out of 719 defects. In order to develop defect repair techniques, researchers can use the curated list of defects and their corresponding fixes that are available as part of our dataset. Each defect in our dataset is associated with a fix pattern and linked to its GitHub issue and pull request, enabling researchers to trace how a reported defect was resolved in practice. As such, the dataset can support the development and evaluation of automated defect repair tools, similar to prior work that has leveraged curated defect-fix datasets for program repair research [11], [20]. Chen et al. [20] trained a sequence-to-sequence model using a large corpus of real bug–fix pairs to automatically generate patches for Java programs. Bader et al. [11] mined over 1,200 historical bug–fix commits to learn recurring fix patterns, enabling it to suggest human-like repairs with high accuracy. Researchers can investigate if the above-mentioned methods can be applied by using our identified fix patterns. We posit prior automated defect repair techniques to under-perform for 8 of the 15 defect categories that have not been reported in prior software systems.

6.2.5 *Prioritizing Validation Efforts Based on Consequences*

Our study shows that configuration defects can lead to serious consequences, such as crashes, outages, and exposure of unauthorized data. In total, 348 out of 719 defects in our dataset map to these severe consequences, underscoring the serious impact of configuration defects on Kubernetes-based deployment. Mapping defect categories to their consequences provides actionable insights: practitioners can prioritize validation efforts based on the severity of potential consequences. For example, the entity referencing category frequently maps to crashes or outages, with 60 out of 125 defects leading to crashes or outages, and could be considered high priority for validation efforts.

6.2.6 *Opportunities for Automating Configuration Inexecutability Detection*

From Section 4.2.1, we observe 52 defects to be related to configuration inexecutability. We find these defects to not exhibit any explicit symptoms, such as crashes or outages, which makes the defect detection process challenging. According to our analysis, practitioners take a reactive approach where they use the 'kubectl' command manually to identify these defects. This approach is time consuming, which necessitates development of automated techniques. One possible future direction can be usage of existing log-based defect localization techniques [27]. Another possible future direction could be application of reachability analysis [110] to detect defects related to configuration inexecutability.

6.3 Threats to Validity

We discuss the limitations of our paper as follows:

Conclusion Validity: The qualitative analysis process is subject to rater bias as the first and second authors derived categories for defects. In the case of disagreements, the last author was the resolver. We acknowledge that the inclusion of the resolver might have added bias in the qualitative analysis process. Answers to RQ3 is limiting, as we use eight tools and may have missed tools not included our paper. We mitigate this limitation by using a systematic selection criteria. Additionally, evaluation results for studied tools is dependent on the dataset created in Section 3.1, which may bias the results. Our definition of false positives, which only considers defects from our curated dataset is limiting as it may underestimate tool capabilities.

Construct Validity: Our study is susceptible to construct validity as the defect identification process depends on the accuracy and completeness of the parsed scripts. ConShifu is susceptible to miss defects as it uses a rule-based approach to identify defects. Furthermore, ConShifu can generate false positives while reporting instances of incorrect Helming and orphanism. ConShifu may fail to detect instances of incorrect Helm if there are no Helm scripts, `values.yaml` files, or templates.

External Validity: Our findings are obtained from OSS repositories, which may not generalize for configuration scripts used in proprietary repositories. We mitigate this limitation by analyzing repositories from GitHub, which is the most popular code sharing platform.

7 Related Work

Our paper is related with existing research on defect categorization and quality assurance aspects of Kubernetes, which we describe in the following subsections:

7.1 Prior Research Related with Defect Categorization

Software defect categorization has been of interest to researchers since the 1990s. In 1992, Chillarege and colleagues [21] proposed the orthogonal defect classification (ODC) taxonomy, which consists of eight defect categories. Since then, researchers have used and extended the ODC taxonomy. For instance, Alannsary and Tian [2] and Silva et al. [90] used ODC to respectively, categorize defects for software-as-a-service and embedded software systems. ODC was also extended by Hunny et al. [46] to classify security vulnerabilities.

Researchers have also developed their own taxonomies because of ODC's limitations for modern software systems [89]. Researchers, such as Yu et al. [107], Wan et al. [96], Cui et al. [26], and Du et al. [32] in separate publications derived defect taxonomies respectively, for container runtime systems, blockchain projects, database systems, and federated learning systems. Makhshari and Mesbah [68], Chen et al. [19], Shen et al. [88], Gao et al. [35], Wang et. at [98], Wang et. al [97] constructed defect taxonomies respectively, for IoT software projects, deep learning-based deployment, deep learning compilers, distributed systems, android applications, and autopilot software systems. Wang et al. [99] analyzed 83 defects in WeChat Mini-Programs, and categorized them into 6 categories. Cotroneo et al. [25] categorized the failures of OpenStack using a bottom-up approach. Hassan et al. [42] conducted an empirical study involving 5,110 state reconciliation defects and classify these defects into 8 categories. Rahman et al. [76] developed a taxonomy of defects in IaC scripts by applying descriptive coding with 1,448 defect-related commits. Humbatova et al. [45] analyzed GitHub issues and Stack Overflow posts to develop a classification of faults for software projects involving deep learning. Wang et al. [100] studied configuration defects that occur when these configurations are provided at runtime for database and web server systems.

7.2 Prior Research Related with Quality Aspects of Kubernetes

Researchers have shown increasing interest in quality assurance for Kubernetes in recent years. Yang et al. [105] focused on vulnerabilities in the orchestration layer, and recommended two practices for enhancing the security of Kubernetes clusters. Kamieniarz et al. [54] studied the security vulnerabilities that can occur in Kubernetes-related deployments. Rahman et al. [77] in particular identified what types of Kubernetes objects are impacted by security weaknesses, such as hard-coded passwords and insecure HTTP. They [77] also quantified correlations between development activity metrics and the presence of security weaknesses. Carmen et al. [16] in their study, created a new taxonomy for Kubernetes scheduling techniques, organizing the techniques into five main domains and highlighting where current scheduling techniques fall short, especially in terms of security and performance. Gu et al. [39], Sun et al. [93], [94], and Xu et al. [103] in separate publications focused on analyzing and detecting defects related to Kubernetes controllers. Xu et al. [103] focused on deriving a taxonomy for defects that occur in Kubernetes operators, which are specialized controllers. Gu et al. [39] and Sun et al. [93], [94] focused on deriving testing techniques that can expose defects in Kubernetes controllers and operators. Barletta et al. [12] analyzed and classified failures in the Kubernetes orchestration layer and developed a fault-injection framework that targets the cluster's etcd datastore.

Prior research has focused on controller and operator related defects [39], [93], [94]. A controller is a core Kubernetes component that continuously monitors the cluster state and reconciles it to match the desired configuration, such as ensuring the correct number of pods, while an operator is a custom controller that automates complex application-specific tasks by managing custom resources. Barletta et al. [12] studied Kubernetes failure at the orchestrator level, focusing on how etcd datastore corruption affects the core orchestration platform rather than defects in individual controllers or operators. The above-mentioned publications focused on failures caused by defects in these core or extension mechanisms, as well as runtime and infrastructure-level issues. In contrast, our work targets configuration defects in the Kubernetes YAML scripts written by practitioners to define Kubernetes resources. Our taxonomy provides a categorization of configuration defects, which highlights configuration scripts as a distinct focus relative to prior work on operators and controllers. The closest in spirit to our work is research conducted by Rahman et al. [77], who only focused on security-related defects. By investigating configuration defects in general, our study identifies categories that prior research has not addressed. The defect categories that we have identified but are not reported in prior Kubernetes-related papers [12], [15], [39], [75], [77], [84]–[87], [93], [94], [113] are: custom resource, data field, entity referencing, incorrect Helming, namespaces, orphanism, pod scheduling, property annotation, unsatisfied dependency, and volume mounting. Also, prior work have not investigated fix patterns, consequences, and evaluation of existing static analysis tools. While working with Kubernetes, practitioners need to use configuration options related to (i) pods and (ii) state reconciliation. To configure pods, i.e., abstractions to group containers, practitioners need to understand non-trivial concepts such as affinity and annotations. Likewise, to configure state reconciliation, developers need to understand concepts such as custom resources. Erroneous usage of the these configuration options can result in defective Kubernetes deployments.

In short, with respect to advancing science, our paper addresses several gaps in Kubernetes-related prior research by:

1) identifying Kubernetes-specific entities, namely namespaces, pods, and properties to be defect-related when using configuration scripts, which prior work has not identified;
2) characterizing how usage of entity referencing can lead to defects in configuration scripts for Kubernetes, which prior work has not examined;
3) providing a mapping between configuration defects and their consequences as well as fix patterns, which prior work has not documented;
4) quantifying the support of existing static analysis tools for detecting identified defect categories, which prior work has not evaluated; and
5) identifying 7 categories of configuration defects, namely custom resource, incorrect Helming, namespaces, orphanism, pod scheduling, property annotation, and volume mounting, that have not been reported in any prior work.

8 Conclusion

Kubernetes is becoming popular in industry as a tool for automated management of containers. Configuration defects in Kubernetes can be consequential and, unfortunately, are not uncommon. This paper reports an empirical study about Kubernetes-related configuration defects alongside their consequences and fix patterns. The goals of this empirical study are (i) to help practitioners who use Kubernetes to detect configuration defects, and (ii) to offer researchers opportunities for improving existing static analysis tools for detecting those defects. Our study includes 719 defects mined from 185 OSS repositories. We identify 15 defect categories for Kubernetes configuration scripts. We find that insights obtained from existing defects can be used to identify previously-unknown defects. For example, using our linter ConShifu, we identify 26 defects that have been accepted as valid defects by the practitioners of the corresponding OSS projects.

Our research study has produced multiple lessons. For example, we provide recommendations for researchers on how existing defects that are available as part of our dataset, can be leveraged to enhance existing static analysis tools and to develop defect repair techniques for Kubernetes. We also advocate for incorporating practitioner feedback and runtime information to improve existing static analysis tools for Kubernetes configuration scripts.

Data Availability

The datasets and source code used for the paper are publicly-available online as a replication package [112]. URL of the package: https://figshare.com/s/5c63f862a1abd95f7708.

Acknowledgments

We thank the PASER group at Auburn University for their valuable feedback. This research was partially funded by the U.S. National Science Foundation (NSF) Award # 2310179, # 2312321, # CNS-2026928, # CCF-2349961, and # CCF-2319472.

References

[1] Adrienverge, “yamllint,” Online, 2024, accessed: 2024-06-23. [Online]. Available: https://github.com/adrienverge/yamllint

[2] M. Alannsary and J. Tian, “Cloud-odc: Defect classification and analysis for the cloud,” pp. 71–77, 2015, copyright - Copyright The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp) 2015; Document feature - Diagrams; Tables; Graphs; ; Last updated - 2015-08-21.

[3] A. Ami, K. Moran, D. Poshyvanyk, and A. Nadkarni, “‘false negative - that one is going to kill you’ - understanding industry perspectives of static analysis based security testing,” in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 23–23. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00019

[4] aquasecurity, "helm - trivy-server should use trivy.repository:trivy.tag for image if defined," 2022, accessed: 2024-06-23. [Online]. Available: https://github.com/aquasecurity/trivy-operator/issues/729

[5] A. B. Aral, "Dear linux, privileged ports must die," August 2022, accessed: 2024-07-17. [Online]. Available: https://ar.al/2022/08/30/dear-linux-privileged-ports-must-die/

[6] Argoproj, "application controller needs a liveness probe," 2019, accessed: 2024-06-23. [Online]. Available: https://github.com/argoproj/argo-cd/issues/1782

[7] ——, "core-install manifest references undefined argocd-server serviceaccount," 2021, accessed: 2024-06-23. [Online]. Available: https://github.com/argoproj/argo-cd/issues/7760

[8] argoproj labs, "operator resources should have unique labels." 2022, accessed: 2024-06-23. [Online]. Available: https://github.com/argoproj-labs/argocd-operator/issues/750

[9] H. Arksey and L. O'Malley, "Scoping studies: towards a methodological framework," *International Journal of Social Research Methodology*, vol. 8, no. 1, pp. 19–32, 2005. [Online]. Available: https://doi.org/10.1080/1364557032000119616

[10] Aws, "vsphere username escaped resulting in failed authentication on creating workload cluster," 2022, accessed: 2024-06-23. [Online]. Available: https://github.com/aws/eks-anywhere/issues/1639

[11] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[12] M. Barletta, M. Cinque, C. Di Martino, Z. T. Kalbarczyk, and R. K. Iyer, "Mutiny! how does kubernetes fail, and what can we do about it?" in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2024, pp. 1–14.

[13] A. Baur, "Packaging of kubernetes applications," in *Proceedings of the 2020 OMI Seminars (PROMIS 2020)*, vol. 1. Universität Ulm, 2021, pp. 1–1.

[14] blackduck, "Policy-as-code," 2023, accessed: 2024-06-23. [Online]. Available: https://www.blackduck.com/glossary/what-is-policy-as-code.html

[15] D. B. Bose, A. Rahman, and S. I. Shamim, "'under-reported'security defects in kubernetes manifests," in *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*. IEEE, 2021, pp. 9–12.

[16] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Comput. Surv.*, vol. 55, no. 7, dec 2022. [Online]. Available: https://doi.org/10.1145/3539606

[17] H. Chart, "The chart best practices guide," 2024, accessed: 2024-06-23. [Online]. Available: https://v2-14-0.helm.sh/docs/chart_best_practices/

[18] Checkov, "Checkov," Online, 2024, accessed: 2024-06-07. [Online]. Available: https://www.checkov.io/

[19] Z. Chen, H. Yao, Y. Lou, Y. Cao, Y. Liu, H. Wang, and X. Liu, "An empirical study on deployment faults of deep learning based mobile applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 674–685.

[20] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[21] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943–956, Nov 1992.

[22] CNCF, "Container Orchestration," https://glossary.cncf.io/container-orchestration/, 2024, [Online; accessed 24-August-2024].

[23] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960. [Online]. Available: http://dx.doi.org/10.1177/001316446002000104

[24] Controlplane, "Kubesec," Online, 2024, accessed: 2024-06-23. [Online]. Available: https://kubesec.io/

[25] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, "How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 200–211. [Online]. Available: https://doi.org/10.1145/3338906.3338916

[26] Z. Cui, W. Dou, Y. Gao, D. Wang, J. Song, Y. Zheng, T. Wang, R. Yang, K. Xu, Y. Hu, J. Wei, and T. Huang, "Understanding transaction bugs in database systems," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639207

[27] H. Dai, H. Li, C.-S. Chen, W. Shang, and T.-H. Chen, "Logram: Efficient log parsing using nn-gram dictionaries," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 879–892, 2022.

[28] Datree, "Datree," Online, 2024, accessed: 2024-06-23. [Online]. Available: https://www.datree.io/

[29] Deckhouse, "During installation main queue stucks with upmeter module," 2023, accessed: 2024-06-23. [Online]. Available: https://github.com/deckhouse/deckhouse/issues/3704

[30] R. Donato, "What is schema validation?" 2023, accessed: 2024-06-23. [Online]. Available: https://www.packetcoders.io/what-is-schema-validation/

[31] D. D'Silva and D. D. Ambawade, "Building a zero trust architecture using kubernetes," in *2021 6th International Conference for Convergence in Technology (I2CT)*, 2021, pp. 1–8.

[32] X. Du, X. Chen, J. Cao, M. Wen, S.-C. Cheung, and H. Jin, "Understanding the bug characteristics and fix strategies of federated learning systems," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1358–1370. [Online]. Available: https://doi.org/10.1145/3611643.3616347

[33] elastic, "[metricbeat] dns lookup failure for node host," 2019, accessed: 2024-06-23. [Online]. Available: https://github.com/elastic/helm-charts/issues/394

[34] G. Elbaz, "Static code analysis: Top 7 methods, pros/cons and best practices," 2023, accessed: 2024-06-23. [Online]. Available: https://www.oligo.security/academy/static-code-analysis

[35] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu, "An empirical study on crash recovery bugs in large-scale distributed systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 539–550. [Online]. Available: https://doi.org/10.1145/3236024.3236030

[36] N. E. Gold and J. Krinke, "Ethical mining: A case study on msr mining challenges," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 265–276. [Online]. Available: https://doi.org/10.1145/3379597.3387462

[37] G. Gousios and D. Spinellis, "Ghtorrent: Github's data from a firehose," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 12–21.

[38] grafana, "Grafana operator 5.4.1 resource limit too low, grafana-operator-controller-manager pod won't start[bug]," 2023, accessed: 2024-06-23. [Online]. Available: https://github.com/grafana/grafana-operator/issues/1255

[39] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu, "Acto: Automatic end-to-end testing for operation correctness of cloud system management," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 96–112. [Online]. Available: https://doi.org/10.1145/3600006.3613161

[40] H. Guan, Y. Xiao, J. Li, Y. Liu, and G. Bai, "A comprehensive study of real-world bugs in machine learning model optimization," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 147–158.

[41] hashicorp, "Consul server statefulset volume name isn't truncated," 2021, accessed: 2024-06-23. [Online]. Available: https://github.com/hashicorp/consul-k8s/issues/798

[42] M. M. Hassan, J. Salvador, S. K. K. Santu, and A. Rahman, "State reconciliation defects in infrastructure as code," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1865–1888, 2024.

[43] W. He, P. Di, M. Ming, C. Zhang, T. Su, S. Li, and Y. Sui, "Finding and understanding defects in static analyzers by constructing automated oracles," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, jul 2024. [Online]. Available: https://doi.org/10.1145/3660781

[44] K. Huang, B. Chen, S. Wu, J. Cao, L. Ma, and X. Peng, "Demystifying dependency bugs in deep learning stack," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 450–462. [Online]. Available: https://doi.org/10.1145/3611643.3616325

[45] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1110–1121. [Online]. Available: https://doi.org/10.1145/3377811.3380395

[46] U. Hunny, M. Zulkernine, and K. Weldemariam, "Osdc: Adapting odc for developing more secure software," 03 2013, pp. 1131–1136.

[47] IEEE, "IEEE standard classification for software anomalies," *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pp. 1–23, Jan 2010.

[48] istio, "Add accessmodes option to helm grafana chart," 2018, accessed: 2024-06-23. [Online]. Available: https://github.com/istio/istio/commit/64a46ebdd9eec1e805a9800e09e687c0968a4462

[49] jaegertracing, "Incorrect image tag in the published yaml," 2021, accessed: 2024-06-23. [Online]. Available: https://github.com/jaegertracing/jaeger-operator/issues/1666

[50] Jake Page, "Kubernetes fail compilation: but they keep getting worse," https://medium.com/@jake.page91/kubernetes-fail-compilation-but-they-keep-getting-worse-c6f4fb3e6b38, 2024, [Online; accessed 29-July-2024].

[51] Jayme Howard, "You Broke Reddit: The Pi-Day Outage," https://www.reddit.com/r/RedditEng/comments/11xx5o0/you_broke_reddit_the_piday_outage/, 2024, [Online; accessed 30-July-2024].

[52] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "The symptoms, causes, and repairs of bugs inside a deep learning library," *Journal of Systems and Software*, vol. 177, p. 110935, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221000327

[53] K. Kamieniarz and W. Mazurczyk, "A comparative study on the security of kubernetes deployments," in *2024 International Wireless Communications and Mobile Computing (IWCMC)*. IEEE, 2024, pp. 0718–0723.

[54] ——, "A comparative study on the security of kubernetes deployments," in *2024 International Wireless Communications and Mobile Computing (IWCMC)*, May 2024, pp. 0718–0723.

[55] Kedacore, "fix: adj indent of extravolumes volumemounts in 14-keda-deployment," 2023, accessed: 2024-06-23. [Online]. Available: https://github.com/kedacore/charts/pull/419

[56] kserve, "Kserve installation fails with kubeflow due to wrong cert injection namespace for servingruntime webhook," 2023, accessed: 2024-06-23. [Online]. Available: https://github.com/kserve/kserve/issues/3187

[57] kubernetes, "Kep-2067: Rename the kubeadm "master" label and taint," 2022. [Online]. Available: https://github.com/kubernetes/enhancements/blob/master/keps/sig-cluster-lifecycle/kubeadm/2067-rename-master-label-taint/README.md

[58] Kubernetes User Case Studies, July 2024. [Online]. Available: https://kubernetes.io/case-studies/

[59] Kubernetes, "Kubernetes Documentation," https://kubernetes.io/docs/home/, 2024, [Online; accessed 14-August-2024].

[60] kubernetes sigs, "Failed to create listener: bind: permission denied," 2021, accessed: 2024-06-23. [Online]. Available: https://github.com/kubernetes-sigs/metrics-server/issues/782

[61] ——, "Error on ebs-csi-controller pod," 2022, accessed: 2024-06-23. [Online]. Available: https://github.com/kubernetes-sigs/aws-ebs-csi-driver/issues/1357

[62] kyverno, "[bug] clusterrole kyverno:events tighten scope on apigroups," 2022, accessed: 2024-06-23. [Online]. Available: https://github.com/kyverno/kyverno/issues/3222

[63] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977. [Online]. Available: http://www.jstor.org/stable/2529310

[64] K. Li, Y. Xue, S. Chen, H. Liu, K. Sun, M. Hu, H. Wang, Y. Liu, and Y. Chen, "Static application security testing (sast) tools for smart contracts: How far are we?" *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660772

[65] linode, "The gatekeeper post install job cannot run pod due to psp and insufficient securitycontext," 2021, accessed: 2024-06-23. [Online]. Available: https://github.com/linode/apl-core/issues/688

[66] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *SIGPLAN Not.*, vol. 51, no. 1, p. 298–312, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2914770.2837617

[67] longhorn, "[bug] backing image resync not work on v1.2.x," 2022, accessed: 2024-06-23. [Online]. Available: https://github.com/longhorn/longhorn/issues/4738

[68] A. Makhshari and A. Mesbah, "Iot bugs and development challenges," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 460–472.

[69] P. Mendis, W. Reeves, M. A. Babar, Y. Zhang, and A. Rahman, "Evaluating the quality of open source ansible playbooks: An executability perspective," in *Proc. of SEA4DQ*, ser. SEA4DQ 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 2–5. [Online]. Available: https://doi.org/10.1145/3663530.3665019

[70] Mirantis, "What are the primary reasons your organization is using Kubernetes?" 2021. [Online]. Available: https://www.mirantis.com/cloud-case-studies/paypal/

[71] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Software Engineering*, pp. 1–35, 2017. [Online]. Available: http://dx.doi.org/10.1007/s10664-017-9512-6

[72] Q.-S. Phan, K.-H. Nguyen, and T. Nguyen, "The challenges of shift left static analysis," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2023, pp. 340–342.

[73] Prometheus-community, "[kube-prometheus-stack] complex config templating in alertmanager results into helm warning," 2023, accessed: 2024-06-23. [Online]. Available: https://github.com/prometheus-community/helm-charts/issues/2950

[74] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[75] A. Rahman, G. Dozier, and Y. Zhang, "Authorship of minor contributors in kubernetes configuration scripts: An exploratory study," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 1424–1427.

[76] A. Rahman, E. Farhana, C. Parnin, and L. Williams, "Gang of eight: A defect taxonomy for infrastructure as code scripts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 752–764. [Online]. Available: https://doi.org/10.1145/3377811.3380409

[77] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, "Security misconfigurations in open source kubernetes manifests: An empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, May 2023. [Online]. Available: https://doi.org/10.1145/3579639

[78] Ravi Patel, "Introduction to Container Orchestration," https://medium.com/@ravipatel.it/introduction-to-container-orchestration-e219e36007ab, 2024, [Online; accessed 22-August-2024].

[79] S. Reis, R. Abreu, M. d'Amorim, and D. Fortunato, "Leveraging practitioners' feedback to improve a security linter," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023.

[80] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, p. 58–66, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3188720

[81] J. Saldaña, *The coding manual for qualitative researchers*. Sage, 2015.

[82] C. B. Seaman, F. Shull, M. Regardie, D. Elbert, R. L. Feldmann, Y. Guo, and S. Godfrey, "Defect categorization: Making use of a decade of widely varying historical data," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 149–157. [Online]. Available: https://doi.org/10.1145/1414004.1414030

[83] SEC, "U.S. SEC," https://www.sec.gov/Archives/edgar/data/1713445/000162828024006294/reddits-1q423.htm, 2024, [Online; accessed 22-Feb-2024].

[84] M. S. I. Shamim, F. A. Bhuiyan, and A. Rahman, "Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," *2020 IEEE Secure Development (SecDev)*, pp. 58–64, 2020.

[85] S. I. Shamim, H. Hu, and A. Rahman, "Dynamic application security testing for kubernetes deployment: An experience report from industry," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 514–519.

[86] ——, "On prescription or off prescription? an empirical study of community-prescribed security configurations for kubernetes," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 707–707.

[87] S. I. Shamim, F. Wu, H. Shahriar, A. Skjellum, and A. Rahman, " Authentic Learning Exercise for Kubernetes Misconfigurations: An Experience Report of Student Perceptions ," in *2025 IEEE/ACM 37th International Conference on Software Engineering Education and Training (CSEET)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 292–302. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CSEET66350.2025.00037

[88] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 968–980. [Online]. Available: https://doi.org/10.1145/3468264.3468591

[89] N. Silva and M. Vieira, "Experience report: Orthogonal classification of safety critical issues," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 156–166.

[90] ——, "Software for embedded systems: a quality assessment based on improved odc taxonomy," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1780–1783. [Online]. Available: https://doi.org/10.1145/2851613.2851908

[91] Snyk, "What is container orchestration?" https://snyk.io/learn/container-security/container-orchestration/, 2024, [Online; accessed 23-August-2024].

[92] StackRox, "Kubelinter documentation," Online, 2024, accessed: 2024-06-23. [Online]. Available: https://docs.kubelinter.io/#/

[93] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, "Automatic reliability testing for cluster management controllers," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 143–159.

[94] X. Sun, W. Ma, J. T. Gu, Z. Ma, T. Chajed, J. Howell, A. Lattuada, O. Padon, L. Suresh, A. Szekeres *et al.*, "Anvil: Verifying liveness of cluster management controllers," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 649–666.

[95] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Softw. Engg.*, vol. 19, no. 6, p. 1665–1705, dec 2014. [Online]. Available: https://doi.org/10.1007/s10664-013-9258-8

[96] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug characteristics in blockchain systems: A large-scale empirical study," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 413–424.

[97] D. Wang, S. Li, G. Xiao, Y. Liu, and Y. Sui, "An exploratory study of autopilot software bugs in unmanned aerial vehicles," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 20–31. [Online]. Available: https://doi.org/10.1145/3468264.3468559

[98] J. Wang, Y. Jiang, T. Su, S. Li, C. Xu, J. Lu, and Z. Su, "Detecting non-crashing functional bugs in android apps via deep-state differential analysis," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 434–446. [Online]. Available: https://doi.org/10.1145/3540250.3549170

[99] T. Wang, Q. Xu, X. Chang, W. Dou, J. Zhu, J. Xie, Y. Deng, J. Yang, J. Yang, J. Wei, and T. Huang, "Characterizing and detecting bugs in wechat mini-programs," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 363–375. [Online]. Available: https://doi.org/10.1145/3510003.3510114

[100] T. Wang, Z. Jia, S. Li, S. Zheng, Y. Yu, E. Xu, S. Peng, and X. Liao, "Understanding and detecting on-the-fly configuration bugs," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 628–639.

[101] S. Wickramasinghe, "Static code analysis: The complete guide to getting started with sca," 2023, accessed: 2024-06-23. [Online]. Available: https://www.splunk.com/en_us/blog/learn/static-code-analysis.html

[102] B. Xu, S. Wu, J. Xiao, H. Jin, Y. Zhang, G. Shi, T. Lin, J. Rao, L. Yi, and J. Jiang, "Sledge: Towards efficient live migration of docker containers," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 2020, pp. 321–328.

[103] Q. Xu, Y. Gao, and J. Wei, "An empirical study on kubernetes operator bugs," in *Proceedings of the 33nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024.

[104] H. Yang, Y. Nong, T. Zhang, X. Luo, and H. Cai, "Learning to detect and localize multilingual bugs," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, jul 2024. [Online]. Available: https://doi.org/10.1145/3660804

[105] Y. Yang, W. Shen, B. Ruan, W. Liu, and K. Ren, "Security challenges in the container cloud," 12 2021, pp. 137–145.

[106] Yannh, "kubeconform," Online, 2024, accessed: 2024-06-23. [Online]. Available: https://github.com/yannh/kubeconform

[107] J. Yu, X. X. Xie, C. Zhang, S. Chen, Y. Li, and W. Shen, "Bugs in pods: Understanding bugs in container runtime systems," in *Proceedings of the 33nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024.

[108] Zalando, "Issues with postgres-ui-operator v1.10.0," 2023, accessed: 2024-06-23. [Online]. Available: https://github.com/zalando/postgres-operator/issues/2302

[109] Zegl, "Kube-score," Online, 2024, accessed: 2024-06-23. [Online]. Available: https://github.com/zegl/kube-score

[110] J. Zhang, R. Piskac, E. Zhai, and T. Xu, "Static detection of silent misconfigurations with deep interaction analysis," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021.

[111] U. Zhang, Yue Paul, , and A. Rahman, "Con-shifu docker image," https://hub.docker.com/r/zyue110026/conshifu-tool, 2024, [Online; accessed 12-Jan-2025].

[112] ——, "Replication package for paper," https://doi.org/10.6084/m9.figshare.26511229.v1, 2024, [Online; accessed 12-Jan-2025].

[113] Y. Zhang, R. Meredith, W. Reeves, J. Coriolano, M. A. Babar, and A. Rahman, "Does generative ai generate smells related to container orchestration?: An exploratory study with kubernetes manifests," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 192–196.

[114] Y. Zhang, J. Murphy, and A. Rahman, "Come for syntax, stay for speed, write secure code: an empirical study of security weaknesses in julia programs," *Empirical Software Engineering*, vol. 30, no. 2, p. 58, 2025.

[115] W. Zheng, C. Feng, T. Yu, X. Yang, and X. Wu, "Towards understanding bugs in an open source cloud management stack: An empirical study of openstack software bugs," *J. Syst. Softw.*, vol. 151, no. C, p. 210–223, May 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2019.02.025

Yue Zhang Yue Zhang is a PhD student at Auburn University. Her research interests is in software engineering and data science. She received the B.E. in Computer Science and Technology from the Anhui Jianzhu University, Hefei, China, in 2021.

Uchswas Paul Uchswas Paul is a Ph.D. student in Computer Science at North Carolina State University, USA. He earned his bachelor's degree from Khulna University of Engineering and Technology in 2018. Before joining his doctorate, he gained experience in industry and academia. His research interests lie in software engineering and large language models.

Marcelo d'Amorim Marcelo d'Amorim is an Associate Professor in Computer Science at the North Carolina State University, USA. He obtained his PhD from the University of Illinois at Urbana-Champaign in 2007 and his MS and BS degrees from UFPE, Brazil, in 2001 and 1997, respectively. Marcelo's research goal is to help developers build correct software. He is interested in preventing, finding, diagnosing, and repairing software bugs and vulnerabilities.

Akond Rahman Akond Rahman is an assistant professor at Auburn University. His research interests include DevOps and secure software development. He graduated with a PhD from North Carolina State University, an M.Sc. in Computer Science and Engineering from University of Connecticut, and a B.Sc. in Computer Science and Engineering from Bangladesh University of Engineering and Technology. He won the ACM SIGSOFT Doctoral Symposium Award at ICSE in 2018, the ACM SIGSOFT Distinguished Paper Award at ICSE in 2019, the CSC Distinguished Dissertation Award, and the COE Distinguished Dissertation Award from NC State in 2020. He actively collaborates with industry practitioners from GitHub, WindRiver, and others. To know more about his work visit https://akondrahman.github.io/