Automatic Testing for Correct Operations of Cloud Systems

Abstract

Computer systems are increasingly being managed by operation programs (termed *operators*) that automate laborious, human-based operations. Cloud-native operators of modern management platforms like Kubernetes, Twine, and ECS implement declarative interfaces based on state reconciliation. An operation declares a desired system state and the operator automatically reconciles the system to the declared state.

Operator correctness is critical, given the impacts of system operations; bugs in operator code put systems in undesired or error states, with severe consequences. However, validating operator correctness is hard due to the enormous state space and complex operation interface. A correct operator must not only satisfy correctness properties of its own code, but also maintain managed systems in desired states. Unfortunately, end-to-end operator testing is significantly lagging behind.

We present Acto, the first automatic end-to-end testing technique for cloud-native operators. Acto uses a state-centric approach to test an operator together with a managed system. Acto continuously instructs an operator to reconcile a system to different states and checks if the system reaches those desired states. Acto models operations as state transitions and systematically realizes state-transition sequences to exercise supported operations in different scenarios. Acto's automated oracles check if a system's state is as desired. To date, Acto helped find 56 serious new bugs (40 confirmed and 28 fixed) in eleven Kubernetes operators with few false alarms.

1 Introduction

Cloud systems are growing in scale and demand beyond what human-based operation can reliably, continuously and efficiently manage. Modern cloud systems are increasingly being managed by operation programs (termed *operators* [2,45]) that automate labor-intensive operations. Operators of cloud management platforms like Kubernetes [37], Twine [76], and ECS [62] implement declarative interfaces based on *state reconciliation*. An operation declares the desired system state and the operator automatically reconciles the system from its current state to the declared state. This "cloud-native" operator pattern simplifies operations and improves efficiency.

The cloud-native operator pattern has led to a thriving ecosystem of high-quality, reusable operator code [50,51,56,70,72]. Take Kubernetes as an example: most cloud systems today have operators to manage them atop the Kubernetes platform. These operators automate important management tasks, e.g., software upgrades, configuration updates, and autoscaling. Even for the same cloud system, different operators

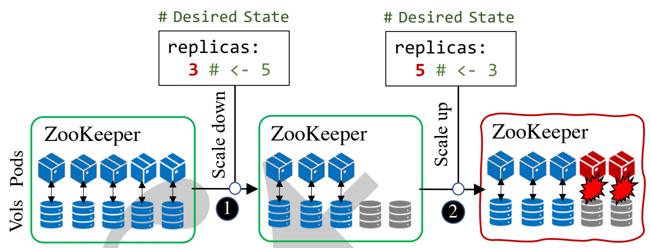


Figure 1: **A safety bug in ZooKeeperOp, a ZooKeeper operator, detected by our tool, Acto [28].** The bug manifests when the operator first scales down and then scales up the ZooKeeper cluster. Newly created pods fall into crash loops.

are implemented by commercial vendors and open-source communities to support different practices and environments.

The rapid development and deployment of operators make their quality assurance a pressing need—operation correctness is critical to system reliability [46,65]. A buggy operator can impair correctly implemented systems in production. Compared with human operator mistakes, which are major causes of system failures [36,46,63–65], bugs in operator code have more magnified impacts due to the nature of automation and widespread software reuse. In fact, buggy operators caused many recent production incidents [40,41,47,54,55,60,77].

Figure 1 shows a safety bug that our technique detects in a Kubernetes operator for managing ZooKeeper. When scaling down a ZooKeeper cluster, the operator only removes the pods, but not the data volumes attached to the pods. If the operator later scales up the ZooKeeper cluster, newly created pods will try to reuse old volumes. Due to membership inconsistencies between new pods and old volumes, the new ZooKeeper nodes fail to start. As a result, the ZooKeeper cluster cannot scale up and is thus vulnerable to overloads.

Figure 2 shows a liveness bug that we detect in an operator for managing TiDB. To update the affinity rule of a TiDB pod [1], the operator must stop the pod and reassign it with a new affinity rule. But, if the new affinity rule is not satisfiable, the pod cannot be reassigned. In this case, the buggy operator waits forever for the assignment to complete. To make matters worse, the operator cannot be restored by resetting the affinity rule: the operator does not carry out new operations before the ongoing one completes, to avoid race conditions.

These two bugs are among a myriad of operator bugs that affect operation correctness. Compared with the management platform (e.g., Kubernetes) and the managed system (e.g., ZooKeeper and TiDB), operator code is often much less tested. For example, we find that existing operators rely mostly on unit tests which cannot check operation correctness end to end,

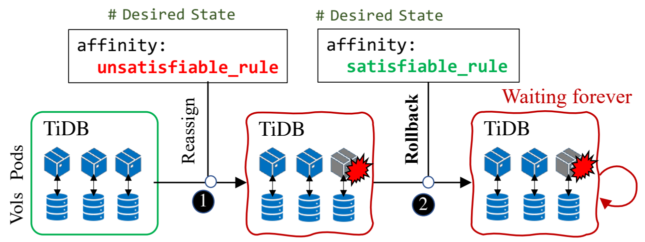


Figure 2: **A liveness bug in TiDBOp, a TiDB operator, detected by Acto [30].** If a declared affinity rule cannot be satisfied, TiDBOp enters an infinite waiting loop and the pod will never be assigned. TiDBOp cannot be recovered by rolling back with a satisfiable affinity rule.

i.e., if an operator reconciles the managed system to desired states. Some operators include a few end-to-end (e2e) tests but only cover small parts of the enormous system state space and the complex operations exposed by declarative interfaces.

We seek a practical testing technique that can test cloud-native operators end to end and can be readily applied to any types of operators for managing different systems. Unfortunately, existing automated test generation techniques like fuzzing [61, 83] or symbolic execution [38, 39] cannot effectively test operators end to end, since they neither model the semantics of operations nor reason about system states. In particular, operator bugs do not commonly manifest as crashes but they drive systems into undesired states (§6.1).

Technique. This paper presents Acto, the first automatic technique and tool for end-to-end testing of cloud-native operators. Acto automatically generates end-to-end tests to check three operation correctness requirements: the operator (1) always reconciles the managed system to desired states, (2) performs managed system recovery from undesired or error states by rolling back to a previous good state, and (3) should not be vulnerable to misoperations, but should prevent them from driving the system into error states.

Acto is state centric. It models an operation as a pair of current system state and a declaration of desired state. A correct operation enables a state transition from the current state to a new state that satisfies the declaration. Within this state-transition model, bugs manifest as (1) undesired transitions in which the new state implicitly violates the declaration, or (2) failure to recover from error states. Vulnerabilities to misoperations cause transitions into explicit error states.

To systematically explore state transitions under different scenarios, Acto generates state declarations that cover all system properties exposed by the operation interface (e.g., `replicas` and `affinity` in Figures 1 and 2). To make generated state declarations semantically meaningful, Acto automatically infers the semantics of properties and predicates. Acto ensures that generated declarations are syntactically valid and that they can exercise operators in diverse ways.

To validate operation correctness under different system states, Acto executes the e2e tests in a test campaign, where the operator is continuously tested under a sequence of operations. Each operation reconciles the system to a new state from which the subsequent operation starts. Acto's test campaigns leverage the level-triggering principle [52]: a correct operator must reconcile the system to the desired state regardless of the start state, and it must recover from error states.

Acto's oracles check for errors that (1) manifest in explicit forms such as unexpected exceptions and panic signals, and (2) only manifest implicitly as mismatches between reconciled system state and declared desired state. To detect implicit mismatches after state reconciliation, Acto checks for consistency in the state views of the operator and the underlying management platform (e.g., Kubernetes); inconsistencies indicate bugs. Acto also employs a differential oracle atop state objects from different state transitions to the same end states, taking advantage of the interpretability and uniformity of those state objects in modern management platforms.

Key results. We implemented Acto for Kubernetes operators. Acto works in two modes: a blackbox mode (Acto-■) that only requires the operator's interface specification (i.e., custom resource definition of Kubernetes operators) and a whitebox mode (Acto-□) that additionally takes the operator's source code for semantics inference and predicate analysis.

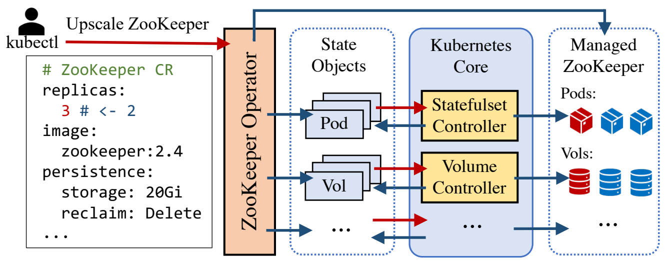
We evaluated Acto on eleven popular Kubernetes operators of various kinds. Acto found 56 new operator bugs in total, among which 40 have been confirmed and 28 have been fixed. Acto also found six bugs in Kubernetes and in the Go runtime that affected multiple operators (all have been confirmed or fixed). The detected bugs lead to severe safety and liveness issues, affecting not only the operators, but also the reliability and security of the managed systems. Lastly, Acto finds a large number of vulnerabilities to misoperations. Acto tests all these operators within eight hours (a nightly run) on a cluster of eight machines; five of eleven operators only need one machine. Acto yields a few false positives: Acto-□ reports no false alarms and Acto-■ has a 0.19% false alarm rate.

Contributions. This paper makes these contributions:

- We present the first automatic end-to-end testing technique for cloud-native operators using a state-centric approach.
- We design and implement Acto, a practical tool that uses the proposed technique to automatically test unmodified Kubernetes operators and can detect many kinds of bugs.
- Acto has already helped improve the quality of eleven popular Kubernetes operators by finding bugs that were fixed by developers. Acto can be run nightly.
- We release Acto as an open-source project, with instructions on how to reproduce our results.

2 Background

Operation programs (i.e., operators) for modern cloud management platforms like Kubernetes [37], Twine [76], and ECS [62] follow a declarative, state-reconciliation design



```
# ZooKeeper CR
replicas:
  3 # <- 2
image:
  zookeeper:2.4
persistence:
  storage: 20Gi
  reclaim: Delete
...
```

Figure 3: **Scaling up a ZooKeeper system (from 2 to 3 replicas) with a new desired state declaration (CR).**

pattern. An operation declares a desired system state and the operator automatically reconciles the system to the declared state. This design pattern simplifies system management operations by removing the need to write ad hoc, imperative scripts for different one-off tasks. The pattern also makes system management declarative and intent driven. In this paper, we use "*cloud-native operators*" [2] to refer to operators that follow this pattern. We give a brief overview of the pattern, using Kubernetes [37] as a representative example.

Declarative operation interface. In Kubernetes, operators expose a declarative interface in the form of *custom resources (CRs)* [3]. A CR defines a system resource and properties that can be modified to manage that resource. A state declaration specifies property values in a CR. Figure 3 shows an example of desired-state declarations for ZooKeeper; it specifies primitive properties like `replicas` and `image`, and composite properties like `persistence`, which has sub-properties. A ZooKeeper operator *reconciles* a managed ZooKeeper cluster to satisfy the declared state. Users express management operations by changing one or more property values in a CR.

Kubernetes operators maintain CR specifications in the OpenAPISchema format [13], which defines constraints on each CR property (e.g., data type and data range). Operations that change a CR are first validated against the specification by the API servers, before being forwarded to the operator.

Operator design pattern. Kubernetes operators follow the state-reconciliation pattern of modern cloud management platforms and control planes, such as Kubernetes, Borg, Omega, Twine, and ECS [37, 62, 71, 76, 78, 80]. An operator continuously reconciles the managed system from its current state to a newly declared desired state, if the current state does not match the declared state. The management platforms maintain their current system states in a collection of *state objects* in strongly consistent datastores (e.g., etcd [7]). Every entity in the system, such as a pod, a volume, and a stateful set (representing a stateful system), has a corresponding state object. State objects have uniform APIs and consistent data schema, making them interpretable and extensible [37].

Figure 3 shows how a ZooKeeper operator scales up a managed ZooKeeper cluster. A user declares the desired state of the ZooKeeper cluster by submitting a new CR that changes the `replicas` property from `2` to `3` via the Kubernetes client (`kubectl`). The operator processing the desired state declaration first confirms that the current number of replicas in the ZooKeeper cluster is different from `3`—only two pod objects for replicas currently exist in `etcd`. To reconcile to the desired state, the operator notifies Kubernetes to increase the statefulset count for replicas. To do so, Kubernetes creates a new pod and a new volume. State reconciliation stops when the desired state with three replicas is reached.

Operation correctness. We define three correctness requirements for operations: the operator (1) always reconciles the managed system to valid, reachable desired states, *regardless* of its current or previous states; (2) can recover the managed system from implicit or explicit error states by rolling back to a previous good state; and (3) should prevent misoperations from driving the managed system into error states. Figure 1 violates the first requirement and Figure 2 violates the second requirement. In this paper, we treat root causes of violations to the first two requirements *bugs* and report them to developers. We refer to root causes of violations of the third requirement as *misoperation vulnerabilities*, which are known to be serious issues [35, 36, 46, 63–65, 82]. We discuss systematic mitigation for misoperation vulnerabilities with developers.

Operation correctness is hard to achieve. Developers face the twin fundamental challenges of (1) anticipating relevant states to explore in the enormous state space, and (2) correctly reconciling managed systems from different start states.

3 Motivating Study

To understand the kinds of tests that operator developers write and the limitations of current testing practices, we study 50 Kubernetes operator projects from GitHub and their tests.

Finding 1. *Most operators that we study rely on unit tests, which cannot validate operation correctness. Only 34% of these studied operators have a few end-to-end tests.*

Checking if a managed system reaches desired states is beyond the scope of unit tests, each of which checks a method in operator code. Such checks need end-to-end (e2e) tests [10] to validate operation correctness of the managed systems.

Typically, an e2e test first causes an operator to carry out an operation, e.g., to deploy, scale, or reconfigure the managed system. Then, the e2e test checks if the operation succeeded by means of assertions that compare the reconciled managed system state with the expected state. However, only 17 (34%) of 50 operators include e2e tests, and those manually written e2e tests are few, with a median of six e2e tests per operator.

We focus the rest of our study on the effectiveness of existing e2e tests, since we address operation correctness. We study four operators from the 50 and their e2e tests: KnativeOp, PCN/MongoOp, RabbitMQOp, and ZooKeeperOp. These operators are developed either by official teams of the managed systems, or by companies that sell services built

Operator	# Properties		Tests with multiple operations	
	# Tested	Total	% (#)	# Ops (Avg)
KnativeOp	8 (2.15%)	372	14.29% (1/7)	6
PCN/MongoOp	70 (1.27%)	5495	38.71% (12/31)	2.58
RabbitMQOp	19 (1.43%)	1332	25.00% (2/8)	2.5
ZooKeeperOp	13 (1.47%)	886	75.00% (6/8)	2

Table 1: **Properties covered by existing e2e tests and characteristics of tests that trigger multiple operations.**

Operator	# Assertions				# State Objects	
	Env.	State	Behav.	Total	Asserted	Total
KnativeOp	18	32	0	50	14 (0.93%)	1506
PCN/MongoOp	2	209	177	388	329 (10.90%)	3017
RabbitMQOp	26	19	29	74	12 (0.42%)	2852
ZooKeeperOp	62	54	0	116	7 (0.24%)	2934

Table 2: **Three types of assertions in existing e2e tests.**

around the managed systems. These four operators contain 7–31 e2e tests; PCN/MongoOp relies only on e2e tests (no unit tests). Table 4 provides more data about these operators.

Finding 2. *Existing e2e tests cover only 1.27–2.15% of supported properties exposed by the operation interface. Also, most tested operations start from the default initial state.*

Table 1 shows that existing e2e tests change very few properties when testing operation correctness in these four operators. We find that some operators' e2e tests do not check basic operations, e.g., backend migration in RabbitMQOp. Also, few e2e tests check operations in multiple configurations, e.g., deploying Zookeeper with persistent *and* ephemeral storage. Acto helps test more operations in multiple configurations.

Operators continuously monitor and reconcile managed systems from any state to the desired states. So, operations should be tested from *different* start states. Consider scaling: given a desired number of replicas, triggering a scale-up or a scale-down procedure depends on the current state. Table 1 (third column) shows that the few e2e tests that check multiple operations only check 2.97 operations on average, a small number compared to how operators work in practice. Most tests trigger only one operation from the *default* initial state.

Finding 3. *State-based assertions in existing e2e tests cover only 0.24–10.90% of managed systems' state-object fields.*

Given the enormous state space, developers likely find it tedious to write assertions on many state-object fields. Table 2 shows a breakdown of three kinds of assertions that we observe in existing e2e tests; they check (1) *the environment* (e.g., can operators request Kubernetes services?); (2) *system states*—is the managed system reconciled to the desired state?; and (3) *managed system behavior*. Assertions on the environment check that operators run in compatible settings; they do not validate operation correctness. State and system-behavior assertions could validate operation correctness. But, in our study, these kinds of assertions only check a small part of the system state or they only check service availability.

Finding 4. *The few assertions on system behavior are basic and mostly check service availability.*

KnativeOp and ZooKeeperOp tests have no assertion on system behavior. In PCN/MongoOp and RabbitMQOp, such assertions only check that the managed system responds to read/write requests from clients. We find a few assertions on system-specific behavior: (1) 36 of 177 assertions in PCN/-MongoOp check backup service availability; and (2) only one of 77 RabbitMQOp assertions checks membership list size.

Implications. Our study shows that current manual testing of operation correctness is significantly limited, even for popular operators with many Github stars (see Table 4, §6). Our results suggest that manually writing e2e tests is tedious and inadequate. So, *automatic* e2e testing of operation correctness is desirable. We believe that such automatic testing is viable and can be done effectively by leveraging the declarative, state-reconciliation pattern of modern cloud-native operators.

4 Technique

Acto is a state-centric testing technique. It tests *operation correctness* by performing end-to-end (e2e) testing of cloud-native operators together with the managed systems. To do so, Acto continuously generates new operations during a test campaign. Then, Acto's oracles check if the operator *always* correctly reconciles the system from each current state to the desired state, or raises an alarm otherwise.

Acto detects bugs when requirements of operation correctness (§2) are violated. Such bugs include those that (1) cause an operator not to reconcile the system to desired states, (2) crash the operator or the system, and (3) prevent the managed system from recovering from an error state. Acto also detects vulnerabilities to misoperations that can drive the systems into explicit error states.

Acto generates minimized e2e test code for every alarm that it raises. These generated tests can help developers to reliably reproduce a bug or a vulnerability, without rerunning the entire test campaign. That is, generated e2e tests only run operations that are necessary to set up the state for reproducing a bug or a vulnerability. Developers can include the generated e2e test in their regression test suite.

Acto is *automatic*—it tests *unmodified* operators and requires no manual annotation, instrumentation, or assertion. The test inputs that Acto automatically generates are *operations*, which drive the operator under test to reconcile the managed system to declared desired states. Acto ensures that generated operations are syntactically valid and represent various scenarios by analyzing the constraints and semantics of properties exposed by an operator's interface. Acto dynamically computes the desired state for triggering the next operation based on the current state.

Acto's test oracles check if the system state after an operation matches the desired state. Automatic test oracle generation is a hard problem in general. Acto's test oracles are enabled by a key opportunity in state-reconciliation based management platforms like Kubernetes: they maintain the system states in uniform, interpretable state objects that can be systematically queried and analyzed.

Usage. Acto works in two modes: a *blackbox* mode (Acto-■) and a *whitebox* mode (Acto-□). Acto-■ takes two inputs: 1) a manifest for building and deploying the target operator, and 2) the specification of state declaration provided by the operator interface, such as the custom resource definition of Kubernetes operators. Both inputs are abundant in mature operator projects; they are widely used for operator development and deployment. In our experience, finding these inputs is straightforward. Acto-□ requires an additional input: the operator's source code for static program analysis. Acto outputs test failures, debugging information about the root cause, and minimized test code that reproduces detected failures.

4.1 Operation Model

Acto models an *operation* as a pair, (S^c, D), where S^c denotes a current system state and D is a declaration of a valid desired state. D is constrained by the operation interface specification (e.g., a CR definition in Kubernetes). If successful, an operation triggers a state transition, $S^c \xrightarrow{D} S^D$, where S^D satisfies D, i.e., $S^D \models D$. D often only specifies a (small) part of the system state. So, there are multiple possible system states that can satisfy D, and, in practice, only a small part of S needs to be examined to check if $S^D \models D$.

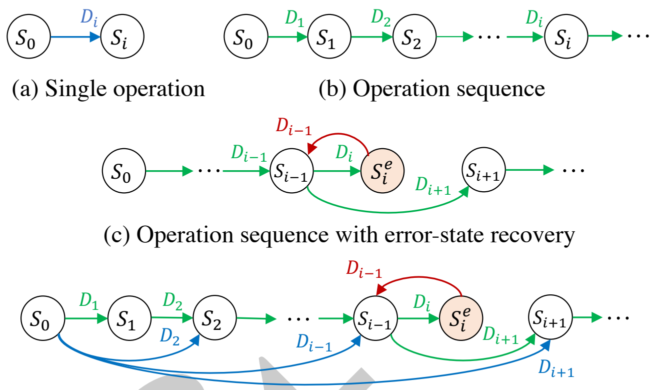
If an operation fails (e.g., due to bugs in operator code), the system enters an error state, $S^e \not\models D$, i.e., S^e does not satisfy the desired state. When $S^e \not\models D$, the operator should be able to rollback the state from S^e with a state transition $S^e \xrightarrow{D_{i-1}} S^c$, where D_{i-1} is the desired state declaration that previously triggered a transition to S^c.

The fundamental challenge in testing operators is the prohibitive cost of testing all elements in the Cartesian product of $S = S^C \cup S^E$ and $\check{D}$, where S^C is the set of all possible valid system states ($c \in C$), S^E is the set of all possible error states ($e \in E$), and $\check{D}$ is the set of all possible declarations of desired state ($D \in \check{D}$). There can be an infinite number of values for different properties that constitute the system state. Exhaustive testing is impossible and any practical testing approach can only exercise a part of the state space, i.e., $S \times \check{D}$.

4.2 Test Strategy

Acto systematically explores the state space using the following three test strategies (Figures 4a–c).

1. Single operation. Acto generates a declaration of a desired state D, triggers an operation to reconcile the current system state S^c to the desired system state S^D, and checks whether $S^D \models D$. The single operation is applied to the initial system state $S^c = S_0$ (starting from a non-initial state requires more operations). This simple single-operation strategy is similar to the current testing practices discussed in §3; it is easy to



(a) Single operation (b) Operation sequence

(c) Operation sequence with error-state recovery

(d) Acto's test exploration strategy

Figure 4: **State transitions of different test strategies.**

implement and reason about. The key challenge is how to explore an effective and representative subset of $\check{D}$.

2. Operation sequence. Acto extends single operations into a test campaign, which consists of a sequence of operations. Test campaigns overcome the limitation of the single-operation strategy, which must always start from the initial state $S^c = S_0$. As discussed in §3, it is important to test whether an operator can reconcile the system to desired states from different, non-initial start states. Reaching an end state from different start states increases the chance of invoking different procedures in the operator code. In a test campaign, earlier operations take the system to new states which become the start states for subsequent operations. Acto generates a test campaign by chaining the expected end states $\{S_i\}$ from the single-operation strategy, and generating a new D_i after each successful reconciliation, as shown in Figure 4b. The result is a sequence of state transitions, $S_0 \xrightarrow{D_1} S_1 \xrightarrow{D_2} \dots \xrightarrow{D_i} S_i \xrightarrow{D_{i+1}} \dots$; Acto checks whether each $S_i \models D_i$, where $i \neq 0$.

3. Error-state recovery. The operation-sequence strategy does not test whether an operator correctly restores a system from implicit or explicit error states. If the system is in an error state S^e, the operator is responsible for recovering from S^e by reconciling the system from S^e back to the prior healthy state S_{i-1}. The subsequent operations start from S_{i-1}, such as in the transition, $S_{i-1} \xrightarrow{D_{i+1}} S_{i+1}$, in Figure 4c. Error states can be reached because of operator bugs that reconcile the system to a state $S^e \not\models D$, or misoperations—semantic errors in D that escape syntactic validation against the interface specification.

Acto combines these three test exploration strategies (Figures 4a–c) to realize the state transition sequences in one test campaign, as shown in Figure 4d.

5 Design and Implementation

This section describes the main components of Acto and how we implement them. These components embody Acto's state-centric testing technique (§4); they generate declarations of desired system states, execute test campaigns, and check reconciled states using automated test oracles.

5.1 Realizing State Transitions

During a test campaign (Figure 4d), Acto automatically generates a new state declaration D_{i+1} based on the current system state S_i to realize a state transition, $S_i \xrightarrow{D_{i+1}} S_{i+1}$. Test campaigns start from the initial state S_0. Acto triggers state transitions with the goals to: (1) cover all properties exposed by the operation interface, and (2) exercise representative operation scenarios based on property semantics.

Acto aims to systematically exercise all properties that are defined in the operation interface. Each new D_{i+1} declares a change to one property in the current state S_i, and any other properties that are needed to satisfy predicates on property relationships (§5.2.4). All state declarations collectively change every property at least once during a test campaign.

Acto tests different scenarios based on the semantics of the changed properties. (Acto automatically infers these semantics, §5.2.2). Table 3 gives a few such scenarios. For example, Acto tests the scale-up-and-scale-down and the scale-down-and-scale-up sequences if a property represents the number of replicas. Acto also tests different pod assignments that trigger the operator to re-configure or re-deploy managed systems differently. We implement scenarios as plugins that can be extended or customized, and users can add more plugins.

In addition to valid operation scenarios, Acto also generates *misoperations*, each of which triggers a state transition to an error state, S^e. For example, Acto generates misoperations that (1) scale the replicas beyond the total number of available physical resources, and (2) set unsatisfiable affinity rules (Table 3). Acto uses misoperations to check if an operator (1) is vulnerable to errors, and (2) can recover from undesired or error states. Acto's oracles (5.3) check the former (is the system in a state S^e?). Acto checks the latter by rolling back S^e to the most recent healthy state. Misoperations that declare semantically erroneous states could escape constraint validation (§5.1). A correct operator should either not carry out an erroneous operation or recover from operation failures.

Property	Scenarios
`Replicas`	Scale up and then down; scale down and then up; upscale over system resource limit.
`Affinity`	Place all pods on one node; spread pods to different nodes; set unsatisfiable affinity rules.
`Storage`	Expand storage volumes; shrink storage volumes; request more storage than is available in a cluster.
`Access`	Switch between normal and privileged roles.

Table 3: **Example scenarios exercised Acto-triggered state transitions.** Scenarios are created based on property semantics inferred by Acto and they can be extended or customized.

5.2 Generating State Declarations

Acto generates desired state declarations, $D \in \check{D}$, that are syntactically valid, resemble real-world scenarios, and satisfy predicates on property relationships. Such desired states improve the efficacy and efficiency of Acto's state space exploration. End-to-end tests are expensive, so a D that does not satisfy these conditions has a low chance to find bugs. We next discuss how Acto generates D to satisfy these conditions.

5.2.1 Extracting Constraints

The operation interface specification defines syntactic validity constraints on state declarations. For example, Kubernetes' OpenAPISchema specification defines constraints on all supported properties. Acto uses these constraints to ensure that all property values in declared desired states are syntactically valid. (Invalid declarations would likely be directly rejected by the API servers before reaching the operator.) For composite properties, Acto uses composite constraints like required properties and also derives constraints from the subproperties. For primitive properties, Acto uses constraints like the type, min/max values (for numeric types), and length (for string type), regular-expression patterns, etc.

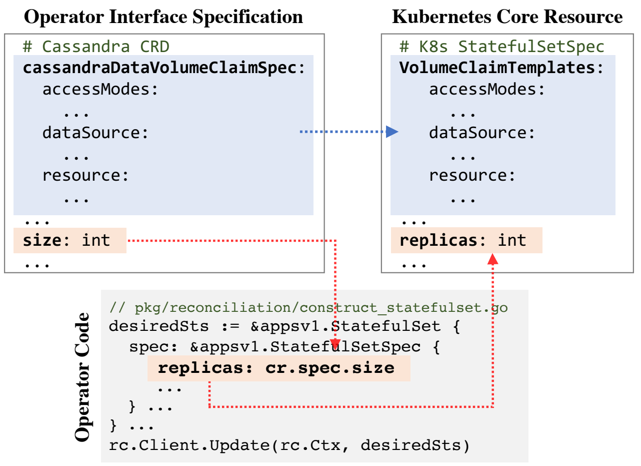
5.2.2 Inferring Property Semantics

To exercise different scenarios (§5.1), Acto changes properties based on their semantics. Acto infers the semantics of a property in the interface specification by mapping it to a set of resource types in the Kubernetes core APIs. Such mapping is feasible because many operations for property changes are eventually delegated to Kubernetes core services.

Inferring semantics from property structure (Acto-■). Acto exploits the insight that property structure is effective for mapping to properties in the Kubernetes core resource specification. Specifically, all Kubernetes core resource types have unique structures. Figure 5 exemplifies how Acto infers semantics from property structure: CassOp has a `cassandraDataVolumeClaimSpec` property with the same structure as the `VolumeClaimTemplates` property in Kubernetes' StatefulSet. So, Acto infers the semantics of `cassandraDataVolumeClaimSpec` using a structural mapping.

Inferring semantics from source code (Acto-□). Acto-■ cannot use property structure to map primitive properties (e.g., `integer`). Also, naming conventions can be ambiguous or unreliable. For example, the integer `size` property in Figure 5 maps to `replicas` in Kubernetes' StatefulSet. To map primitive properties, Acto-□ analyzes operator code. The idea is to track the data flow of the property value in operator code and analyze how the values are used. If a property value is passed to a Kubernetes API or assigned to a Kubernetes resource object, Acto-□ maps the property to a Kubernetes object that stores its value, as shown in Figure 5.

Acto-□ implements a static taint analysis to track property values. The initial taints are pointers and references to the desired-state declaration (e.g., `cr.spec` in Figure 5) and the taints are propagated via data-flow dependencies. The analysis



```
# Cassandra CRD
cassandraDataVolumeClaimSpec:
  accessModes:
    ...
  dataSource:
    ...
  resource:
    ...
...
size: int
...
```

```
# K8s StatefulSetSpec
VolumeClaimTemplates:
   accessModes:
     ...
   dataSource:
     ...
   resource:
     ...
...
replicas: int
...
```

```
// pkg/reconciliation/construct_statefulset.go
desiredSts := &appsv1.StatefulSet {
  spec: &appsv1.StatefulSetSpec {
    replicas: cr.spec.size
    ...
  } ...
} ...
rc.Client.Update(rc.Ctx, desiredSts)
```

Figure 5: **Semantic analysis maps properties in the operation interface to properties in a Kubernetes core resource.**

is field sensitive—to track each primitive (sub-)property in the declaration—, inter-procedural and context sensitive.

5.2.3 Generating Property Values

To generate values for properties with inferred semantics, Acto currently implements 57 property-specific generators based on Kubernetes resource semantics. Most of these properties are composite. The generation focuses on high-level semantics to exercise different scenarios (Table 3). We find that most properties exposed by operation interfaces (83% on average in our evaluated operators) can be mapped to Kubernetes resources. Acto's generators are invoked at runtime. Some generators read environment and runtime information to inform value generation (e.g., an unsatisfiable affinity rule).

For properties whose semantics Acto cannot infer, Acto mutates current values based on their data types while satisfying syntactic constraints (§5.2.1). Acto only mutates primitive sub-properties of composite properties. Acto's mutation ensures syntactic validity but does not guarantee semantic meaningfulness. Mutated values that are not semantically meaningful help check for vulnerabilities to misoperations.

5.2.4 Satisfying Predicates

The values that Acto generates should satisfy predicates, in the form of property dependencies, for changed property values to trigger state transitions. For example, an operation that changes a backup policy only triggers a state transition if backup is also turned on. But, dependencies among properties are often not specified, so Acto automatically infers them.

Inferring dependencies from naming convention (Acto-■). Property names that are exposed by the operation interface provide hints from which dependencies can be inferred. In Kubernetes, dependencies can be identified by feature toggles—each composite property has a Boolean sub-property whose name contains "enabled". For example, operations that change PCN/MongoOp's backup policy must also set `Backup.Enabled` to `True`. Acto-■ infers dependencies on each property that uses this convention based on a breadth-first search that iteratively collects feature toggles. We find this simple heuristic to be effective—it captures 98.05% of control dependencies that we find. Not all dependencies are identifiable from feature toggles, but we only find a small number of other subtle dependencies.

Inferring dependencies using control-flow analysis (Acto-□). Acto-□ analyzes control-flow relationships among program variables in operator code to detect dependencies among property values that do not follow the "*enabled*" naming convention. This analysis is similar to those used for finding dependencies among program inputs [42, 82].

Property p_2 depends on property p_1, i.e., $p_1 \xleftarrow{\text{dep}} p_2$, if p_2 is only used when p_1 satisfies a predicate. Acto-□ searches for control dependencies, $(p_1, \varphi, c) \xleftarrow{\text{dep}} p_2$, where c is some value and φ is a predicate, e.g., an arithmetic, logic, string, or object comparison. Specifically, if a predicate φ *dominates* a sink statement of property p_2 and φ is *not postdominated* by the sink, then there is a control-flow dependency between φ and p_2, i.e., p_2 is used only when φ is `True`. Sinks consume property values, e.g., a `call` to an external API. Further, if φ is determined by comparing the value of p_1 with c, then Acto-□ records a *control* dependency, $(p_1, \varphi, c) \xleftarrow{\text{dep}} p_2$. If p_2 has multiple sinks, Acto-□ reports a control dependency, $(p_1, \varphi, c) \xleftarrow{\text{dep}} p_2$, iff *all* sinks of p_2 depend on (p_1, φ, c).

5.3 Test Oracles

Acto's oracles check whether the state to which the managed system is reconciled matches the specified desired state. If there is a match, Acto reports the operation as successful. Otherwise, Acto signals an alarm that the user can inspect to find bugs or vulnerabilities to misoperations.

The complexity of Acto's oracles depends on whether mismatches between reconciled and desired states manifest explicitly or implicitly. Acto implements oracles to check for state mismatches that manifest as explicit *error states*, such as exceptions, error codes, and timeouts. These oracles 1) scan an operator's error log for unexpected exceptions, e.g., the panic signal in Go; 2) check runtime status of the managed system (recorded in state objects); and 3) check whether an operation returns an error code or fails to complete on time.

Acto's oracles that check for explicit errors are insufficient—many operator bugs manifest as *implicit-state mismatches*, with no explicit symptoms. To find such bugs, Acto also implements oracles to check whether $S_i \models D_i$ for each state transition $S_{i-1} \xrightarrow{D_i} S_i$. Checking $S_i \models D_i$ is challenging: 1) S_i and D_i are represented differently—D_i is a specification [3] and S_i is embodied in state objects [17]—and 2) satisfiability ($\models$) is domain-specific; its semantics may not be obvious. Acto uses two types of oracles to detect implicit-state mismatch:

- *Consistency oracle* (§5.2.2). Acto checks whether $S_i \models D_i$ from the operator and the management platform (e.g.,

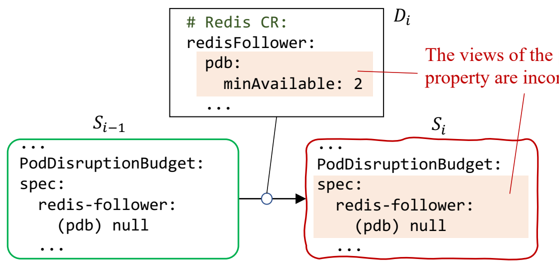


Figure 6: **An OCK/RedisOp bug that is detected by Acto's consistency oracle [24].** The `PodDisruptionBudget` state object has a `null` `pdb`, inconsistent with the `pdb` declared in D_i.

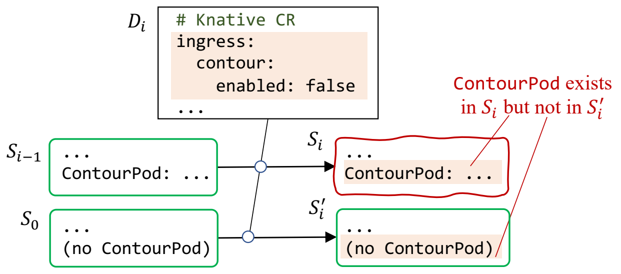


Figure 7: **A KnativeOp bug that is detected by Acto's differential oracle [20].** Contour continues to manage ingress after an operation explicitly disables it.

Kubernetes) views. A buggy operator's view may show $S_i \models D_i$ while the management view shows $S_i \not\models D_i$. Such view inconsistencies likely indicate the presence of bugs.

- *Differential oracle* (§5.3.2). This oracle leverages the level-triggering principle [52] that operators *should* follow: the same desired state should be reached from different start states. So, for each transition pair, $S_{i-1} \xrightarrow{D_i} S_i$ and $S_0 \xrightarrow{D_i} S_i'$, Acto checks whether S_i and S_i' match after state reconciliation based on D_i. This differential oracle also checks whether the operator can recover from an error state, S^e, by checking whether the system state after a rollback matches S_{i-1}, the preceding state before the error.

5.3.1 Consistency Oracle

Some bugs occur if an operator stops reconciliation because the system is in state $S_i \models D$ in the operator's view, but $S_i \not\models D$ in the management platform's view. To detect such bugs, Acto additionally checks whether the management platform's view matches D, based on the platform's description of the reconciled state. In Kubernetes, the platform's view is encoded in `spec` sections of state objects, which are jointly maintained by all running controllers and operators.

For each transition $S_{i-1} \xrightarrow{D_i} S_i$, Acto attempts to match each property p (specified in D_i) to corresponding `spec` fields in the state objects. If a match is found, it indicates that the platform agrees with the operator. Otherwise, Acto raises an alarm.

Figure 6 shows a bug detected by the consistency oracle. OCK/RedisOp should reconcile the system to a declared state with a `pdb` property for Redis followers (to ensure that replicas are available during managed disruptions [16]). But, the property in D_i is not consistent with Kubernetes' view of Redis followers, in which there is no `pdb`. The root cause is that OCK/RedisOp was missing code to support `pdb` for followers, risking the Redis availability during transient disruptions. Such bugs are common due to the operation interface complexity, especially as software evolves [34].

Acto uses property structure analysis (§5.2.2) to infer correspondences between fields in the `spec` section of state objects and a declared property. A declared property could match fields in multiple state objects, but not every matched field is relevant to the property. For example, `PodDisruptionBudget` objects that are not used by Redis followers could also define `pdb`. Acto uses the insight that state object changes occur in small increments, because Acto changes a few properties at a time. So, Acto only matches a specified property to *changed* fields. Acto raises an alarm if a matched field's value is different from the declared property's value, or if a property change does not cause any changes to matched field values.

5.3.2 Differential Oracle

The differential oracle does not check against D_i; it checks that an operator 1) reconciles to the matching desired states from different states S_{i-1} and S_0, and 2) recovers from (implicit or explicit) error state S^e to state S_{i-1}. Acto rolls back to S_{i-1} to continue exploration from a known good state.

Figure 7 shows a bug detected by the differential oracle. There, the Boolean KnativeOp property `contour.enabled` enables or disables Contour, an ingress controller. But, a KnativeOp bug makes it impossible to disable Contour after it is enabled. The consistency oracle does not detect this bug: it is hard to automatically map the Boolean property to the existence of a Contour pod. The differential oracle detects the bug because a Contour pod appears in S_i, but not in S_i'.

Comparison with a second transition that starts from initial state S_0 results from Acto's exploration strategy (Figure 4d). Our choice of S_0 is justified by the fact that S_0 is always a good state and it is used frequently in manually written e2e tests (§3). Conceptually, Acto can compare with a second transition that starts from any good state.

Note that reporting alarms for any difference in the state objects of S_i and S_i' would be brittle and lead to false positives, because execution-specific values like timestamps, IP addresses, and port numbers may change nondeterministically. Acto excludes execution-specific fields when comparing state objects. Acto automatically labels those fields by (1) running the transition $S_0 \xrightarrow{D_1} S_1$ multiple times as a calibration and labeling fields with values varying across runs, and (2) running $S_0 \xrightarrow{D_i} S_i$ multiple times, iff the differential oracle fires an alarm on S_i, to ensure relevant fields are not nondeterministic.

5.4 Reproduction and Debugging

Acto generates minimized e2e test code for every alarm that it raises. When a test fails (the system is in an error state S^e), Acto records failure information (e.g., a dump of the error state, log messages, and system status). Then, Acto rolls back to a valid state S_{i-1} and continues the test campaign.

To generate test code, Acto minimizes the operation sequence that reached S^e to include only two operations, (S_0, D_{i-1}) and (S_{i-1}, D_i). Here, (S_0, D_{i-1}) reconciles the system state to S_{i-1}. Acto outputs the minimized sequence as an executable function that developers can include in their regression test suite after fixing the bug. In our experience, the recorded failure information suffices to effectively locate root causes of test failures. Since the minimized test code reliably reproduces the bug, interactive debuggers [4, 6] can also be used. Acto users can suppress alarms by writing annotations.

5.5 Implementation

We implement Acto for Kubernetes operators. Acto-■ has 12,100 lines of Python code. Acto-□ is built on top of Acto-■ and has an additional 5,700 lines of Go code for program analysis. We currently support operators that are written in Go, the most commonly used language for writing operators. Acto runs tests on virtualized Kubernetes clusters. It supports three backends, Kind [9], Minikube [12], and K3d [8].

Static analysis in Acto-□. We use `ssa` [15] which provides intra-procedural static single-assignment (SSA) representation. We use `pointer` [14] for alias analysis, which implements the Andersen-style point-to analysis [31].

State convergence. Acto applies test oracles only after the system state converges. Convergence time ranges from one second to 10 minutes, so setting a fixed timer would be unreliable. Acto uses a reset timer to check for convergence—it resets the timer when it observes a system event, until no event occurs and the timer times out. We conservatively set the timer to three times the system restart time.

Test parallelization. To speed up testing, Acto partitions generated operation sequences, $[(S_0, D_1), (S_1, D_2), \ldots, (S_x, D_{x+1})]$, into multiple tests and runs them in parallel. To run three partitions of this sequence in parallel, Acto creates three tests corresponding to 1) $S_0 \xrightarrow{D_1} S_1 \xrightarrow{D_2}, \ldots, \xrightarrow{D_i} S_i$, 2) $S_0 \xrightarrow{D_i} S_i \xrightarrow{D_{i+1}}, \ldots, \xrightarrow{D_n} S_n$, and 3) $S_0 \xrightarrow{D_n} S_n \xrightarrow{D_{n+1}}, \ldots, \xrightarrow{D_x} S_x$. If S_i is an error state, it is "rolled back" based on D_{i-1}. Acto can run multiple test partitions on one machine, each in a virtualized Kubernetes cluster with a separate namespace. This approach saves time as test runs wait for convergence. Acto keeps container file systems in memory to reduce the image loading time.

6 Evaluation

Acto's premise is that fully automatic end-to-end correctness testing for unmodified operators is viable and effective. We answer three research questions: 1) Can Acto effectively find new bugs in real-world operators? 2) How efficient is Acto? 3) Are Acto's signaled alarms trustworthy?

We apply Acto to eleven popular open-source Kubernetes operators which manage nine cloud systems (Table 4). All evaluated operators are developed by the official teams of the managed systems, or by companies that sell services built around the managed systems. Test suites in the evaluated operators have similar characteristics as those in §3.

Operator	Systems	Dev.	# Stars	LOC	# E2E Tests
CassOp	Cassandra	K8ssandra	140	23.1K	48
CockroachOp	CockroachDB	Official	224	17.4K	21
KnativeOp	Knative	Official	144	16.3K	7
OCK/RedisOp	Redis	OCK	456	2.5K	0
OFC/MongoOp	MongoDB	Official	878	17.1K	62
PCN/MongoOp	MongoDB	Percona	236	15.0K	31
RabbitMQOp	RabbitMQ	Official	617	14.7K	8
SAH/RedisOp	Redis	Spotahome	950	10.5K	1
TiDBOp	TiDB	Official	1087	132.8K	131
XtraDBOp	XtraDB	Percona	422	15.5K	37
ZooKeeperOp	ZooKeeper	Pravega	316	5.5K	8

Table 4: **The Kubernetes operators that we evaluate.**

Our main evaluation results are summarized as follows:

- Acto finds 56 new bugs in eleven operators; 40 bugs in the operators have been confirmed; 28 have been fixed. Acto also finds six bugs in Kubernetes and in the Go runtime that affect multiple operators; all were confirmed or fixed.
- Acto's test campaigns take less than eight hours per operator on a cluster of eight machines (a nightly run). Five of eleven operators only need one machine.
- Acto generates few false positives: Acto-□ reports no false alarms and Acto-■ has a very low false alarm rate: 0.19%.

6.1 Finding New Bugs and Vulnerabilities

Acto finds previously unknown bugs in all evaluated operators, 56 bugs in total (Table 5). We reported all these bugs. So far, 40 were confirmed and 28 have been fixed. No bug report was rejected. Acto-□ found all 56 bugs. Acto-■ missed one bug, due to not being able to infer the semantics of a primitive property that is needed to generate a scenario.

Acto generates e2e test code to reproduce all 56 bugs that it detects; developers can add these e2e tests to their regression test suite (§5.4). In fact, for six bug fixes, developers added regression tests that perform the same state transition generated by Acto. Our experience shows that the generated e2e test code is invaluable for debugging and validating bug fixes.

Many bugs detected by Acto have severe consequences: managed-system failures, reliability issues, and security issues (Table 6). Note that the evaluated operators are all popular open-source projects (Github "#Stars" column, Table 4), suggesting that operator correctness is hard to achieve.

Acto also finds six bugs in Kubernetes and in the Go runtime that affect multiple operators. These bugs cause wrong or imprecise quantity conversions [25], incompatibility between declarations and API-server validation [18], crashes due to Go's generated shared object [19], etc. All these six bugs were confirmed or fixed after we reported them.

Acto also detects 630 misoperation vulnerabilities (§6.1.2). Each vulnerability corresponds to a unique misoperation that drives an operator into an error state.

Operator	Undesired State	Error State		Recovery Failure	Total
		System	Operator		
CassOp	2	0	0	2	4
CockroachOp	3	0	2	0	5
KnativeOp	1	0	2	0	3
OCK/RedisOp	4	0	3	1	8
OFC/MongoOp	3	1	2	2	8
PCN/MongoOp	4	0	0	1	5
RabbitMQOp	3	0	0	0	3
SAH/RedisOp	2	1	0	1	4
TiDBOp	2	1	0	1	4
XtraDBOp	4	0	1	1	6
ZooKeeperOp	4	1 (0)	0	1	6 (5)
Total	32	4 (3)	10	10	56 (55)

Table 5: **New bugs detected by Acto-□ (Acto-■) in the evaluated operators.** Acto also detected six new bugs in Kubernetes and Go runtime that affect multiple operators.

6.1.1 Bugs Detected by Acto

Acto detects bugs that violate the first two operation correctness requirements: driving managed systems into undesired or error states, or failing to recover from error states.

Undesired state. Acto found 32 bugs, where an operator does not reach the desired state, but neither the operator nor the managed system reports errors explicitly. The consequences of these bugs are neither latent nor easy to observe (e.g., security vulnerabilities). These bugs have different root causes in code, but a common theme is that the operator stops reconciliation before the desired state is reached. We showed two such bugs in Figures 6 and 7. These bugs show the importance of modeling operations as state transitions and testing different state transitions to the same declared states (§4.1).

Error state. Acto found 14 bugs that result in runtime errors or crashes of the managed system or the operator. Among these, four bugs caused runtime errors in the managed systems (such as the one in Figure 1). In another example [29], when testing TiDBOp, Acto generates a valid operation that turns on `binlog` to replicate data using the TiDB binlog. However, TiDB binlog requires a pump cluster to record and sort binlogs, which is not set up by TiDBOp. So, TiDBOp restarts TiDB nodes to load the new configuration, but the replicas crash because of the missing pump cluster.

Acto also found ten bugs that caused operator failure. For example, CockroachOp crashed due to an "index-out-of-range" error when parsing a valid state declaration generated by Acto [26]. The crash brought down the webhook service [5] that the operator uses to validate declarations. After restart, CockroachOp crashed again due to the offending declaration and it got stuck in a crash-then-restart loop.

Recovery failure. Acto detected ten bugs that lead to serious liveness issues (e.g., permanent operator failures) that can neither be addressed by restarting the operator nor by issuing new operations (such as the one in Figure 2). Acto detected these bugs by testing rollback operations with the differential oracle. Our investigation reveals a common coding practice: operators perform new operations only after the system is in a stable state. This practice is a double-edged sword: it prevents bugs caused by racing operations and reduces risks during upgrade, but it makes failure recovery difficult, because it also blocks rollback operations if the system is in an error state.

Consequence	Example	# Bugs
System failure	MongoDB is down and cannot recover [22]	5
Reliability issue	Redis is not protected by disruption budget [24]	15
Security issue	CockroachDB uses outdated secrets [27]	2
Resource issue	Redis runs with no resource guarantee [23]	9
Operation outage	CockroachOp crashes and cannot recover [26]	18
Misconfiguration	Ingress controller cannot be disabled [20]	15

Table 6: **Consequences of the 56 detected bugs in Table 5.** One bug can have multiple consequences.

Test Oracle	# Bugs (Percentage)
Consistency oracle	23 (41.07%)
Differential oracle for normal state transition	25 (44.64%)
Differential oracle for rollback state transition	10 (17.86%)
Regular error check (e.g., exceptions, error codes)	14 (25.00%)

Table 7: **The breakdown of the number of bugs detected by the oracles.** Same bug can be detected by multiple oracles.

6.1.2 Misoperation Vulnerabilities Detected by Acto

Acto-□ detects 630 misoperation vulnerabilities that violate the third operation correctness requirement (Acto-■ detects 616 of these 630). Each vulnerability corresponds to a unique property. Acto detects these vulnerabilities by generating declared states with unsatisfiable affinity rules, misconfigured security contexts, unavailable resources, etc. (Table 3). *All* these vulnerabilities can lead to severe consequences including entire system failure, partial service failures, and reliability issues. In practice, the triggering misoperations could result from human mistakes or wrong policies. These results show that operator developers do not anticipate and defend well against misoperations, which are frequently reported as major causes of system failures [35, 36, 46, 63–65, 82].

We actively discuss with developers on potential mitigation (e.g., by more rigorous early checks). In practice, some of these vulnerabilities are difficult to prevent. The reason lies in the challenges of encoding sufficient domain knowledge in operators to check the semantics of requested operations. For example, it is hard to replicate Kubernetes core scheduler's complex logic [75]. Checking some misoperations requires knowledge of managed systems. State rollback can be an effective mitigation strategy, but it does not always work—over 35% of 630 misoperation vulnerabilities cannot be mitigated by rollbacks due to the recovery-failure bugs in §6.1.1.

6.1.3 Effectiveness of Different Oracles

Acto's consistency and differential oracles catch 43 of the 56 bugs (Table 7). The consistency oracle detects 23 bugs by matching and comparing properties in state declarations to the `spec` sections in state objects (§5.3.1). The differential oracle catches ten more bugs that are triggered during normal state transitions. It also catches all ten recovery-failure bugs during rollback state transitions. The regular error checks detect 14 bugs by checking process status of the operator and runtime status of the managed system (recorded in the state objects).

6.1.4 Coverage

Acto achieves 100% property coverage for every operator—Acto generates at least one operation for each property (§5.1). Acto's effectiveness over manually-written tests (§3) comes from its ability to cover more properties and their values, and more transitions from different states (including error states), with automated oracles. In 38 of 56 detected bugs, the relevant property is not covered by existing tests. Relevant properties for the other 18 bugs are covered, but these bugs elude existing tests because a revealing transition is not exercised. For example, in CassOp, existing tests check that labels [11] are correctly added to pods, but Acto detects a bug [21] that can only be triggered when pod labels are deleted.

6.2 Test Efficiency

Table 8 shows the machine hours that Acto-□'s test campaigns take per operator. All experiments are run on Cloudlab Clemson c6420 machines with 2 Intel Xeon Gold 6142 CPUs (16 cores) and 376 GB of memory, with Ubuntu 20.04 LTS. Campaign times vary from 4.72 to 57.51 hours across operators. Using eight machines, test campaigns for all operators finish in less than eight hours. So, Acto-□ can be run nightly.

Acto's efficiency comes from test parallelization (§5.5). By default, Acto spawns 16 parallel workers to run tests on each machine. But, parallelism can be reduced if the operator or the managed system requires more resources (e.g., memory).

Semantic analysis for composite properties (§5.2.2) drastically reduces the number of operations in test campaigns and allows Acto to focus on high-level semantics of composite properties to exercise representative scenarios, rather than mutating fine-grained primitive (sub-)properties.

Acto-■ takes 8.47% less time on average than Acto-□ because it generates, on average, 48 fewer test operations per operator than Acto-□. The reason is that Acto-■ cannot infer semantics for a few primitive properties and thus cannot generate operations for several scenarios; it only mutates current values within the constraints of a property (§5.2.3).

6.3 False Positives

Acto alarms have a low false positive rate. Acto-□ reports no false alarm. Every test failure during the test campaigns points to either a bug in the operator code or a misoperation vulnerability. In total, Acto-□ reports 2243 test failures: 738 test failures are caused by the 56 bugs in the operator and six bugs in Kubernetes and Go runtime, and 1505 test failures are caused by 630 misoperation vulnerabilities. Fixing one bug or vulnerability may resolve multiple test failures. We are automating alarm clustering based on fault localization [67, 79], but it is now beyond the scope of testing.

Operator	Testing Time (Machine Hours)			# Ops	# Workers
	Generation	Execution	Total		
CassOp	0.02	10.39	10.41	568	16
CockroachOp	0.02	6.08	6.10	371	16
KnativeOp	0.04	6.25	6.29	774	16
OCK/RedisOp	0.02	9.72	9.75	597	16
OFC/MongoOp	0.01	5.73	5.74	434	16
PCN/MongoOp	0.04	26.55	26.58	1749	12
RabbitMQOp	0.03	4.69	4.72	394	16
SAH/RedisOp	0.02	7.92	7.94	718	16
TiDBOp	0.03	16.08	16.11	824	12
XtraDBOp	0.03	57.48	57.51	1950	8
ZooKeeperOp	0.02	8.54	8.55	740	16

Table 8: **Acto-□ test campaign time per operator.**

Acto-■ reports four false alarms in total. It reports 2071 test failures in total; among them, 653 test failures are caused by 55 bugs in operators and six bugs in Kubernetes or Go; 1414 test failures are caused by 616 misoperation vulnerabilities. Therefore, the overall false positive rate of Acto-■ is 0.19%, or 4 out of 2071 alarms. All four false alarms are caused by unsatisfied predicates when Acto-■ changes properties. As discussed in §5.2.4, Acto-■ is unable to infer dependencies that do not follow the naming convention. For example, in ZooKeeperOp, the property, `ephemeral`, depends on a predicate: another property, `storageType`, must also be set to "ephemeral". Hence, Acto-■ fails to satisfy the predicate when changing the `ephemeral` property, but it expects a state change and raises a false alarm. These dependencies are captured by Acto-□ through control-flow analysis (§5.2.4).

6.4 Implications and Discussion

We reflect on our experience on finding root causes of detected bugs and vulnerabilities, and discuss some implications.

Operation coverage. It is nontrivial to validate operators under the declarative model. A key challenge is to reach desired states from many different start states (including error states). We observe that operators invoke different imperative procedures, based on how a declared state differs from the current state. However, it can be tedious and error-prone to cover all such conditions. In fact, most bugs that Acto finds do not manifest when performing operations from the initial state S_0. Operations from the initial state are likely already validated by developers manually or by writing tests. Modeling and testing diverse state transitions are critical to validating operation correctness (§4.1). Declarative programming [75] may make operator testing less error-prone.

Traditional code coverage criteria are insufficient for testing, because they do not consider system states: testing code on one state may not be adequate for a different state. Also, the root causes of some operator bugs are missing code.

Reducing risks. Operations can pose new risks to managed systems—what happens if an operation fails during execution? An operation can span a series of procedures. For example, a reconfiguration operation needs to (1) stop the current running node (with the old configuration); then (2) start a new node (with new configuration). The failure of either procedure is risky. First, such a failure could leave the operator in an intermediate state. As shown in §6.1.1, recovery failures are common. Second, some procedures can open a small window of downtime (e.g., stopping the current node). That downtime grows if a new node fails to start. It is safer to retire the old node *after* the new node starts successfully. But, in practice, this safe start order can be hard to implement, due to the semantic requirements of the managed system and version incompatibility of the changes [59, 84]. For example, a ZooKeeper cluster cannot have two leaders at once, to avoid a split brain. So, a reconfiguration operation must first stop the old leader before starting the new one. System support for speculative execution or emulation can help.

Closing the knowledge gaps. Operations must respect the constraints of the managed system. Otherwise, an operation can harm the managed system. The TiDBOp bug in §6.1.1 is one example. The essential gap lies in the fact that operator developers may not be the managed-system developers, or they may not be aware of subtle, complex constraints. Since operation correctness should be a first-class concern in reliable system design, a rigorous interface between the operator and the managed systems is needed to close these gaps.

7 Limitation and Future Work

Acto is a first step towards thorough validation of operation program (operator) correctness for modern cloud management platforms. Like any testing technique, Acto is incomplete and it can miss bugs. Acto does not cover all possible managed system states and transitions; doing so is prohibitively expensive. Rather, Acto balances cost and coverage: it covers each property at least once and it exercises diverse scenarios based on the semantics of operations. Still, there is plenty of room for future work to improve Acto's state-space exploration.

Acto has other soundness and completeness issues: (1) the predicate analysis of Acto-■ is incomplete, resulting in false alarms; (2) Acto-□'s control-flow analysis only captures predicates that manifest as control-flow dependencies (we did not observe any other kind); and (3) Acto's automated oracles do not incorporate domain knowledge about managed systems and they rely only on state objects that are managed by the platform. Hence, those oracles may not capture complex failure symptoms like gray and partial failures [32, 49, 53, 58].

Acto currently does not target fault-tolerance bugs. Recent tools like Sieve [74] inject faults (e.g., node failures and network delays) and check operator safety and liveness. We plan to integrate tools like Sieve with Acto: (1) Sieve's inputs are e2e tests; Acto generates systematic e2e tests to make fault injection more comprehensive, and (2) fault injectors can generate diverse error states for Acto to test operator recovery.

Lastly, Acto currently focuses on testing individual operators. But, a system may be managed by multiple operators in practice. So, operation correctness could be violated by conflicting operations from different operators. We plan to extend Acto to test interdependent operators together. A key challenge will be to address a larger state space and to reason about state transitions in interleaving operation schedules.

8 Related Work

Prior work identified operation errors as major causes of production failures [33, 35, 36, 46, 48, 63–66, 81]; they result mostly from human mistakes. As human-based operations are increasingly being replaced by automated operation programs, the correctness of those programs is critical. Acto is a first step towards addressing automatic testing of operation programs.

We believe that Acto's ideas can apply beyond Kubernetes to other cloud platforms like Borg [80], Twine [76], and ECS [62]. These platforms also adopt declarative, state-reconciliation design patterns for operators or controllers, as a result of many design iterations [37] and discussions [44, 69].

DCM [75] uses declarative programming to synthesize cluster managers based on constraint solving; the idea can potentially be extended for custom operators. However, most operators are currently written in imperative code.

Acto is complementary to prior work on software deployment [43, 57, 68, 84] and configuration [59, 73]. Acto checks programs that perform those operations rather than the correctness of code or configuration changes.

Sieve [74] tests the fault tolerance of cluster management controllers using fault injection. Operators are custom controllers for managing systems running on the clusters. Acto is fundamentally different from and complementary to Sieve and other fault injectors (as discussed in §7).

Acto can potentially be further enhanced with ideas from fuzzing [61,83] and symbolic execution [38,39]. But, naïve application of these techniques is unlikely to yield benefits. For example, without reasoning about state transition, techniques only guided by code coverage will be insufficient (§6.4).

9 Concluding Remarks

With the rapidly growing practice of automating operations and deploying operators in production, operator correctness has become a critical component of cloud system reliability. This paper presents Acto, an automatic technique for testing cloud-native operators end to end with the managed systems. We show that Acto's state-centric approach enables effective and practical end-to-end testing that is readily applicable to existing operators and complements the significant inadequacy of manually written tests. Our goal now is to make Acto a common utility in developing and testing operators, towards correct automation of cloud operations.

References

[1] Assigning Pods to Nodes. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/.

[2] Cloud Native Computing Foundation Operator White Paper. https://www.cncf.io/wp-content/uploads/2021/07/CNCF_Operator_WhitePaper.pdf.

[3] Custom Resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[4] Debugging Go Code with GDB. https://go.dev/doc/gdb.

[5] Dynamic Admission Control. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/.

[6] Ephemeral Containers. https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/,.

[7] etcd. https://etcd.io/.

[8] K3d. https://github.com/k3d-io/k3d.

[9] Kind. https://kind.sigs.k8s.io/.

[10] Kubernetes End-to-end Testing for Everyone. https://kubernetes.io/blog/2019/03/22/kubernetes-end-to-end-testing-for-everyone/.

[11] Labels and Selectors. https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/.

[12] Minikube. https://minikube.sigs.k8s.io/.

[13] OpenAPI Specification. https://swagger.io/specification/#schema-object.

[14] Package pointer. https://pkg.go.dev/golang.org/x/tools/go/pointer.

[15] Package ssa. https://pkg.go.dev/golang.org/x/tools/go/ssa.

[16] Specifying a Disruption Budget for your Application. https://kubernetes.io/docs/tasks/run-application/configure-pdb/.

[17] Understanding Kubernetes Objects. https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/.

[18] Automatically generated regex validation for Quantity does not match the validation used by unmarshalerDecoder. https://github.com/kubernetes-sigs/controller-tools/issues/665, 2022.

[19] cmd/cgo: allow cgo to pass strings or []bytes bigger than 1«30. https://go-review.googlesource.com/c/go/+/418557, 2022.

[20] Contour pod is not deleted when disabled by user. https://github.com/knative/operator/pull/1176, 2022.

[21] K8SSAND-1503 Unable to remove the additional labels on the seed service through CR. https://github.com/k8ssandra/cass-operator/pull/344, 2022.

[22] Mongodb system is down and unable to recover when the featureCompatibilityVersion is not specified and changed to an invalid value. https://github.com/mongodb/mongodb-kubernetes-operator/pull/1118, 2022.

[23] Redis does not run with resource request/limit set by cr.spec.resources. https://github.com/OT-CONTAINER-KIT/redis-operator/issues/290, 2022.

[24] Specifying the field redisFollower.pdb does not have any effect. https://github.com/OT-CONTAINER-KIT/redis-operator/pull/301, 2022.

[25] The number conversion of Value() of type Quantity is incorrect. https://github.com/kubernetes/kubernetes/issues/110653, 2022.

[26] The operator crashes if the image name does not contain colon. https://github.com/cockroachdb/cockroach-operator/pull/922, 2022.

[27] Updating the field spec.ingress.sql.tls.secretName is not reflected in the sql ingress object. https://github.com/cockroachdb/cockroach-operator/issues/920, 2022.

[28] Zookeeper pod keeps crashing when scaling down and up. https://github.com/pravega/zookeeper-operator/pull/526, 2022.

[29] tidb crash loop when enabling binlog. https://github.com/pingcap/tidb-operator/issues/4945, 2023.

[30] tidb-operator unable to recover an unhealthy cluster even with manual revert. https://github.com/pingcap/tidb-operator/issues/4946, 2023.

[31] ANDERSEN, L. O. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[32] ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. Fail-Stutter Fault Tolerance. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (May 2001).

[33] BARROSO, L. A., HÖLZLE, U., AND RANGANATHAN, P. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*, 3 ed. Morgan and Claypool Publishers, 2018.

[34] BEHRANG, F., COHEN, M. B., AND ORSO, A. Users Beware: Preference Inconsistencies Ahead. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)* (Aug. 2015).

[35] BIANCHINI, R., MARTIN, R. P., NAGARAJA, K., NGUYEN, T. D., AND OLIVEIRA, F. Human-Aware Computer System Design. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)* (June 2005).

[36] BROWN, A. B., AND PATTERSON, D. A. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the 2003 USENIX Annual Technical Conference (ATC'03)* (June 2003).

[37] BURNS, B., GRANT, B., OPPENHEIMER, D., BREWER, E., AND WILKES, J. Borg, Omega, and Kubernetes. *Communications of the ACM 59*, 5 (May 2016), 50–57.

[38] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)* (Dec. 2008).

[39] CADAR, C., AND SEN, K. Symbolic Execution For Software Testing: Three Decades Later. *Communications of the ACM 56*, 2 (Feb. 2013), 82–90.

[40] CEBULA, M., AND SHERROD, B. 10 Weird Ways to Blow Up Your Kubernetes. In *KubeCon North America* (Nov. 2019).

[41] CHEKRYGIN, I. Keep the Space Shuttle Flying: Writing Robust Operators. In *KubeCon Europe* (May 2019).

[42] CHEN, Q., WANG, T., LEGUNSEN, O., LI, S., AND XU, T. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)* (Nov. 2020).

[43] CRAMERI, O., KNEŽEVIĆ, N., KOSTIĆ, D., BIANCHINI, R., AND ZWAENEPOEL:, W. Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System. In *Proceedings of the 21st Symposium on Operating System Principles (SOSP'07)* (Oct. 2007).

[44] DETREVILLE, J. Making System Configuration More Declarative. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)* (June 2005).

[45] DOBIES, J., AND WOOD, J. *Kubernetes Operators: Automating the Container Orchestration Platform*. O'Reilly Media, Inc., 2020.

[46] GRAY, J. Why Do Computers Stop and What Can Be Done About It? *Tandem Technical Report 85.7* (June 1985).

[47] GUILLOUX, S. Writing a Kubernetes Operator: the Hard Parts. In *KubeCon North America* (Nov. 2019).

[48] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SOCC'16)* (Oct. 2016).

[49] GUNAWI, H. S., SUMINTO, R. O., SEARS, R., GOLLIHER, C., SUNDARARAMAN, S., LIN, X., EMAMI, T., SHENG, W., BIDOKHTI, N., MCCAFFREY, C., SRINIVASAN, D., PANDA, B., BAPTIST, A., GRIDER, G., FIELDS, P. M., HARMS, K., ROSS, R. B., JACOBSON, A., RICCI, R., WEBB, K., ALVARO, P., RUNESHA, H. B., HAO, M., AND LI, H. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)* (Feb. 2018).

[50] HAASE, S. How an Operator Becomes the Hero of the Edge. In *OperatorCon* (May 2019).

[51] HALL, C. AWS, Google, Microsoft, Red Hat's New Registry to Act as Clearing House for Kubernetes Operators. https://www.datacenterknowledge.com/open-source/aws-google-microsoft-red-hats-new-registry-act-clearing-house-kubernetes-operators, Mar. 2019.

[52] HOCKIN, T. Kubernetes: Edge vs. Level Triggered Logic. https://speakerdeck.com/thockin/edge-vs-level-triggered-logic, June 2017.

[53] HUANG, P., GUO, C., ZHOU, L., LORCH, J. R., DANG, Y., CHINTALAPATI, M., AND YAO, R. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS-XVI)* (May 2017).

[54] KUMAR, H., AND ŠAFRÁNEK, J. Storage on Kubernetes - Learning From Failures. In *KubeCon North America* (Nov. 2019).

[55] LAGRESLE, M. Moving to Kubernetes: the Bad and the Ugly. In *ContainerDays* (June 2019).

[56] LANDER, R. Kubernetes Operators: Should You Use Them? https://tanzu.vmware.com/developer/blog/kubernetes-operators-should-you-use-them/, July 2021.

[57] LI, Z., CHENG, Q., HSIEH, K., DANG, Y., HUANG, P., SINGH, P., YANG, X., LIN, Q., WU, Y., LEVY, S., AND CHINTALAPATI, M. Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)* (Feb. 2020).

[58] LOU, C., HUANG, P., AND SMITH, S. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)* (Feb. 2020).

[59] MA, S., ZHOU, F., BOND, M. D., AND WANG, Y. Finding Heterogeneous-Unsafe Configuration Parameters in Cloud Systems. In *Proceedings of the 16th ACM European Conference on Computer Systems (EuroSys'21)* (Apr. 2021).

[60] MADHU, C. Preventing Controller Sprawl From Taking Down Your Cluster. In *KubeCon North America* (Oct. 2022).

[61] MANES, V. J., HAN, H., HAN, C., CHA, S. K., EGELE, M., SCHWARTZ, E. J., AND WOO, M. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering 47*, 11 (Nov. 2021), 2312–2331.

[62] MELISSARIS, T., NABAR, K., RADUT, R., REHMTULLA, S., SHI, A., CHANDRASHEKAR, S., AND PAPAPANAGIOTOU, I. Elastic Cloud Services: Scaling Snowflake's Control Plane. In *Proceedings of the 13th ACM Symposium on Cloud Computing (SOCC'22)* (Nov. 2022).

[63] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (Dec. 2004).

[64] OLIVEIRA, F., TJANG, A., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Barricade: Defending Systems Against Operator Mistakes. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)* (Apr. 2010).

[65] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why Do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)* (Mar. 2003).

[66] PATTERSON, D., BROWN, A., BROADWELL, P., CANDEA, G., CHEN, M., CUTLER, J., ENRIQUEZ, P., FOX, A., KICIMAN, E., MERZBACHER, M., OPPENHEIMER, D., SASTRY, N., TETZLAFF, W., TRAUPMAN, J., AND TREUHAFT, N. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Tech. Rep. UCB//CSD-02-1175, University of California Berkeley, Mar. 2002.

[67] PHAM, V.-T., KHURANA, S., ROY, S., AND ROYCHOUDHURY, A. Bucketing Failing Tests via Symbolic Analysis. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE'17)* (Apr. 2017).

[68] PINA, L., ANDRONIDIS, A., HICKS, M., AND CADAR, C. MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)* (Apr. 2019).

[69] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Escape Capsule: Explicit State is Robust and Scalable. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS-XIV)* (May 2013).

[70] RATIS, P. Lessons Learned using the Operator Pattern to build a Kubernetes Platform. In *USENIX SREcon* (Oct. 2021).

[71] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)* (Apr. 2013).

[72] SOSA, C., AND BHATIA, P. Application management made easier with Kubernetes Operators on GCP Marketplace. https://cloud.google.com/blog/products/containers-kubernetes/application-management-made-easier-with-kubernete-operators-on-gcp-marketplace, May 2019.

[73] SUN, X., CHENG, R., CHEN, J., ANG, E., LEGUNSEN, O., AND XU, T. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[74] SUN, X., LUO, W., GU, J. T., GANESAN, A., ALAGAPPAN, R., GASCH, M., SURESH, L., AND XU, T. Automatic Reliability Testing for Cluster Management Controllers. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[75] SURESH, L., AO LOFF, J., KALIM, F., JYOTHI, S. A., NARODYTSKA, N., RYZHYK, L., GAMAGE, S., OKI, B., JAIN, P., AND GASCH, M. Building Scalable and Flexible Cluster Managers Using Declarative Programming. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[76] TANG, C., YU, K., VEERARAGHAVAN, K., KALDOR, J., MICHELSON, S., KOOBURAT, T., ANBUDURAI, A., CLARK, M., GOGIA, K., CHENG, L., CHRISTENSEN, B., GARTRELL, A., KHUTORNENKO, M., KULKARNI, S., PAWLOWSKI, M., PELKONEN, T., RODRIGUES, A., TIBREWAL, R., VENKATESAN, V., AND ZHANG, P. Twine: A Unified Cluster Management System for Shared Infrastructure. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[77] TEMPLETON, G., AND DAVIDSON, S. How a Couple of Characters (and GitOps) Brought Down Our Site. In *KubeCon Europe* (May 2022).

[78] TIRMAZI, M., BARKER, A., DENG, N., HAQUE, M. E., QIN, Z. G., HAND, S., HARCHOL-BALTER, M., AND WILKES, J. Borg: The Next Generation. In *Proceedings of the 15th ACM European Conference on Computer Systems (EuroSys'20)* (Apr. 2020).

[79] VAN TONDER, R., KOTHEIMER, J., AND GOUES, C. L. Semantic Crash Bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)* (Sept. 2018).

[80] VERMA, A., PEDROSA, L., KORUPOLU, M., OPPENHEIMER, D., TUNE, E., AND WILKES, J. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)* (Apr. 2015).

[81] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (Dec. 2004).

[82] XU, T., ZHANG, J., HUANG, P., ZHENG, J., SHENG, T., YUAN, D., ZHOU, Y., AND PASUPATHY, S. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)* (Nov. 2013).

[83] ZELLER, A., GOPINATH, R., BÖHME, M., FRASER, G., AND HOLLER, C. *The Fuzzing Book: Tools and Techniques for Generating Software Tests*. https://www.fuzzingbook.org/.

[84] ZHANG, Y., YANG, J., JIN, Z., SETHI, U., RODRIGUES, K., LU, S., AND YUAN, D. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)* (Oct. 2021).