# An Empirical Study on Kubernetes Operator Bugs

Qingxin Xu
Key Laboratory of System Software, Institute of Software Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
xuqingxin19@otcaix.iscas.ac.cn

Yu Gao*
Key Laboratory of System Software, Institute of Software Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
gaoyu15@otcaix.iscas.ac.cn

Jun Wei*
Key Laboratory of System Software, Institute of Software Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
wj@otcaix.iscas.ac.cn

## Abstract

Kubernetes is the leading cluster management platform, and within Kubernetes, an operator is an application-specific program that leverages the Kubernetes API to automate operation tasks for managing an application deployed on a Kubernetes cluster. Users can declare a desired state for the managed cluster, specifying their configuration preferences. The operator program is responsible for reconciling the cluster's actual state to align with the desired state. However, the complex, dynamic, and distributed nature of the overall system can introduce operator bugs, and lead to severe consequences, e.g., outages and undesired cluster state.

In this paper, we conduct the first comprehensive study on 210 operator bugs from 36 Kubernetes operators. For all the studied bugs, we investigate their root causes, manifestations, impacts and fixing. Our study reveals many interesting findings that can guide the detection and testing of operator bugs, as well as the development of more reliable operators.

## CCS Concepts

• **General and reference → Empirical studies**; • **Computer systems organization → Distributed architectures**; **Reliability**.

## Keywords

Kubernetes, operation, operator bugs, empirical study

**ACM Reference Format:**
Qingxin Xu, Yu Gao, and Jun Wei. 2024. An Empirical Study on Kubernetes Operator Bugs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680396

*Yu Gao and Jun Wei are the corresponding authors.

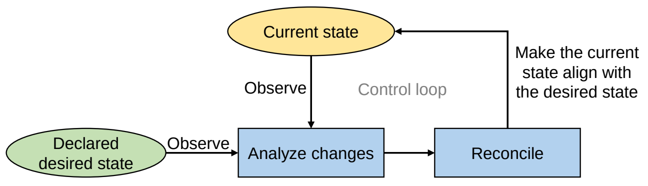


**Figure 1: Control loop.**

## 1 Introduction

In cloud computing era, more and more cloud applications have been developed to run on top of various cluster management platforms [19, 21, 24, 25, 60, 64, 66]. These cluster management platforms expose a restricted set of commands and operations to users through APIs, with the goal of simplifying the management and operation of application clusters. Presently, Kubernetes has emerged as the most widely adopted cluster management platform [22, 23].

To automate complex application-specific operation tasks beyond what Kubernetes itself provides, e.g., software upgrades, configuration updates and autoscaling, operation programs called *operators* [15, 17] are introduced. Operators follow the Kubernetes control loop principle, and extend the Kubernetes API to automate and customize the management of applications running on Kubernetes. As shown in Figure 1, users can specify their configuration preferences by declaring the desired state of an application cluster. Operators continuously monitor the state of the managed cluster, analyze state changes and reconcile the cluster's current state to align with the desired state. The Kubernetes operator pattern has gained significant popularity and adoption in the Kubernetes ecosystem. Many cloud systems, e.g., Prometheus [9], etcd [8], MySQL [12, 18], Elasticsearch [11] and Kafka [10] today are managed by Kubernetes operators.

The correctness of operators is critical to cloud system reliability. Unfortunately, operator bugs pose a significant threat to the reliability of cloud systems. Operators run within complex, dynamic and distributed environments. To achieve the desired state, a single operator can contain sophisticated workflow logic that coordinates multiple steps and collaborates with multiple built-in Kubernetes controllers, leading to intricate interactions and potential conflicts. Moreover, operators face frequent operation changes with varying goals from users, as well as unpredictable environmental failures such as crashes and network disruptions. This combination makes Kubernetes operators prone to operator bugs. These operator bugs can lead to severe consequences such as application outages, data loss, and security issues [13, 14, 16, 33, 46–50].

Some approaches have been proposed to test and verify the correctness of Kubernetes operators [41, 62]. Sieve [62] tests the reliability of operators by injecting faults (e.g., node crashes, network delays) and perturbing the operators' views of the current system state. Acto [41] tests the functional correctness of operators by automatically generating a sequence of desired state declarations. Kivi [54] verifies the correctness of Kubernetes controllers with respect to a set of properties by model checking the interactions among controllers and events at the model level. Anvil [63] verifies whether the controller implementation in Rust satisfies eventually stable reconciliation for all executions. However, it remains unclear whether there are patterns among operator bugs and what bugs existing approaches may fail to detect. We also lack an operator bug dataset for future research. We believe that an in-depth study of operator bugs and an operator bug dataset can greatly promote the reliability research in operators on cluster management platforms.

In this paper, we propose the first comprehensive study on 210 Kubernetes operator bugs collected from 36 popular open-source operators developed by official teams of the managed systems, commercial vendors and open-source communities. These operators are adopted for managing various important cloud systems, e.g., Cassandra [1], MySQL [6], Redis [3], TiDB [7], etcd [5], MongoDB [2], Prometheus [4], etc. We thoroughly study these bugs and try to answer the following research questions.

- **RQ1 (Root cause):** What are root causes for operator bugs?
- **RQ2 (Bug manifestation):** How do operator bugs manifest themselves? How are operator bugs triggered?
- **RQ3 (Bug impact):** What impacts do operator bugs have?
- **RQ4 (Bug fixing):** How are operator bugs fixed?
- **RQ5 (Detection capability of existing approaches):** How effectively can existing approaches detect operator bugs?

Through the in-depth investigation on Kubernetes operator bugs against the above five research questions, we obtain some interesting findings. We summarize some main findings as follows.

- Operator bugs are caused by four types of bug patterns, i.e., incorrect access control configuration (4%), incorrect custom resource definition (9%), incorrect state observation and analysis (60%), and incorrect reconciliation (27%). These bug patterns motivate new approaches to detect operator bugs.
- Almost all operator bugs can be triggered by no more than three operation requests, no more than two state property changes, and no more than two faults. 86% of operator bugs can be triggered deterministically by executing a sequence of input events in a certain order. This indicates that we can detect operator bugs with relatively small workloads.
- 83% of operator bugs require updating specific state properties, or updating the properties with specific values. 10% of operator bugs require invalid operation requests. This indicates that we should consider these special cases when devising test scenarios.
- 54% of operator bugs only lead to silent failures such as unstable state and undesired state. This suggests that we need to design new test oracles to effectively detect silent operator bugs.
- 55% operator bugs can be detected by existing testing approaches [41, 62]. This indicates that we need to develop more effective operator bug testing and detection approaches.

In summary, we make the following main contributions.

- We present the first empirical study on Kubernetes operator bugs from five aspects, i.e., root causes, bug manifestations, bug impacts, bug fixing and detection capability of existing approaches. Our findings can open up new research directions in combating operator bugs.
- Our 210 documented operator bugs can serve as a bug benchmark for future work on combating Kubernetes operator bugs. We have made our collected operator bugs and analysis results available at https://doi.org/10.5281/zenodo.13340387.

## 2 Preliminaries

In this section, we introduce basic concepts used in the paper.

### 2.1 Kubernetes

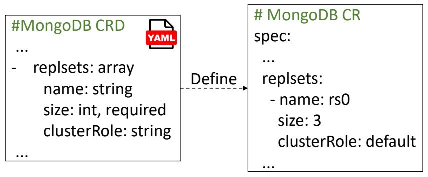
Kubernetes (also known as K8s) provides a powerful framework for orchestrating and managing containerized applications. The underlying physical resources (e.g., CPU, memory and storage) present in the Kubernetes cluster are abstracted into a set of logical resources that Kubernetes can manage and orchestrate, such as Pod (workload resource), Service (networking resource), Persistent Volume (storage resource), ConfigMap (configuration resource) and Secret (security resource). We also refer to these resources as Kubernetes *built-in resources*.

*Kubernetes objects* (we also refer to them as *state objects* or *resource objects*), are persistent entities used to manage and interact with the underlying resources in a Kubernetes cluster. These objects represent the state of the managed cluster and define how the cluster should behave. Kubernetes objects have nested spec and status properties, describing the desired and current states, respectively. The Kubernetes API server exposes the Kubernetes API to allow querying and manipulating the state of Kubernetes objects.

Kubernetes offers a set of *built-in controllers* that operate asynchronously, e.g., ReplicaSet controller and StatefulSet controller. These built-in controllers continuously watch the actual and the desired states of the resources they manage, and ensure that the actual state of the these resources matches the desired state defined by the relevant Kubernetes objects [20]. Built-in controllers provide essential features like scaling and self-healing to manage applications. However, for more complicated operation tasks that are specific to individual applications, such as application upgrades and configuration changes, users must write ad-hoc scripts for different one-off tasks, which can be cumbersome and error-prone.

### 2.2 Kubernetes Operator

To automate repeatable operation tasks for individual applications, the operators were introduced. Operators enable users to describe complex operation tasks in a declarative way. Additionally, they leverage custom controllers that encapsulate application-specific logic and operation knowledge to handle complex operation tasks.
■ **Components in Kubernetes operator:** Kubernetes operators are built upon the following two kinds of components.



```
#MongoDB CRD
 ...
-   replsets: array
      name: string
      size: int, required
      clusterRole: string
 ...
```

```
# MongoDB CR
spec:
  ...
  replsets:
   - name: rs0
     size: 3
     clusterRole: default
  ...
```

**Figure 2: An example of the custom resource definition (CRD) and an instance of the associated custom resource (CR).**

*Custom resource definition (CRD).* Custom resource definitions enable operator developers to extend the Kubernetes API by defining their own custom resources (CRs). A custom resource stores structured data in custom properties (or fields). Users can declaratively describe the desired state of a cluster by creating or modifying instances of custom resources defined by CRDs. As shown in Figure 2, CRDs define the schema of custom resources, listing all the configurations (i.e., properties) available to users of the operator. Additionally, CRDs define data types and validation rules, such as required fields, allowed value ranges, and regular expression patterns, for the custom resource properties.

*Custom controller.* Kubernetes operators utilize custom controllers to monitor and reconcile the desired state of custom resources. Custom controllers are typically implemented as Kubernetes controllers. They follow the control loop principle shown in Figure 1, and extend the functionality of Kubernetes to applications by leveraging the Kubernetes API and built-in controllers.

■ **Operator pattern:** Kubernetes operators follow a declarative approach to manage the applications running on Kubernetes. We use Figure 3 to illustrate how Kubernetes operators work.

*Step 1: Deploy and configure operators.* Operator developers first need to set up the environment for operators. After deploying the operator's custom controller on Kubernetes, developers need to configure API access control for operators to ensure that the operator has the appropriate permissions to access the relevant entities (①). This involves creating objects related to API access control, such as Roles and RoleBindings.

Then developers need to register CRDs associated with the custom controller to the API server (②). CRDs are typically described in YAML or JSON manifest files. Once the manifest file is applied, a corresponding CRD object will be created. Then the API server can serve custom resources defined by the CRD, and enable users to interact with custom resources through the Kubernetes API.

*Step 2: Observe and analyze changes.* Once the operator is ready, the custom controller continuously watches for changes in the resources it is responsible for (③). Users declare a desired system state by creating or modifying instances of custom resources (④). The API server is responsible for validating the operations that change a CR, ensuring that the CR adheres to specific rules and constraints defined in the CRD, and forwarding change events to the operator (⑤). When changes are observed, the operator analyzes the desired state and current state of the managed cluster to determine whether a reconciliation should be applied.

*Step 3: Reconcile the managed cluster.* If the current state of the cluster deviates from the desired state, the operator starts making changes aiming to align the current state with the desired state. The

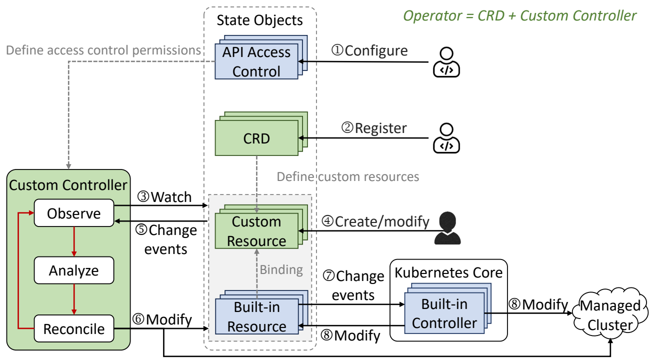


**Figure 3: The operator pattern in Kubernetes. CRD refers to Custom Resource Definition.**

operator can modify the state of some Kubernetes objects including custom resources and built-in resources such as Pods (⑥). These changes can trigger another round of the operator's reconciliation process, or activate the reconciliation process of built-in controllers (⑦). Ultimately, the actual state of the managed cluster will be reconciled to the desired state (⑧).

The Kubernetes operator will continuously repeat the above observing, analyzing and reconciling process. In this paper, we use *reconciliation iteration* to refer to a single cycle of the reconciliation process performed by the operator.

## 2.3 Vulnerabilities in Kubernetes Operators

The correctness of operators is critical to cloud system reliability. Although Kubernetes operators are designed to be resilient, the complex operation scenarios make operators prone to errors. First, users can frequently change desired states with different goals. The desired states declared by users can be either valid or invalid. Operators may fail to reconcile the system state due to overlooking certain operation scenarios. Second, the operator may be incorrectly deployed or configured. For example, the custom resource definitions can be incorrect, or the custom controller might not have the necessary permissions, etc. Third, the reconciliation process of operators can involve intricate interactions and asynchronous issues. A single operator can coordinate multiple steps to reconcile the manged cluster to a desired state. It can also collaborate with multiple built-in controllers, and interact with external components (e.g., databases). Moreover, operators have to combat unpredictable environmental failures such as node crashes.

## 3 Methodology

In this section, we introduce how we collect (Section 3.1) and analyze (Section 3.2) bugs, and discuss threats to validity (Section 3.3).

## 3.1 Collecting Operator Bugs

We collect 210 real-world Kubernetes operator bugs in 36 popular open-source operator projects hosted on GitHub. Table 1 shows the statistics about our studied operators. The developers of these operators include official teams of the managed systems, commercial vendors and open-source communities. These operators cover different versions of Kubernetes and are used to manage a diverse set of systems, including distributed storage systems (e.g., Cassandra, etcd), database management systems (e.g., MySQL, TiDB), monitoring/tracing systems (e.g., Prometheus, Jaeger), data processing systems (e.g., Apache Kafka, Apache NiFi), machine learning platforms (e.g., Kubeflow), search engines (e.g., Elasticsearch), and so on. Operators involved in our study range from tens of thousands to millions of lines of code.

For the target operator projects, we select operator bugs via the following steps. (1) We utilize keywords such as "operator OR controller", "bug", "reconcile", "is:closed", "link:pr" and their variations to search potentially relevant issues from their corresponding issue repositories (e.g. Github, Percona). This search returns us with 3,049 issues. (2) For each issue, we manually inspect the issue description, developer comments, fixing patches, and rebuild bug scenarios step by step. To ensure the accuracy of our study and provide more meaningful conclusions, we filter out issues that lack clear bug descriptions, are not confirmed as bugs by developers or do not have available fixing information. We also exclude issues that are not related to the operator mechanism, issues caused by simple programming errors (e.g., typos), and issues that we cannot clearly understand or duplicate with existing bugs. We finally keep 210 operator bugs for further investigation.

## 3.2 Analyzing Operator Bugs

To answer our five research questions, we perform an in-depth analysis of the 210 operator bugs based on their issue descriptions, developer discussions, available fixing patches and source code. We further assign the bugs into different categories according to their root causes (Section 4), bug manifestations (Section 5), bug impacts (Section 6) and bug fixing (Section 7). Besides, we check whether each operator bug can be detected by existing approaches [41, 62], and analyze their detection capabilities (Section 8).

We adopt the open card sorting approach to build the categories for operator bugs' root causes, manifestation, impacts and fixing, which is widely-used in existing empirical bug studies [30, 34, 38, 43, 69, 70]. For each operator bug, we extract necessary information according to its bug report, and write down its detailed bug scenario step by step. For bug root causes, we create four initial categories based on the operator's lifecycle. During bug investigation process, we further reveal more interesting categories. For bug manifestations, we extract related information, e.g., the number of operation requests and faults, from the bug triggering process. For bug impacts, we build the categories based on the final impacts caused by operator bugs. For bug fixing, we extract necessary information from issue reports, and available fixing patches. To figure out whether an operator bug can be detected by existing approaches [41, 62], we carefully study these approaches, and check whether these approaches can support an operator bug's triggering conditions and design a proper oracle to identify the bug, theoretically.

During the above process, we try to assign each bug into existing categories in a dimension based on the extracted information. If we cannot find a suitable category for a bug, we create a new category. We also refine categories if necessary. All operator bugs are studied by all authors of the paper and reviewed through multiple rounds to ensure correctness and consistency.

**Table 1: Target Operators and Collected Operator Bugs**

| Operator | System | Dev. | # Stars | LOC | # Bug |
|---|---|---|---|---|---|
| IST/CassOp | Cassandra | Instaclustr | 237 | 10.9K | 3 |
| RabbitMQOp | RabbitMQ | Official | 778 | 27.8K | 5 |
| NifiOp | Apache NiFi | Community | 106 | 98.5K | 2 |
| KafkafOp | Apache Kafka | Strimzi | 4424 | 373.9K | 8 |
| OOS/CassOp | Cassandra | OOS | 185 | 49.4K | 4 |
| PCNA/MDBOp | MongoDB | Percona | 300 | 145.8K | 31 |
| CloudOp | Elasticsearch | Official | 2461 | 438.5K | 7 |
| VMOp | VictoriaMetrics | Official | 378 | 85.0K | 7 |
| PostgreDBOp | PostgreSQL | Zalando | 3936 | 47.1K | 8 |
| ZooKeeperOp | ZooKeeper | Pravega | 358 | 20.7K | 7 |
| ActionsOp | Github Actions | Official | 4166 | 97.9K | 7 |
| MinioOP | MinIO | Official | 1082 | 281.4K | 2 |
| OTELOp | OpenTelemetry | Official | 1035 | 117.6K | 4 |
| PromOp | Prometheus | Official | 8702 | 305.5K | 6 |
| XtradbOp | MySQL | Percona | 494 | 141.1K | 15 |
| KubeLogOp | Logging mech. | Official | 1477 | 130.4K | 6 |
| JaegerOp | Jaeger | Official | 981 | 119.7K | 5 |
| MysqlOp | MySQL | Percona | 95 | 70.5K | 4 |
| AwxOp | Ansible AWX | Official | 1130 | 14.0K | 1 |
| CMT/CassOp | Cassandra | Community | 12 | 51.0K | 2 |
| KubeflowOp | Kubeflow | Official | 1439 | 96.8K | 10 |
| K8S/CassOp | Cassandra | K8ssandra | 166 | 53.6K | 3 |
| SPH/RedisOp | Redis | Spotahome | 1436 | 45.1K | 1 |
| OCK/RedisOp | Redis | OCK | 663 | 302.1K | 4 |
| CockroachOp | Cockroach | Official | 264 | 35.9K | 2 |
| TidbOp | TiDB | Official | 1178 | 400.9K | 5 |
| OFC/MDBOp | MongoDB | Official | 1136 | 31.9K | 4 |
| KnativeOp | Knative | Official | 174 | 1884.0K | 3 |
| EtcdOp | Etcd | Official | 1737 | 13.1K | 1 |
| SplunkOp | Splunk | Official | 192 | 190.6K | 1 |
| ElasticsearchOp | Elasticsearch | Zalando | 351 | 16.1K | 2 |
| IstioOp | Istio | Banzai Cloud | 535 | 100.6K | 2 |
| OpenSearchOp | OpenSearch | Official | 343 | 41.6K | 16 |
| TerraformOp | Terraform | Official | 449 | 1035K | 4 |
| ArgocdOp | Argo CD | Official | 567 | 303.3K | 4 |
| FDBOp | FoundationDB | Official | 222 | 109.3K | 14 |

OOS refers to Orange Open Source, and OCK refers to Opstree Container Kit.

## 3.3 Threats to Validity

Kubernetes is considered the most representative and popular cluster management platform. It has become the de facto standard for deploying and managing cloud-native applications. We collect operator bugs from 36 popular Kubernetes operators developed by official teams of the managed systems, commercial vendors and open-source communities. These operators have tens to thousands of stars on GitHub, and are utilized for managing diverse systems with varying functionalities. We have not intentionally ignored any operator bugs in these Kubernetes operators. We believe that our studied operator bugs provide a representative sample of operator bugs in Kubernetes operators.

As an empirical and qualitative bug study, our study results rely on the understanding of the researchers involved, which can introduce implicit bias towards individual researchers' expertise. We take several measures to mitigate this threat. We adopt widely-used empirical bug analysis protocols in existing studies [30, 34, 38, 43, 69, 70], e.g., how to collect and analyze bugs. All the operator bugs studied in this paper have been thoroughly discussed and

**Table 2: Root Causes of Operator Bugs**

| Bug Pattern | | #Bug |
|---|---|---|
| Incorrect access control configuration (4%) | | 9 |
| Incorrect CRD (9%) | Incorrect CRD specification | 10 |
| | Incorrect validation logic | 10 |
| Incorrect state observation and analysis (60%) | No observation of resource changes | 5 |
| | No analysis of resource changes | 38 |
| | Incorrect analysis logic | 29 |
| | Incorrect state identification | 53 |
| Incorrect reconciliation (27%) | Incorrect resource update | 19 |
| | K8s specification violation | 9 |
| | Reconciliation loop | 7 |
| | No synchronization of resource operations | 7 |
| | Incorrect order of resource operations | 3 |
| | Inconsistent reconciliation | 2 |
| | Others | 9 |

confirmed by all authors. In cases of disagreements regarding an operator bug, we conduct further investigations until a consensus is reached. For the sake of the reliability of our study results, we filter out those bugs that lack clear bug descriptions, as well as those that we cannot clearly understand or reach a consensus on. This is a necessary trade-off to maintain the accuracy of our study results.

## 4 Root Cause

We classify our studied operator into four main categories according to their root causes as show in Table 2.

### 4.1 Incorrect Access Control Configuration

To secure cluster resources, Kubernetes operators must be authorized with proper permissions to access relevant Kubernetes objects and interact with the managed system. To achieve this, developers need to configure objects related to API access control. For example, developers can create RBAC (Role-Based Access Control) objects to manage permissions based on user roles, and create ABAC (Attribute-Based Access Control) objects to manage permissions based on fine-grained attributes of the user. By creating such objects, developers can flexibly and precisely control what actions an operator can perform. Bugs in above processes can make operators fail to access certain entities, or introduce security vulnerabilities.

In our study, 9 operator bugs are caused by incorrect access control configuration. Take CMT/CassOp#30[1] as an example, the installation of the operator fails due to lacking access permissions to nodes. Another example is OpenSearchOp#717, in which the "clusterrole/codegen" marker is not appended to the controller, preventing the operator from accessing certain resources. Access control configuration is fundamental for running operators. However, these configurations can be complicated, involving defining fine-grained permissions, coordinating access control policies across multiple namespaces, ensuring seamless access control across third-party integrations, and adapting the access control configuration to the changing requirements of operators. Developers may easily overlook some important settings.

[1]All discussed issues contain hyperlinks, and are clickable.

**Finding 1**: *4% (9/210) of operator bugs are caused by incorrect access control configuration, preventing operators from operating Kubernetes resources.*

### 4.2 Incorrect Custom Resource Definition

The custom resource definition (CRD) serves as a schema or blueprint for the custom resources that it defines, declaring the expected properties (or fields), data types, and validation rules that the custom resources must conform to. When creating or modifying a custom resource, it must adhere to the structure and properties outlined in the associated CRD. However, the specification of the CRD and the associated CR validation logic can be incorrect. There are two bug scenarios here.

**Incorrect CRD specification (10 bugs).** A CRD may not be well-specified, which can prevent the operator from being functional, lead to inconsistent behaviors or result in unexpected errors. For example, the properties in CRDs can be specified with incorrect data types, incorrect default values and incorrect validation rules. Operator developers may overlook the specification of certain properties they care about, or include redundant properties in the CRD. For example, in PostgreDBOp#838, the CRD incorrectly specifies the `s3_force_path_style` property as a string type, when it should be a boolean. In another bug RabbitMQOp#404, the `replicas` property is specified as optional in the CRD. However, users cannot create a CR without specifying the `replicas` property.

**Incorrect validation logic (10 bugs).** When a custom resource is created, modified, or deleted, the Kubernetes API server performs the basic validation based on the associated CRD, including verifying data types, validating required properties, etc. Except for the validation rules defined in CRDs, operators have the flexibility to implement additional validations for managing the CRs, e.g., ensuring the existence of a "customerId" in the value of the customer property. However, the additional validation logic implemented by operators may be missing or incorrect. In TerraformOp#18, the operator does not provide additional validation logic to ensure the value of the `output` property is a valid JSON string. When the operator tries to transform the value of the `output` property, which is an invalid JSON string, into JSON format, the operator encounters a null pointer exception. In PromOp#3942, the operator does not implement additional validation logic to check the value of the property `metricRelabeling.action`. As a result, users can upload an invalid action value, which can cause the operator to fail in loading the configuration.

**Finding 2**: *9% (20/210) of operator bugs are caused by incorrect CRDs, including incorrect CRD specifications and incorrect CR validation logic.*

### 4.3 Incorrect State Observation and Analysis

Once the operator is ready, it runs in an infinite loop, continuously watching and analyzing the changes to the state of both custom resources and built-in resources that it manages. Bugs that are caused by incorrect observation or analysis of change events can be categorized into four categories.

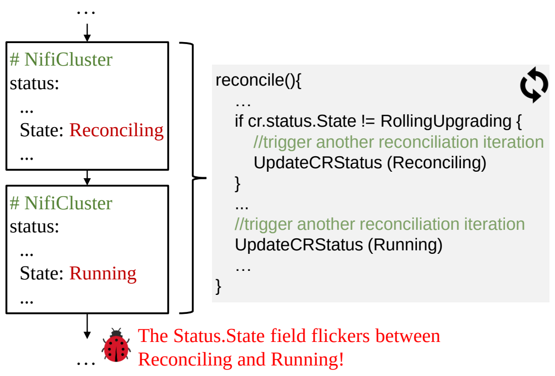
**No observation of resource changes (5 bugs).** To receive notifications about changes to concerned resources, an operator needs to register listeners on specific resources with the Kubernetes API server. If the operator fails to correctly watch specific resources, it will be unaware of any updates occurring to those resources. As a result, the operator will not be able to take reconciliation actions based on those updates. In VMOp#222, the operator does not register listeners on the ConfigMap and Secret resources. This makes the operator never receive notifications about changes to these two kinds of resources, thus cannot perform the corresponding reconciliation actions.

**No analysis of resource changes (38 bugs).** When the operator observes a resource change event, it will examine the differences between the current state and the desired state of the resource to accurately identify which properties of the resource have changed. Based on the differences found during the comparison, the operator determines the appropriate actions to bring the managed cluster into the desired state. Lacking the analysis on changes in certain resource properties will lead to the operator being unable to react to the changes, even the change events have been observed. In MinioOP#449, the operator does not analyze the change of the property `env` in CR. Therefore, even the operator can observe the change in CR, it will never take actions to reflect it in the cluster.

**Incorrect analysis logic (29 bugs).** The operator analyzes specific changes and determine which reconciliation actions should be performed, by comparing the current system state and the expected system state. However, the analysis logic can be incorrect.

- The state comparison logic can be incorrect. For example, in VMOp#298, the operator should add the values of two properties, i.e., `BasicAuth` and `TLSConfig`, to the vmalert parameters if they are not null. However, the operator developers mistakenly nest the `TLSConfig != nil` check within the `BasicAuth != nil` check. As a result, users cannot add the value of `TLSConfig` to the vmalert parameters when `BasicAuth` is set to null. 15 bugs belong to this category.
- The resources chosen for comparison can be incorrect or incomplete. For example, in K8SPS#227, there are two properties `MySQL.Expose` and `Proxy.Router.Expose` in the custom resource. The MySQL operator is unable to reconcile the modifications on the `MySQL.Expose` property, because the operator mistakenly uses `Proxy.Router.Expose` instead of `MySQL.Expose` when analyzing the changes. In another bug PCNA/MDBOp#578, when reconciling the ssl configuration, the MongoDB operator only checks if the ssl secret exists. If it does not exist, the operator creates the ssl secret and then proceeds to create the corresponding ssl-internal secret. However, an issue arises when the operator crashes between these two reconciliation actions. Upon restarting, the operator fails to recover the reconciliation process because it detects that the ssl secret already exists. 14 bugs belong to this category.

**Incorrect state identification (53 bugs).** Operators should accurately retrieve the system states for comparison, particularly the states of relevant resource objects. However, a buggy operator may encounter errors when retrieving resource states, or incorrectly identify erroneous or outdated resource states. Take VMOp#215 as an example, the operator overlooks the possibility of certain



```
reconcile(){
  ...
  if cr.status.State != RollingUpgrading {
    //trigger another reconciliation iteration
    UpdateCRStatus (Reconciling)
  }
  ...
  //trigger another reconciliation iteration
  UpdateCRStatus (Running)
  ...
}
```

**Figure 4: Bug NifiOp#119: Reconciliation loop. In this bug, the reconciliation actions continuously trigger new reconciliation iterations.**

properties being unassigned in the custom resource, and attempts to access sub-properties under a null property.

**Finding 3**: *60% (125/210) of operator bugs are caused by incorrect observation or analysis of state changes, i.e., no observation of resource changes, no analysis of resource changes, incorrect analysis logic and incorrect state identification.*

## 4.4 Incorrect Reconciliation

During the reconciliation process, operators typically utilize the Kubernetes API to manipulate Kubernetes objects, and collaborate with multiple built-in Kubernetes controllers that operate asynchronously to reconcile the desired system state. The bug scenarios caused by incorrect reconciliation are as follows.

**Incorrect resource update (19 bugs).** During reconciliation, the operator may mistakenly update unexpected resources, or update erroneous states induced from bugs or invalid CR declarations that bypass the validation mechanisms to the resources. For example, in ActionsOp#48, the operator mistakenly appends the same volume twice to the pod. This causes the operator to fail in creating the pod resource. In K8S/CassOp#1023, the operator stores the invalid pod name provided by users in `status.NodeReplacements`, causing the operator to become partially nonfunctional.

**K8s specification violation (9 bugs).** Operators rely on the underlying Kubernetes platform for managing the application cluster. During this process, operators should ensure that the reconciliation actions follow the specifications and requirements of the Kubernetes platform. Otherwise, the reconciliation will fail. In PromOp#4944, the Prometheus operator attempts to create a volume with a name that exceeds the 63-character limit imposed by the Kubernetes platform. This violates the naming constraints set by Kubernetes, leading to failures in the volume creation process.

**Reconciliation loop (7 bugs).** Kubernetes operators follow the control loop principle shown in Figure 1. Operators continuously watch state changes and perform reconciliation for the changes. When performing reconciliation actions, operators may update the state of some resource objects, which can potentially trigger another round of reconciliation. Bugs within this process can cause

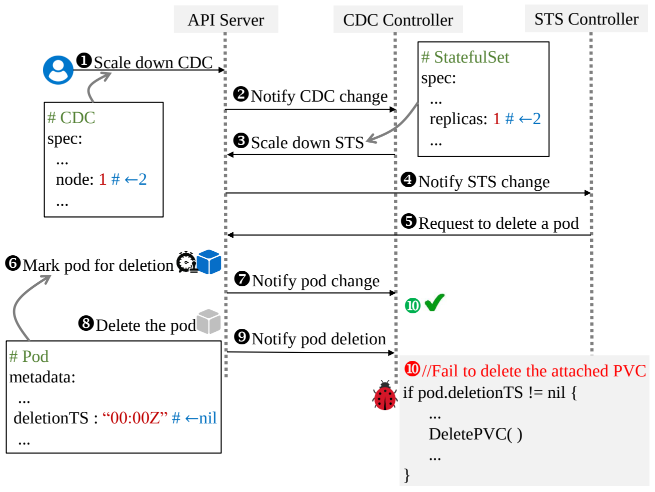


**Figure 5: Bug IST/CassOp#398: No synchronization of resource operations. CDC refers to CassandraDateCenter, STS refers to StatefulSet. In this bug, the quick deletion of pod causes its associated PVC to not be removed.**

operators to endlessly perform reconciliation, leaving the managed cluster unstable. For example, in NifiOp#119 shown in Figure 4, during each reconciliation iteration, the NifiCluster operator updates the `State` property of its associated custom resource, i.e., NifiCluster, to "Reconciling" if the property is not already set to "Reconciling". Subsequently, the operator updates `State` to "Running". This leads to the NifiCluster's state being modified in every iteration of the reconciliation loop, resulting in the `State` field flickering between "Reconciling" and "Running".

**No synchronization of resource operations (7 bugs).** In order to bring the managed cluster to the desired state, the operator may need to update multiple resources and coordinate with multiple controllers. However, the operations on certain resources have specific dependencies, requiring them to be executed in a particular order. Lacking synchronization for the operations on such resources can lead to an incorrect system state.

In IST/CassOp#398 shown in Figure 5, the user tries to scale down the cluster by changing the `node` property in the CR, i.e., CassandraDateCenter (CDC), from 2 to 1 (❶). Once the CDC change event is detected (❷), the CDC controller takes actions to modify the `replica` property of a built-in resource called StatefulSet (STS) from 2 to 1 (❸). The built-in STS controller is notified about the change of STS (❹), and initiates a request to the API server to delete a pod (❺). The pod to be deleted will first be marked for deletion by setting the `deletionTimestamp` field of the pod object to a non-nil value (❻), which will trigger a pod change event (❼). The CDC controller is expected to handle the event and remove the PVC attached to the pod marked for deletion (⑩). However, if the pod is deleted quickly (❽) before the CDC controller handles the pod change event, the CDC controller will fail to remove the attached PVC, since it cannot see the deletion marking event (⑩).

**Incorrect order of resource operations (3 bugs).** During the reconciliation process, although operations on multiple resources are executed in a synchronous manner, the execution order of these operations may be incorrect or inappropriate. This will result in a failed reconciliation, or prevent the operator from handling unexpected faults. Take FDB#1226 as an example, when users migrate the cluster from using DNS names to IP addresses, the FoundationDB operator removes the DNS service before replacing the transaction subsystem. However, the replacement process still requires the use of the DNS names, causing the reconciliation to get stuck.

In another example NifiOp#49, when the user specifies a new configuration, the NiFi operator handles this operation task by breaking it down into multiple fine-grained tasks that can be conducted across multiple reconciliation iterations. First, if Nifi operator detects inconsistencies between the CR and the ConfigMap resource, it will update the ConfigMap resource to align it with the CR, and then update "ConfigOutofSync" to CR's status field. Subsequently, the operator checks whether "ConfigOutofSync" is set. If true, the operator will delete and restart the pod for loading the new configuration. This reconciliation process works fine when there are no faults. However, when the NiFi operator crashes after updating the ConfigMap resource and before updating "ConfigOutofSync" to CR's status field, the restarted operator will not restart the pod to load the new configuration, since "ConfigOutofSync" is not set. To combat unexpected operator crashes, "ConfigOutofSync" should be set before updating the ConfigMap resource.

**Inconsistent reconciliation (2 bugs).** A single operator can coordinate multiple steps and collaborate with multiple controllers to reconcile the system state. Inconsistent decisions can cause reconciliation conflicts. For example, in IST/CassOp#400, for a scale-down request, the operator decides to decommission the last pod returned by the Kubernetes API `List` and updates the pod's status to prevent it from being used during deletion. Then the operator relies on the built-in StatefulSet controller to delete a pod from the cluster, which will select the pod with the largest ordinal to delete. However, there is no guaranteed order in the `List` API by default. As a result, the operator may decommission a different pod than the one selected for deletion by the StatefulSet controller. This can cause the deletion operation to be blocked forever.

**Others (9 bugs).** We categorize the remaining operator bugs caused by incorrect reconciliation logic as others. Take PromOp#3801 as an example, when creating a group of shards, the operator mistakenly exits the loop after creating the first shard. As a result, the remaining shards are never created.

**Finding 4**: *27% (56/210) of operator bugs are caused by incorrect reconciliation, e.g., incorrect resource update, K8s specification violation, reconciliation loop, no synchronization of resource operations, and incorrect order of resource operations.*

## 5 Bug Manifestation

We study the input conditions for triggering an operator bug, e.g., the number of operation requests and faults (Section 5.1). We also discuss the complexity of the bug triggering process, e.g., the number of resources and controllers involved during the bug manifestation process (Section 5.2). Finally, we discuss whether operator bugs can be triggered deterministically (Section 5.3).

## 5.1 Input Conditions

To trigger an operator bug, we need to initiate some operation requests, e.g., changing one or more property values in a custom resource or a built-in resource. As shown in Figure 6a, We find that most of the bugs (84%) involve no more than 2 operation requests. Specifically, 14 bugs involve operation requests on built-in resources. All the bugs require at least one operation request, i.e., the create CR request for deploying the cluster. In addition to the cluster deployment request, 18% of operator bugs require other operation requests that create or delete a resource object, and 50% of bugs also require the operation requests that change one or more properties of a resource object. As shown in Figure 6b, most of the bugs (83%) require the change of no more than 2 properties. For the bugs that require modification requests, an average of 3.64 property values are changed per bug.

We further divide the operator bugs into three categories according to the complexity of involved operation requests. (1) 17% of operator bugs only require simple operations, e.g., change the value of any single property of a resource object, or delete a resource object. (2) 30% of operator bugs require changing the values of specific resource properties, adding/deleting specific properties in the resource object, or creating resource objects with specific properties. (3) 53% of operator bugs require creating/updating resource objects with specific property values. Some of them (34 bugs) involve specific constraints among multiple property values.

We also investigate whether the performed operation requests are valid operations, i.e., declaring an achievable desired state with valid values. As a result, 21 bugs require invalid operations that would bring the managed system to an error state, e.g., providing a wrong pod name in K8S/CassOp#1023 and changing maxUnavailable from 0 to -1 in CloudOp#2034.

Faults is another input condition for triggering an operator bug. For example, in NifiOp#49 we mentioned before, the operator bug manifests when the NiFi operator crashes between updating the ConfigMap resource and updating "ConfigOutofSync" to CR's status. We cannot reproduce this bug without injecting the crash fault. As shown in Figure 6c, we find that only 19 bugs (9%) need faults, including node crashes, network delays and network disconnection. For these bugs, the required number of faults does not exceed 3.

**Finding 5**: *Most of the operator bugs can be triggered with no more than 3 operation requests. 83% of operator bugs require updating specific properties in the resource object, or updating the properties with specific values. 9% of operator bugs require faults such as node crashes.*

## 5.2 Complexity of the Bug Triggering Process

Before a bug manifests itself, the bug triggering process may involve multiple controllers (including custom controllers of Kubernetes operators and built-in controllers) manipulating multiple resources across multiple reconciliation iterations. The involved resources, controllers and reconciliation iterations are not the conditions for triggering an operator bug, but rather the characteristics that manifest during the bug triggering process. Such characteristics indicate the complexity of an operator bug.

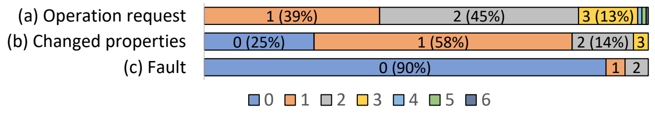


Figure 6: Distributions of input conditions.

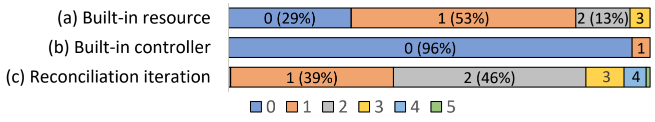


**Figure 7: Distributions of involved built-in resources, built-in controllers and reconciliation iterations during the bug manifestation process.**

Figure 7a shows the distribution of involved built-in resources for our studied operator bugs. 71% of bugs involve reading or writing built-in resources. The top 3 most commonly manipulated built-in resources are Pod, StatefulSet, and PersistentVolumeClaim. Around 50% of bugs involve the manipulation of these three types of resources.

Figure 7b shows the distribution of involved built-in controllers. Only a small portion of operator bugs (4%) involve built-in controllers. The results indicate that most of the operator bugs lie in the single operator rather than the collaboration with multiple built-in controllers.

Figure 7c shows the number of reconciliation iterations involved for custom controllers. 39% of bugs require only one reconciliation iteration. Most of the bugs (94%) involve no more than 3 reconciliation iterations during the bug triggering process.

**Finding 6**: *About haft of operator bugs involve manipulating built-in resources. Most of the operator bugs lie in the single operator. 39% of operator bugs involve only one reconciliation iteration.*

## 5.3 Bug Determinism

We determine whether an operator bug can be triggered deterministically by checking if the bug is guaranteed to manifest when the required sequence of input events occurs. For almost bugs (86%), we only need to explore the combination and permutation of input events, i.e., operation requests and faults, but no additional timing relationship. Among the 29 non-deterministic bugs, 6 bugs require specific ordering of internal events. 19 bugs require an input event to occur either before or after some internal events. The above two types of non-deterministic bugs can be reliably reproduced by controlling the execution order of the involved non-deterministic events. 4 bugs require the use of non-deterministic APIs, such as the `List` API, which cannot ensure the same result order across multiple calls by default. These bugs can be triggered by repeatedly executing their input events.

**Finding 7**: *86% of operator bugs can be triggered deterministically by executing a sequence of input events in a certain order.*

**Table 3: Fix Complexity**

| | 25th percentile | Median | 75th percentile | Max |
|---|---|---|---|---|
| # of days to confirm | 0 | 1 | 6 | 673 |
| # of days to fix | 1 | 7 | 25 | 395 |
| # of changed files | 1 | 2 | 4 | 21 |
| LOC of patch | 9 | 34 | 92 | 2091 |

## 6 Bug Impacts

To better understand how severe operator bugs are, we study their failure symptoms. We classify the studied operator bugs into five categories.

**Operator outage.** Kubernetes can automatically restart the crashed operator pod to mitigate unexpected operator crashes. However, 31 bugs cause operators to repeatedly crash and restart, resulting in the operators becoming unavailable.

**Operator malfunction.** 42 bugs cause operators to stop serving certain operation requests. For example, K8S/CassOp#315 causes the operator to fail to decommission nodes or perform a rolling restart. Note that 5% of bugs lead to silent operator malfunctions, that is, they do not report explicit errors, such as bugs caused by no observation of resource changes.

**Explicit system errors.** 34 bugs can cause reconciliation to fail, while also producing explicit system errors, e.g., node crashes, hangs, and error messages in execution logs. For example, PCNA/MDBOp#565 can cause the cluster to get stuck in the initializing state. In FDBOp#1185, the operator fails to create the Deployment resource and throws an error.

**Unstable state.** 10 bugs cause the managed cluster to constantly switch between different states, but no errors are reported explicitly, e.g., NifiOp#119 caused by reconciliation loop.

**Undesired state.** 93 bugs cause the manged cluster to ultimately reach an undesired state without reporting any errors. For example, PCNA/MDBOp#336 causes the status of the CR to be inconsistent with the cluster status.

To avoid double-counting, each bug is assigned to the first applicable category according to the impact classification order shown above. Overall, 54% of operator bugs only lead to silent failure symptoms, i.e., unstable state, undesired state and silent operator malfunction.

**Finding 8**: *Operator bugs can impact both operators and the managed cluster. 54% of operator bugs only lead to silent failures, i.e., unstable state, undesired state and silent operator malfunction.*

## 7 Bug Fixing

We study the complexity of operator bug fixing and how developers fix these bugs.

**Fix complexity.** We measure the fixing complexity with four metrics. (1) The number of days to confirm a bug. (2) The number of days to fix a bug. (3) The number of changed files. (4) The lines of code changed in the fixing patch. We show the results in Table 3. On average, operator bugs take 13 days to confirm, 28 days to fix, and involve 3 changed files, 97 lines of code.

**Fix strategy.** The fixes for operator bugs are tailored to their specific root causes and the underlying operators. For operator bugs caused by incorrect access control configuration, they are fixed by granting appropriate permissions to operators, e.g., CMT/CassOp#30 discussed in Section 4.1. Bugs caused by incorrect CRD are fixed by modifying the corresponding CRD specification or modifying the validation logic in the operator code, e.g., KubeLogOp#1002 and TerraformOp#18. For bugs caused by incorrect state observation and analysis, developers can fix them by registering event listeners on their concerned resource types or modifying the state analysis and reconciliation logic, e.g., retrieving the correct resource state for comparison in bug VMOp#222.

Operator bugs caused by incorrect reconciliation typically require fixing the related state analysis and reconciliation logic. Specifically, bugs caused by no synchronization of resource operations or incorrect order of resource operations can be fixed by adding synchronization mechanisms or adjusting the order of resource operations. For example, for IST/CassOp#398 discussed in Section 4.4, it was fixed by adding a finalizer for each Cassandra pod to ensure that a pod is only removed after its corresponding PVC resource has been deleted. Kubernetes ensures that a Kubernetes object can only be deleted after the finalizer that is associated with the object has been removed.

**Finding 9**: *The fix for operator bugs highly depends on the root causes and the underlying operators. Certain operator bugs, e.g., bugs caused by no synchronization of resource operations, can be fixed by a small set of strategies.*

## 8 Detection Capability of Existing Approaches

We investigate the effectiveness of the-state-of-the-art operator/controller testing approaches, Sieve [62] and Acto [41], in detecting operator bugs. Our study covers 210 operator bugs across 36 Kubernetes operator projects and multiple Kubernetes versions. Directly running these two tools to assess their detection capability for the studied bugs poses significant challenges. It would involve substantial efforts in setting up the necessary environments and require a massive amount of testing time. Therefore, we have chosen to theoretically analyze the maximum detection capability of existing approaches for the operator bugs we studied.

Sieve [62] can automatically test the reliability of operators by perturbing the operator's view of the current cluster state. It depends on user-provided test workloads to drive the target operator, collects reference traces in the absence of faults, and then generates test plans to perturb the operator's view of the current cluster state based on the traces and predefined fault injection rules. Sieve detects bugs by checking obvious failure symptoms such as crashes and error messages, and by comparing cluster states between the fault-free and faulty runs. Since Sieve relies on specific fault conditions (i.e., node crash, network delay and network disconnection) to detect operator bugs, we only assess its detection capability for fault-related operator bugs. If an operator bug requires fault scenarios not supported by Sieve's fault injection rules, e.g., pod crashes, we consider that Sieve cannot detect the bug.

Acto [41] tests the correctness of Kubernetes operators by generating a sequence of desired cluster state declarations. Acto predefines some representative operation scenarios based on property semantics and property data types. Then Acto tries to generate syntactically valid and semantically meaningful desired state declarations, to cover all exposed properties in the CRD and pre-defined representative operation scenarios. Acto detects bugs by checking explicit failure symptoms, mismatches between the reconciled system state and the declared desired state, and inconsistencies between different state transitions to the same end states. If an operator bug satisfies one of the following conditions, Acto cannot detect it. (1) The bug requires faults. (2) The bug requires operation scenarios beyond the representative ones defined by Acto, e.g., operation requests across multiple resources, resource creation/deletion request, changing properties that are not exposed in the CRD. (3) The bug causes failure symptoms that cannot be covered by Acto's test oracles, e.g., domain-specific failures.

We investigate all 210 operator bugs, and check whether they satisfy Sieve and Acto's preconditions. We find that for the 20 fault-related operator bugs, 18 bugs satisfy Sieve's preconditions, and can be detected by Sieve. 98 bugs satisfy Acto's preconditions, and can be detected by Acto. In theory, the combination of Acto and Sieve can detect 116 operator bugs. However, we should note that Acto cannot cover all possible system states and transitions. Acto also uses a randomized approach to explore the combinations of changes in different properties. Additionally, Acto and Sieve do not control the execution order of non-deterministic internal events. This makes existing approaches inefficient in generating operation scenarios that can trigger bugs, as well as detecting nondeterministic operator bugs.

**Finding 10**: *55% of operator bugs can be detected by existing operator/controller testing approaches.*

## 9 Lessons Learned

In this section, we discuss lessons learned, implications to existing approaches and opportunities for new research in combating Kubernetes operator bugs. As the first empirical study on Kubernetes operator bugs, we believe that our findings can help operator developers and researchers to improve operators' reliability.

### 9.1 Operator Bug Detection and Fixing

Operator bugs can cause severe consequences, impacting both the operator and the managed application (Finding 8). Resolving operator bugs is of great significance for improving the reliability of cloud applications. Our study reveals that operator bugs can be summarized into 4 kinds of bug patterns, i.e., incorrect access control configuration (Finding 1), incorrect custom resource definition (Finding 2), incorrect state observation and analysis (Finding 3), and incorrect reconciliation (Finding 4). These bug patterns shed new light and guidance on operator bug detection.

**Permission-guided bug detection.** To perform reconciliation actions on Kubernetes resources, operators should be authorized with proper permissions. Finding 1 implies that incorrect access control configuration for operators can introduce operator bugs. This suggests that it is possible to detect operator bugs by analyzing the components related to permissions in the operator. The key challenge is to analyze the permissions necessary for operator reconciliation and examine whether the actual permissions granted to the operator are appropriate and sufficient for it to properly operate on the necessary resources.

**Specification-guided bug detection.** Correct operators must not violate various specification, e.g., the specification of custom resource definition and the specification of Kubernetes platform. Finding 2 and Finding 4 imply that specification violation can introduce operator bugs. We can summarize specification violation rules, and detect these bugs based on the rules.

**Resource-dependency-guided bug detection.** In an application cluster, Kubernetes resources may have dependencies, and operations on these resources should be synchronized and executed in a certain order. Finding 4 implies that no synchronization of resource operations and incorrect order of resource operations can introduce operator bugs. It is necessary for us to build a knowledge graph regarding the dependencies of Kubernetes resources and establish rules for resource operations. Based on the knowledge graph and rules, we can effectively detect operator bugs.

**Control-loop-guided bug detection.** Kubernetes operators follow the Kubernetes control loop principle. An operator runs as an infinite loop that continuously observing, analyzing and reconciling the state of the managed cluster. The preceding steps in this process will impact the execution of subsequent steps. Finding 3 and Finding 4 show that operator bugs can be caused by no observation of state changes, no analysis of state changes, and reconciliation loop where reconciliation actions trigger new state changes and another round of reconciliation. This indicates that we can detect bugs caused by the lack of certain control loop steps and bugs caused by incorrect read-write patterns through the analysis of the operator code.

**Operator bug fixing.** Finding 9 shows that the fixes for operator bugs are highly upon their root causes and the underlying operator. However, operator bugs introduced by certain root causes, e.g., incorrect CRD, no observation of resource changes and no synchronization of resource operations, are fixed by a small set of strategies with a few lines of code. This finding implies unique research opportunities for automated fixing of operator bugs.

### 9.2 Operator Testing

**Workload generation.** A well-designed operator workload generation approach can greatly improve the effectiveness of exposing operator bugs. This involves generating a sequence of operation requests. Each operation request can declare a new desired state aimed at a single resource by changing the value of one or more properties in the current state. A workload may contain changes on multiple resources, which may trigger the execution of multiple operators and built-in controllers. Even though Acto can generate representative operation scenarios [41], it is incomplete and can miss bugs (Finding 10). We need to generate workloads that support more features, such as modifications to multiple resources. Finding 7 shows that most operator bugs can be triggered by no more than 3 operation requests and no more than 3 property changes. This indicates that operator bugs usually follow the small scope hypothesis, and can be effectively detected with small workloads.

**State space exploration.** Validating the correctness of operators is extremely challenging due to the enormous system state space resulting from the combination of all possible cluster states, all possible declarations of desired state, and all possible interleavings of non-deterministic events, including operation requests, faults, and internal system events. Existing approaches explore representative operation scenarios [41] and representative fault scenarios [62] separately, which cannot systematically explore the entire state space and thus fail to comprehensively detect operator bugs (Finding 10). Further more, Acto [41] uses a randomized approach to explore the combinations of changes in different properties. We need to consider the combination of multiple factors that affect operator behaviors, and work out more effective strategies for effectively exploring the system state space.

**Oracle design.** Finding 8 shows that 54% of operator bugs causes implicit failure symptoms, e.g., unstable state and undesired state. To detect these bugs, we need to design more effective oracles. Test oracles adopted by exist testing approaches [41, 62] can effectively capture a lot of operator bugs. However, Finding 10 shows that exist approaches can overlook certain failure symptoms, and fail to detect certain bugs. This indicates that researchers need to develop new test oracles to reveal such silent operator bugs.

The reconciliation logic of operators can be complicated. A single operator can coordinate multiple steps and collaborate with multiple built-in Kubernetes controllers, leading to intricate interactions and potential conflicts. This easily leads to operator developers introducing various problems during the coding process. Fortunately, operator programs follow the control loop principle and specific resource operation rules. This provides an opportunity for us to propose new approaches to assist developers in generating operator programs. These approaches can help simplify the coding process, improve code quality, and enhance the overall effectiveness of operators.

## 10 Related Work

We introduce related works that we have not discussed yet.

**Empirical bug studies.** Barletta et al. [29] performed a qualitative field failure data analysis of 81 real-world Kubernetes incidents reported in online sources. They analyze how Kubernetes fails by building a Fault-Error-Failure dependency chain for Kubernetes. Hassan et al. [44] conduct a qualitative analysis with 5,110 state reconciliation defects mined from Ansible orchestrator. Ansible orchestrator and Kubernetes operator both uses state reconciliation to maintain the consistency between the desired state and actual state. However, Ansible orchestrator focuses on managing infrastructure state through executing a series of tasks, while Kubernetes operators take a more declarative approach to manage custom resources and their relationships within a Kubernetes cluster.

There are also some empirical studies for other specific types of bugs, e.g., concurrency bugs [52, 68], fault related bugs in distributed systems [27, 28, 36, 38], exception-related bugs [30], bugs in deep learning systems [32, 42], bugs in database systems [35], bugs in cloud systems [43] and so on. We adopt similar study methodologies to existing studies in the process of bug selection, bug analysis, and bug categorization. To the best of our knowledge, our work is the first comprehensive study on Kubernetes operator bugs.

**Kubernetes controller verification.** Sun et al. [63] formalize a general property called eventually stable reconciliation (ESR) for controller correctness. Then they propose a framework called Anvil for developing controllers implemented in Rust and verifying that the controller implementation satisfies ESR for all executions. Kivi [54] models the high-level logic of Kubernetes controllers, and verifies whether controllers in a specific deployment can violate a set of user-provided properties by exhaustively model checking the interactions among controllers and events at the model level.

Different from previous discussed automated testing approaches [41, 62] in Section 8, those verification approaches aim to ensure the correctness of Kubernetes controllers with respect to one or more properties at the model or implementation level. They can preclude bugs before deploying controllers and uncover issues that can be missed by existing testing approaches. However, these verification approaches typically require more manual efforts, e.g., developer-provided proofs tailored for individual controllers.

**Distributed system testing and bug detection.** A lot of approaches, e.g., fuzzing approaches [39, 59], model-based testing approaches [37, 45, 53], implementation-level model checkers [51, 58, 61, 71, 72], model checking guided testing approaches [37, 65, 67, 73], and pattern-based bug detection approaches [31, 40, 55–57] have been proposed to test distributed systems and detect specific distributed system bugs. These approaches can be generalized to Kubernetes operator/controller. However, we must consider the characteristics of the operator and operator bugs to improve the effectiveness of these approaches in detecting operator bugs.

## 11 Conclusion

The complex, dynamic, and distributed nature of the overall system makes Kubernetes operators prone to operator bugs, which can lead to severe consequences. We conduct the first comprehensive study on 210 Kubernetes operator bugs from 36 open-source operator projects. We obtain many interesting findings and lessons with respect to bug root causes, manifestations, impacts and fixes. We hope our study can inspire more actions from diverse researchers in the areas of operator reliability.

## 12 Data Availability

The dataset of our paper is publicly available at Zenodo [26].

## Acknowledgments

This work was partially supported by National Natural Science Foundation of China (62302493), Major Project of ISCAS (ISCAS-ZD-202302) and China Southern Power Grid Company Limited under Project 037800KK52220005.

## References

[1] 2009. *apache/cassandra.* Retrieved March 31, 2024 from https://github.com/apache/cassandra
[2] 2009. *mongodb/mongo.* Retrieved March 31, 2024 from https://github.com/mongodb/mongo
[3] 2009. *redis/redis.* Retrieved March 31, 2024 from https://github.com/redis/redis
[4] 2012. *prometheus/prometheus.* Retrieved March 31, 2024 from https://github.com/prometheus/prometheus
[5] 2013. *etcd-io/etcd.* Retrieved March 31, 2024 from https://github.com/etcd-io/etcd
[6] 2014. *mysql/mysql-server.* Retrieved March 31, 2024 from https://github.com/mysql/mysql-server

[7] 2015. *pingcap/tidb*. Retrieved March 31, 2024 from https://github.com/pingcap/tidb
[8] 2016. *coreos/etcd-operator*. Retrieved March 31, 2024 from https://github.com/coreos/etcd-operator
[9] 2016. *prometheus-operator/prometheus-operator*. Retrieved March 31, 2024 from https://github.com/prometheus-operator/prometheus-operator
[10] 2016. *strimzi/strimzi-kafka-operator*. Retrieved March 31, 2024 from https://github.com/strimzi/strimzi-kafka-operator
[11] 2018. *elastic/cloud-on-k8s*. Retrieved March 31, 2024 from https://github.com/elastic/cloud-on-k8s
[12] 2018. *percona/percona-xtradb-cluster-operator*. Retrieved March 31, 2024 from https://github.com/percona/percona-xtradb-cluster-operator
[13] 2019. *How a Production Outage Was Caused Using Kubernetes Pod Priorities*. Retrieved March 30, 2024 from https://grafana.com/blog/2019/07/24/how-a-production-outage-was-caused-using-kubernetes-pod-priorities/
[14] 2019. *Outage post-mortem*. Retrieved March 30, 2024 from https://updates.moonlightwork.com/outage-post-mortem-87370
[15] 2020. *Cloud Native Computing Foundation Operator White Paper*. Retrieved March 31, 2024 from https://www.cncf.io/wp-content/uploads/2021/07/CNCF_Operator_WhitePaper.pdf
[16] 2020. *Kubernetes Outages with real-world case studies*. Retrieved March 30, 2024 from https://kubevious.io/blog/post/kubernetes-outages-with-real-world-case-studies
[17] 2020. *O'Reilly: Kubernetes Operators: Automating the Container Orchestration Platform*. Retrieved March 31, 2024 from https://www.redhat.com/en/resources/oreilly-kubernetes-operators-automation-ebook
[18] 2021. *percona/percona-server-mysql-operator*. Retrieved March 31, 2024 from https://github.com/percona/percona-server-mysql-operator
[19] 2022. *CNCF Annual Survey 2022 | Cloud Native Computing Foundation*. Retrieved March 31, 2024 from https://www.cncf.io/reports/cncf-annual-survey-2022/
[20] 2023. *Controllers | Kubernetes*. Retrieved March 31, 2024 from https://kubernetes.io/docs/concepts/architecture/controller/
[21] 2023. *Kubernetes in the wild report 2023*. Retrieved March 31, 2024 from https://www.dynatrace.com/news/blog/kubernetes-in-the-wild-2023/
[22] 2023. *The State of Kubernetes 2023*. Retrieved March 31, 2024 from https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/docs/vmware-ebook-state-of-kubernetes.pdf
[23] 2024. *Kubernetes - statistics & facts*. Retrieved March 31, 2024 from https://www.statista.com/topics/8409/kubernetes/#topicOverview
[24] 2024. *Kubernetes Document*. Retrieved March 31, 2024 from https://kubernetes.io/docs/home/
[25] 2024. *vSphere: virtualization solution for data center consolidation and enhanced application availability*. Retrieved March 30, 2024 from https://www.vmware.com/products/vsphere.html
[26] 2024. *Zenodo*. Retrieved Aug 12, 2024 from https://doi.org/10.5281/zenodo.13340387
[27] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 351–368. https://doi.org/10.5555/3488766.3488786
[28] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 51–68. https://doi.org/10.5555/3291168.3291173
[29] Marco Barletta, Marcello Cinque, Catello Di Martino, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2024. Mutiny! How Does Kubernetes Fail, and What Can We Do About It?. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.
[30] Haicheng Chen, Wensheng Dou, Yanyan Jiang, and Feng Qin. 2019. Understanding Exception-Related Bugs in Large-Scale Cloud Systems. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 339–351. https://doi.org/10.1109/ASE.2019.00040
[31] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 536–547. https://doi.org/10.1145/3324884.3416548
[32] Junjie Chen, Yihua Liang, Qingchao Shen, Jiajun Jiang, and Shuochuan Li. 2023. Toward Understanding Deep Learning Framework Bugs. *ACM Trans. Softw. Eng. Methodol.* 32, 6 (2023), 31 pages. https://doi.org/10.1145/3587155
[33] ContainerDays. 2019. Moving to Kubernetes: the Bad and the Ugly - Maxime Lagresle. https://www.youtube.com/watch?v=MoIdU0J0f0E
[34] Ziyu Cui, Wensheng Dou, Yu Gao, Dong Wang, Jiansen Song, Yingying Zheng, Tao Wang, Rui Yang, Kang Xu, Yixin Hu, et al. 2024. Understanding Transaction Bugs in Database Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 947–947. https://doi.org/10.1145/3597503.3639207
[35] Ziyu Cui, Wensheng Dou, Yu Gao, Dong Wang, Jiansen Song, Yingying Zheng, Tao Wang, Rui Yang, Kang Xu, Yixin Hu, Jun Wei, and Tao Huang. 2024. Understanding Transaction Bugs in Database Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 1–13. https://doi.org/10.1145/3597503.3639207
[36] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. 2018. Understanding Real-World Timeout Problems in Cloud Server Systems. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 1–11. https://doi.org/10.1109/IC2E.2018.00022
[37] A Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. 2020. eXtreme Modelling in Practice. *Proceedings of the International Conference on Very Large Data Bases (VLDB)* 13, 9 (2020), 1346–1358. https://doi.org/10.14778/3397230.3397233
[38] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ECSE/FSE)*. 539–550. https://doi.org/10.1145/3236024.3236030
[39] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. 2023. Coverage Guided Fault Injection for Cloud Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 2211–2223. https://doi.org/10.1109/ICSE48619.2023.00186
[40] Yu Gao, Dong Wang, Qianwang Dai, Wensheng Dou, and Jun Wei. 2022. Common Data Guided Crash Injection for Cloud Systems. In *Proceedings of the International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 36–40. https://doi.org/10.1145/3510454.3516852
[41] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. 2023. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 96–112. https://doi.org/10.1145/3600006.3613161
[42] Hao Guan, Ying Xiao, Jiaying Li, Yepang Liu, and Guangdong Bai. 2023. A Comprehensive Study of Real-World Bugs in Machine Learning Model Optimization. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 147–158. https://doi.org/10.1109/ICSE48619.2023.00024
[43] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. 1–14. https://doi.org/10.1145/2670979.2670986
[44] Md Mahadi Hassan, John Salvador, Shubhra Kanti Karmaker Santu, and Akond Rahman. 2024. State Reconciliation Defects in Infrastructure as Code. In *Proceedings of the ACM International Conference on the Foundations of Software Engineering (FSE)*. 1865–1888. https://doi.org/10.1145/3660790
[45] Beom Heyn Kim, Taesoo Kim, and David Lie. 2022. Modulo: Finding Convergence Failure Bugs in Distributed Systems with Divergence Resync Models. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 383–398.
[46] KubeCon. 2019. Keep the Space Shuttle Flying: Writing Robust Operators - Illya Chekrygin, Upbound. https://www.youtube.com/watch?v=uf97lOApOv8
[47] KubeCon. 2019. Storage on Kubernetes - Learning From Failures - Hemant Kumar & Jan Šafránek, Red Hat. https://www.youtube.com/watch?v=zuGjw595OiQ
[48] KubeCon. 2019. Writing a Kubernetes Operator: the Hard Parts - Sebastien Guilloux, Elastic. https://www.youtube.com/watch?v=wMqzAOp15wo
[49] KubeCon. 2022. How a Couple of Characters (and GitOps) Brought Down Our Site - Guy Templeton & Stuart Davidson, Skyscanner. https://www.youtube.com/watch?v=FiEm2zOuHsg
[50] KubeCon. 2022. Preventing Controller Sprawl From Taking Down Your Cluster - When a Scalable Pattern Stops Being Scalable - Madhu C.S., Robinhood Markets. https://www.youtube.com/watch?v=fu5GXo7jmV0
[51] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 399–414. https://doi.org/10.5555/2685048.2685080
[52] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 517–530. https://doi.org/10.1145/2872362.2872374
[53] Yishuai Li, Benjamin C Pierce, and Steve Zdancewic. 2021. Model-based Testing of Networked Applications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 529–539. https://doi.org/10.1145/3460319.3464798
[54] Bingzhe Liu, Gangmuk Lim, Ryan Beckett, and P. Brighten Godfrey. 2024. Kivi: Verification for Cluster Management. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*. 509–527. https://www.usenix.org/conference/atc24/presentation/liu-bingzhe
[55] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 677–691. https://doi.org/10.1145/3093337.3037735

[56] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. *ACM SIGPLAN Notices* 53, 2 (2018), 419–431. https://doi.org/10.1145/3296957.3177161

[57] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 114–130. https://doi.org/10.1145/3341301.3359645

[58] Jeffrey F Lukman, Huan Ke, Cesar A Stuardo, Riza O Suminto, Daniar H Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, et al. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 1–16. https://doi.org/10.1145/3302424.3303986

[59] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. 2023. Greybox Fuzzing of Distributed Systems. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1615–1629. https://doi.org/10.1145/3576915.3623097

[60] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 351–364. https://doi.org/10.1145/2465351.2465386

[61] Jiri Simsa, Randy Bryant, and Garth Gibson. 2010. dBug: Systematic Evaluation of Distributed Systems. In *Proceedings of the International Workshop on Systems Software Verification (SSV)*. https://doi.org/10.5555/1929004.1929007

[62] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing for Cluster Management Controllers. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 143–159. https://www.usenix.org/conference/osdi22/presentation/sun

[63] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. 2024. Anvil: Verifying Liveness of Cluster Management Controllers. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 649–666. https://www.usenix.org/conference/osdi24/presentation/sun-xudong

[64] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, et al. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 787–803. https://doi.org/10.5555/3488766.3488811

[65] Ruize Tang, Xudong Sun, Yu Huang, Wei Yuyang, Lingzhi Ouyang, and Xiaoxing Ma. 2024. SandTable: Scalable Distributed System Model Checking with Specification-Level State Exploration. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 736–753. https://doi.org/10.1145/3627703.3650077

[66] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 1–17. https://doi.org/10.1145/2741948.2741964

[67] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. 2023. Model Checking Guided Testing for Distributed Systems. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. 127–143. https://doi.org/10.1145/3552326.3587442

[68] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A Comprehensive Study on Real World Concurrency Bugs in Node.js. In *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 520–531. https://doi.org/10.5555/3155562.3155628

[69] Tao Wang, Qingxin Xu, Xiaoning Chang, Wensheng Dou, Jiaxin Zhu, Jinhui Xie, Yuetang Deng, Jianbo Yang, Jiaheng Yang, Jun Wei, et al. 2022. Characterizing and Detecting Bugs in WeChat Mini-Programs. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 363–375. https://doi.org/10.1145/3510003.3510114

[70] Tao Wang, Kangkang Zhang, Wei Chen, Wensheng Dou, Jiaxin Zhu, Jun Wei, and Tao Huang. 2022. Understanding Device Integration Bugs in Smart Home System. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 429–441. https://doi.org/10.1145/3533767.3534365

[71] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. 2009. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. https://doi.org/10.1145/1731060.1731062

[72] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 213–228. https://doi.org/10.5555/1558977.1558992

[73] Yuqi Zhang, Yu Huang, Hengfeng Wei, and Xiaoxing Ma. 2024. Model-checking-driven explorative testing of CRDT designs and implementations. *Journal of Software: Evolution and Process* 36, 4 (2024), e2555. https://doi.org/10.1002/smr.2555

Received 2024-04-12; accepted 2024-07-03