Reliable Cloud System Management: Challenges and Tools

by

Jiawei Tyler Gu

Dissertation

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2026

Urbana, Illinois

Doctoral Committee:

Associate Professor Tianyin Xu, Chair and Director of Research
Assistant Professor Ramnatthan Alagappan
Professor Indranil Gupta
Professor Darko Marinov
Dr. Chen Wang, IBM

Abstract

Modern cloud systems are increasingly managed by operator programs that continuously reconcile managed systems to desired states declared by users. While operators largely eliminate inadvertent human mistakes, their own bugs can directly and continuously damage bug-free systems in production, leading to data loss, service outages, and security vulnerabilities. As operators become the paradigm for cloud system management, ensuring their reliability is a pressing concern.

This dissertation enhances cloud system management reliability by developing a deep understanding of operator reliability challenges and building practical testing tools that automatically detect serious bugs in operators. It makes following three contributions.

First, we conduct a systematic study of 412 real-world operator failures across 13 popular Kubernetes operators. Our analysis reveals that the fundamental challenge of operator correctness stems from the multifaceted complexity of operators' interactions across four dimensions: managed systems (42%), the cloud platform (30%), the user interface (23%), and co-located operators (5%). The dominant category, erroneous interactions with managed systems, reflects the absence of well-defined management interfaces: critical operations have complex semantics that are poorly documented and never formally specified.

Second, we present Acto, the first fully automatic end-to-end testing tool for cloud system operators. Acto takes a state-centric approach, systematically generating operations that drive diverse state transitions and detect bugs with two principled automated oracles. Applied to eleven popular Kubernetes operators, Acto found 56 previously unknown bugs (42 confirmed and 30 fixed) along with six bugs in Kubernetes itself and the Go runtime.

Third, we present Oat, a testing tool that builds on Acto's framework to target operator-system interaction bugs, which is the largest yet unaddressed operator failure pattern. Oat generates semantically meaningful values for system-specific properties, injects faults against managed systems, and provides enhanced oracles that check system-internal state. Applied to six mature Kubernetes operators, Oat detected 86 new bugs with no false alarms, finding bugs in every tested operator.

Together, these contributions demonstrate that a deep empirical understanding of operator failures enables the design of practical testing tools that uncover serious and new bugs in cloud system operators.

To whom I love,
to who love me,
and to myself.

Acknowledgements

I would like to take this moment to thank the people who have helped me through my PhD journey. Looking back, I realize that the skills I take away from this journey are inseparable from the people I have shared it with. *Waking up today, I be feeling loved.*

First and foremost, I would like to thank my PhD advisor, Tianyin Xu. I am deeply moved by Tianyin's genuine care for and investment in his students. I am not the kind of student who gets things right on the first try, and yet Tianyin taught me relentlessly, again and again, with a patience I will always be grateful for. I still remember the early days of my first project, Acto, when Tianyin would spend hours with me explaining how to run a research meeting and how to make meaningful progress on hard problems. Throughout my PhD, he consistently encouraged us to take on every opportunity that would sharpen the skills a researcher needs, including writing, presenting, and collaborating, even when doing so demanded more of his own time. Beyond research, Tianyin has become a role model with qualities that I want to embody: his relentless pursuit of excellence, his willingness to be critical of himself, his deep reflection on the goals that he wants to achieve, and his deep care for the people around him. His words surface in my head from time to time when I face challenges. Having Tianyin as my advisor alone made my entire PhD journey worth it.

I am deeply grateful to Chen Wang, who has been a wonderful collaborator and mentor throughout my PhD. Chen offered tremendous support in securing resources for my research and in pushing Acto towards greater impact. She continually fought for opportunities on our behave, which is how we were able to present Acto at a major industry venue, KubeCon. I am also grateful to Owolabi Legunsen. Although I was never formally his student, Owolabi devoted an enormous amount of effort to my project. His rigor and attention to detail not only strengthened the project itself but also shaped me into a better researcher.

I would like to thank the rest of my PhD committee: Ramnatthan Alagappan, Darko Marinov and Indranil Gupta. I had the privilege of collaborating with Ram on the Sieve paper early in my PhD, and I learned a lot from taking his classes and from our many discussions since. Aishwarya and Ram shared their pizza with me late night during the SOSP deadline run, a kindness that I have not forgotten. I have had many discussions with Darko throughout my PhD, and I have enjoyed every one of them. Darko always asks sharp, thought-provoking questions, and I now find myself preparing for presentations by asking what Darko would ask. Indy has had a significant influence on my PhD journey. I took his Distributed Systems course and was impressed by his enthusiasm for teaching. I am also grateful for the suggestions Indy offered during my preliminary exam and defense,

which greatly improved this dissertation.

I have been fortunate to receive mentorship from others beyond my committee, and I would like to sincerely thank Suman Nath and Brian Detering. Suman mentored me during my first internship at Microsoft Research, where he was tremendously supportive throughout. I deeply appreciate the opportunity he gave me to understand reliability issues in real production environments. Brian mentored me during my internship at Meta, guiding me through my summer project with remarkable investment. Brian set me up for success: he explained in great detail how to execute well on the project, how to increase visibility, how to talk to customers to increase the impact of my project, and he fought for me in evaluation meetings. His advice on succeeding in industry will stay with me throughout my professional career.

My PhD would be incomplete and impossible without the support of my friends. I would like to thank all the members of xLab, and in particular those I worked closely with during my PhD: Xudong Sun, Zhen Tang, Xinze (William) Zheng, Yuxuan (Matt) Jiang, Kunle Li, Muhammad Taha, Kai-Hsun (Kevin) Chen, Yiming Su, Wenqing Luo, and Bogdan Alexandru Stoica. Xudong has been a valuable mentor to me. He has always been patient with my random ideas, and when I came to him anxious about something, he would chat with me, offering reassurance and then thoughtful advice. I still remember my first day in Champaign, when Xudong took me to lunch, and the afternoon sitting beside him in the lab, learning how to build a Docker image. In countless ways, Xudong set the example for me: how to engineer carefully, how to run a meeting, how to drive a project, how to write a paper. Working with Zhen was another deeply fulfilling experience. Zhen and I spent countless nights discussing and debating failure patterns for our study, and I would not have made it through without him; his perseverance and steady growth continue to impress me. I am grateful to Bogdan for his kind heart and for the tremendous help he has given me on my writing. I always had a lot of fun hacking things together with Wenqing, from our course project to playing with Vim configurations late into the evening. Kevin's endless curiosity and desire to learn continue to inspire me; our conversations have given me invaluable lessons, including the idea that building an identity others can recognize is central to career development. I was also fortunate to have Yuxuan, Kunle, William, Taha, and Yiming on the team. They proactively proposed new solutions to the problems we faced and greatly improved our results.

I would also like to thank the friends who made my PhD journey a lovely experience. Siyuan Chai, Yinfang Chen, Shuai Wang, Xuhao Luo, and I entered the program in the same year, and we have shared countless meals and events together. I always had a lot of fun chatting with Jinghao Jia, Jiyuan Zhang, and Wentao Zhang about the bizarre

behaviors of programming languages and systems. I was very lucky to find in Hao Lin and Zhen Tang two table-tennis partners at just my level——it made for many joyful matches. I am grateful to Jinghan Sun for his company during the summer of 2023; we walked half an hour back from the bus station every day, and Jinghan patiently listened to me the whole way. It was a difficult time for me, and his company made it far less so. Jinghan and I also dined out and drank boba tea every day, and we became so out of shape at the end of the summer. Finally, I would like to thank Shizhuo (Dylan) Zhang, Lan Li, Jiang Lan, Yongzhou Chen, Zikun Liu, Qinhan Xia, Daixuan Li, Gangmuk Lim, Eashan Gupta, Henry Zhu, Xiaojuan Ma, Xinying Zheng, Lilia Tang, Ahan Gupta, Kaizhuo Yan, Chirag Shetty, Bill Tao, and Dazhen Chen for all the quality time we have shared.

A special thank-you goes to my partner, Xinyi Wei, and to our puppy, DwenDwen. Thank you for filling my life with joy, and for making me look forward with hope to the next chapter after this PhD.

I want to thank my parents and my family. Without their support, I would not have had the freedom to pursue my dreams. I have always wanted to go out into the world and discover who I am, but looking back, my family has always been the peaceful harbor I could return to.

Finally, I want to thank the music, films, and shows that have kept me going through these years. The coherent narratives carried by great albums—including *Blonde* by Frank Ocean (thanks to Qinhan for the recommendation), *Prince Charming* by Masiwei, *Waking Up After the Rain* by Asen, *Chromakopia* by Tyler the Creator, and *Bully* and *My Beautiful Dark Twisted Fantasy* by Ye—have long inspired me to craft an "album" of my own, and I am glad to now present my piece: this dissertation.

Contents

Chapter 1 Introduction

Modern software systems undergo frequent and complex management operations [143, 55] such as reconfiguration, rolling out new versions, and autoscaling. As systems grow in scale and complexity beyond what human-based operations can reliably and efficiently manage, they are increasingly being managed by operation programs. On modern cloud platforms like Kubernetes, these management programs are commonly referred to as "operators" [30], to draw analogies with human operators [43].

Operators are long-running, deterministic management programs that continuously manage systems on top of the cloud platform. An operator follows the state-reconciliation pattern: it observes the current state of the managed system, compares it against a desired state specified through a declarative interface, and issues corrective actions to close any difference. Different from traditional infrastructure-as-code (IaC) scripts [122, 44], operators are long-running production services on the cloud that *continuously* manage *production systems* through their lifecycle, including deployment, reconfiguration, failover, recovery, and so on, and must reason about the full space of state transitions rather than a single deployment path.

While operator programs largely eliminate inadvertent human mistakes [99, 112, 13, 108, 103], their own correctness has unprecedented impacts. A runaway operator can directly and continuously damage bug-free systems in production. Recent studies show that software bugs in operators can lead to disastrous consequences like data loss, service unavailability, and security issues [57, 137, 161, 84], along with other significant production incidents [80, 58, 25, 78, 146, 22].

As operators become the new paradigm for cloud system management, their reliability has become an emerging and pressing problem. The goal of this dissertation is to enhance the cloud system management reliability by (1) understanding the fundamental reliability challenges in cloud system operators and (2) building practical and effective tooling support for detecting reliability issues in operators.

1.1 The Problems

1.1.1 What are the reliability challenges of operators?

An operator, by design, involves complex, multi-dimensional interactions which introduce unique reliability challenges beyond generic software bugs. Unlike typical software that manages its own internal state, an operator must interact with the managed system through system-specific APIs that differ even among similar systems. Beyond the managed systems, operators rely on the cloud platform to provision and manage system resources

(pods, data volumes, network policies, etc.) that are allocated to the managed systems. Furthermore, operators do not operate in isolation. In practical cloud infrastructures, multiple operators coexist on the same cluster, potentially interfering with each other. An operator must also interact with user inputs through the declarative interface. The interactions among these dimensions—the managed system, the cloud platform, co-located operators, and the user interface—create a multifold complexity that is specific to operators and not well captured by existing failure studies.

Recent work has developed testing and verification techniques [140, 137, 57, 139, 7, 84] for operator-like programs. However, it is unclear whether and how much these efforts have addressed *real-world* operator reliability, as they target specific, predefined bug patterns, so it is hard to tell if they cover major types of production operator failures. Without a comprehensive, empirical understanding of how these dimensions manifest as real-world failures, the research community lacks a principled foundation for building tools and techniques to improve operator reliability. A systematic study of real-world operator failures is therefore a prerequisite for making informed decisions about where to invest effort in improving operator reliability.

To shed light on operator reliability research, this dissertation seeks to develop a deep understanding of the reliability challenges of operators.

1.1.2 How to automatically test operators end-to-end?

In practice, operator developers mainly rely on unit tests and integration tests for quality assurance. Unit tests verify individual functions in isolation, while integration tests typically run the operator against a mocked or simulated cluster environment. While these testing strategies are valuable for catching basic logical errors, they cannot check operation correctness end to end, 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 system 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 [91] or symbolic execution [17] cannot effectively test operators end to end, since they neither reason about the semantics of operations nor check the system states. In particular, operator bugs do not commonly manifest as crashes, but drive systems into undesired states (§3.6.1).

The fundamental challenge of testing operators automatically is to explore their large state space. An operation drives the system from an existing state to a new desired state. The state space of operators is the Cartesian product of the existing state and the desired

state. Exhaustively exploring every single element is prohibitively expensive.

This dissertation explores how to build an efficient and practical automatic testing tool that can be widely applied to any type of operators.

1.1.3 How to test operator-system interactions?

Even with an end-to-end testing framework for operators, a critical gap remains in testing the most challenging part of operator behavior: its interactions with the managed system. As we will show in our empirical study (Chapter 2), erroneous interactions with the managed system constitute the largest pattern of operator failures in the field, contributing to 42% of the bugs we analyzed.

Existing testing approaches, including end-to-end testing tools like Acto, are system-agnostic, thus cannot generate operations which can effectively exercise the interaction with systems. The importance of operator reliability demands *automatic* testing techniques for detecting defects in operator-system interactions and preventing interaction failures. Based on our findings from Chapter 2, such testing must advance existing techniques in two aspects:

First, the tool must systematically exercise the operator-system interactions. Specifically, it must generate operation commands that can mutate system-specific properties through the declarative user interface. Existing tools for operator and controller testing [137, 57] are *system agnostic*—they only mutate system resources properties, but skip system-specific properties.

Second, the new tool must inject faults against systems. No existing testing tool (Table 1) considers faults that happen to the managed systems. They only reason about faults on the cloud platform or the operators.

This dissertation explores how to develop an automatic testing tool for the operator-system interaction.

1.2 Thesis Statement

This thesis argues that a principled understanding of reliability challenges in cloud system management is key to designing automated testing techniques that can systematically detect critical defects and prevent catastrophic operational failures.

Our empirical study reveals that the fundamental challenge of operator reliability lies in their complex interactions with external entities, including the managed system, the cloud platform, co-located operators, and the user interface, among which the interaction with managed systems is the largest contributor. This insight guides the design of two testing

tools. Acto exploits operators' declarative interface to automatically generate end-to-end tests covering the enormous state space of operations and employs automated oracles that detect silent state mismatches. OAT targets operator-system interactions by synthesizing semantically meaningful values for system-specific properties and injecting faults against managed systems to test failover, recovery, and fault tolerance during operations.

1.3 Contributions

1.3.1 An empirical study of operator failures

This dissertation first develops a comprehensive understanding of today's operator reliability challenges by conducting an in-depth analysis on 412 real-world operator failures. Our analysis reveals that the unique, fundamental challenge of operator correctness comes from multifold complexity of operators' interactions with (1) managed systems, (2) cloud platform, (3) co-located operators, and (4) user interface.

Among these interactions, we found that the interaction with the managed system is the dominant source of operator failures, accounting for 42% of the failures we studied. In essence, today's cloud systems lack well-defined management interfaces for operators. However, many critical management operations have sophisticated semantics and their correctness relies on system configuration, states, execution environments, etc.; many of which are not exposed explicitly. It is challenging for operator developers to capture all fine-grained operation semantics and observe system internal states, especially considering that operator developers may not be system developers (instead, they are system users). As a result, interactions between operators and systems are often *ad hoc* and error-prone.

Beyond the system interaction failures, our analysis also shows significant reliability challenges in operators' interaction with the cloud platform (30%), co-located operators (5%), and the user interface (23%). For each interaction category, our study identifies and characterizes concrete failure patterns. Based on the findings from the study, we discuss the coverage of existing solutions for the operator failure patterns, and pinpoint the key research gaps where new tools and techniques are most needed.

1.3.2 Acto: automatic end-to-end testing for cloud system operators

Given the pressing problem of operator reliability, this dissertation develops a practical testing tool that can be readily applied to a wide variety of cloud system operators. We present Acto, the first automatic technique and tool for end-to-end testing of cloud system 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 be resilient to *misoperations* (i.e., operation errors) by preventing them from driving the system into error states.

To effectively explore the large input space of operators, Acto employs three test strategies. Single-operation testing checks whether the operator can reach different desired states individually. Operation-sequence testing verifies that the operator can reconcile the system correctly from different starting states. Error-state testing evaluates whether the operator can restore the system when it encounters error states.

Because operator bugs typically manifest as silent state mismatches rather than crashes, detecting them requires advanced automated oracles. Acto provides two automatic oracles to check whether the state to which the managed system is reconciled matches the specified desired state. The consistency oracle checks whether the actual system state is consistent with the operator's view and the cloud platform's view of the system. The differential oracle exploits the level-triggering principle that operators follow: the same desired state should be reached regardless of the starting state. Acto compares the resulting system states for the same desired state applied from different starting states and flags any discrepancies.

We apply Acto to eleven popular open-source Kubernetes operators which manage nine cloud systems. Acto found 56 new operator bugs in eleven popular Kubernetes operators, among which 42 have been confirmed and 30 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). Acto reports no false alarm in whitebox mode and maintains a low false alarm rate in blackbox mode (0.19%).

1.3.3 Oat: automatic testing of operator-system interactions

Our empirical study identifies interactions with the managed systems as the single largest source of operator failures (42%), however, no existing tool targets these operator-system interactions. While Acto tests operators end-to-end, it is system-agnostic: it mutates only generic system resource properties and cannot exercise system-specific operations that trigger the most prevalent class of bugs. This dissertation presents Oat, a testing tool designed specifically for the operator-system interaction.

Oat follows Acto's end-to-end testing paradigm, generating operations that trigger state transitions and checking the outcomes with automated oracles, but extends it in two critical aspects. First, Oat generates semantically meaningful values for system-specific

properties to trigger diverse state transitions that exercise operator-system interactions. Generating valid values is a key challenge, since system-specific properties typically accept only a specific set of inputs and random fuzzing produces mostly invalid values that are immediately rejected. Oat addresses this through two complementary techniques: mining developer-written unit tests and usage examples for realistic property values, and querying a large language model to generate values for properties not covered by existing examples.

Second, Oat injects faults against the managed system. Oat injects transient faults (container crashes, network delays, and network partitions) to test whether operators correctly perform failover and recovery from transient error states. Oat also combines fault injection with new desired state declarations to test whether operators can tolerate faults in the middle of operations and still drive the system to the correct state.

Finally, Oat provides enhanced oracle mechanisms to detect bugs that are invisible through the Kubernetes API alone. Optional user-provided state monitors expose system-internal state to check against the desired configuration, while system workloads continuously monitor availability throughout state transitions, catching transient violations that a post-transition check would miss.

We apply Oat to six popular, mature Kubernetes operators which manage critical cloud systems. Oat detected 86 *new* bugs and reported no false alarm. Oat found bugs in every tested operator and bugs in all studied patterns.

Chapter 2 An Empirical Study of Operator Failures

As established in Chapter 1, operator reliability is an emerging and pressing concern: a buggy operator can directly fail bug-free systems in production. Recent work has developed testing and verification techniques [140, 137, 57, 139, 7, 84] for operator-like programs, but these target specific, predefined bug patterns. Without a comprehensive understanding of how real-world operator failures arise, it is unclear whether and how much these efforts have addressed *real-world* operator reliability, as they target specific, predefined bug patterns (§2.1.2), so it is hard to tell if they cover major types of production operator failures.

This chapter presents a systematic study of 412 real-world operator failures collected from 13 popular, mature Kubernetes operators managing a diverse set of cloud systems. Our study is guided by the observation that an operator, by design, involves complex, multi-dimensional interactions— with the managed system, the cloud platform, co-located operators, and the user interface—which introduce unique reliability challenges beyond generic software bugs. Our goal is to (1) demystify real-world operator reliability challenges based on an in-depth analysis of 412 documented operator failures, (2) pinpoint gaps in state-of-the-art techniques for operator reliability, and (3) shed light on potential solutions including system design, runtime support, as well as software testing and verification.

In this chapter, we discuss emerging challenges of operator reliability in the context of cloud system management. In principle, a reliable operator must (1) always reconcile the managed systems to their desired states, (2) recover systems from undesired or error states, (3) tolerate transient faults such as node crashes, and (4) be resilient to misoperations. Recent work developed testing and verification techniques [140, 137, 57, 139, 7, 84] for operator-like programs. However, it is unclear whether and how much these efforts have addressed *real-world* operator reliability, as they target specific, predefined bug patterns (§2.1.2), so it is hard to tell if they cover major types of production operator failures.

Our goal is to (1) demystify real-world operator reliability challenges based on an in-depth analysis of 412 documented operator failures, (2) pinpoint gaps in state-of-the-art techniques for operator reliability, and (3) shed light on potential solutions including system design, runtime support, as well as software testing and verification. Different from recent work [161] on characterizing how *generic* software bugs manifest in operators (see §2.8), our work focuses on the essential complexity of softwarizing cloud system management and the fundamental challenge of ensuring correct interactions between operators and the cloud systems they manage.

Our study yields several findings. A main finding is that *erroneous interactions with*

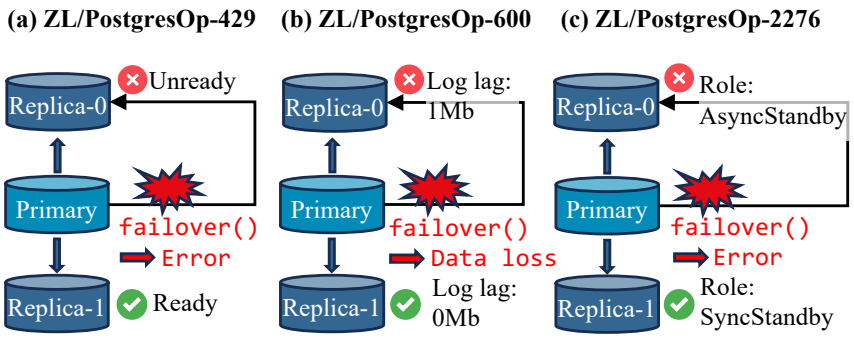


Figure 1: **Three failover operation failures of a commercial PostgreSQL operator (ZL/PostgresOp).** Each failure was caused by the operator's violating a different precondition required by PostgreSQL's failover operation.

the managed systems are the dominant causes of operator failures in the field—they contribute 42% to the studied operator failures, outnumbering other interaction issues. In essence, today's cloud systems lack well-defined management interfaces for operators. However, many critical management operations have sophisticated semantics and their correctness relies on system configuration, states, execution environments, etc.; many of them not exposed explicitly. It is challenging for operator developers to capture all fine-grained operation semantics and observe system internal states, especially considering that operator developers may not be system developers (instead, they are system users). As a result, interactions between operators and systems are often *ad hoc* and error-prone.

Figure 1 illustrates the problem using real-world examples from a commercial PostgreSQL operator. The code for performing a failover operation was reported buggy and got (partially) fixed at least three times; each failure led developers to discover a precondition for safe failover, which was previously unknown to them. The first bug [47] caused a PostgreSQL outage as the operator failed to complete the failover operation—the operator tried to promote a bootstrapping PostgreSQL node to be the next leader and never retried when the promotion failed. The second bug [69] led to data loss: PostgreSQL was configured to run in asynchronous replication mode, and the operator mistakenly promoted a node with large write-ahead log lag to be the new leader. The data loss could have been prevented by promoting a node with up-to-date logs. The third bug [50] failed the failover operation as the operator did not choose the nodes with a `SyncStandby` role as leader candidates. The role is required when PostgreSQL is configured to run in synchronous replication mode. In each case, developers patched the operator to satisfy violated preconditions and added new tests a posteriori.

In addition to the system interaction failures, our analysis also shows significant

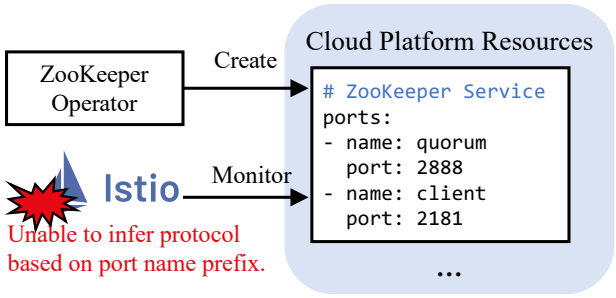


Figure 2: **A co-located operator interaction failure between the ZooKeeperOp and Istio [171].** Istio requires other operators to create Services with port names prefixed by protocols, e.g., "tcp-quorum."

reliability challenges in operators' interaction with the cloud platform (30%), co-located operators (5%), and the user interface (23%). Figure 2 shows a co-located operator interaction failure caused by the ZooKeeperOp violating an implicit convention of Istio. In Kubernetes clusters with Istio installed to monitor network traffic, all Service port names are required to follow Istio's convention of using have the network protocol as the prefix (e.g., "tcp-quorum"). ZooKeeperOp failed to follow this convention, preventing Istio from accurately identifying and categorizing ZooKeeper's communication traffic [171]. This failure shows that the lack of coordination between co-located operators can cause silent monitoring failures.

This chapter makes three main contributions:

- A discussion on emerging challenges of operator reliability for softwarized cloud system management;
- An analysis of 412 operator failures, with a focus on those where the operator failed to manage cloud systems;
- Study dataset: https://github.com/xlab-uiuc/acto/tree/tocs-ae.

2.1 Background

Software operators are management programs running atop modern cloud platforms such as Kubernetes [16], Twine [144], and ECS [95]. They constitute the management plane of cloud systems. Different from traditional infrastructure-as-code (IaC) that are often ad hoc, one-off scripts [122, 44], operators are developed as long-running production services embodied in reusable, well-maintained system programs [30].

In modern cloud platforms like Kubernetes, operators are implemented as custom controllers [106] that continuously reconcile the system from its current states to desired states. The desired states are specified through a *declarative* interface (e.g., Custom

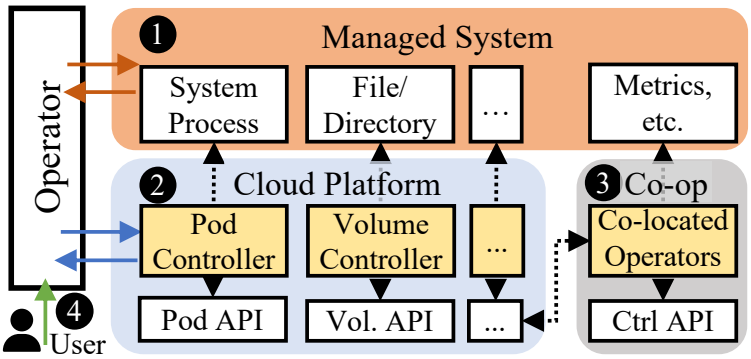


Figure 3: **Multi-dimensional interactions between an operator and its managed system, environment, and users.**

Resource [39] in Kubernetes). In this way, users declare *what* states they want their systems to be in, and the operator addresses *how* to drive systems to reach the desired states. Operators invoke Kubernetes APIs to allocate system resources (e.g., pods and volumes) creating system execution environments. Operators also interact with the running system to update its runtime behavior.

2.1.1 Interactions

An operator, by design, involves complex, multi-dimensional interactions (Figure 3), which introduce unique reliability challenges beyond generic software bugs.

❶ **Interactions with the managed system.** Operators are designed for managing systems and thus must interact with them. Such interactions are often *system-specific* because systems, even when similar, have different APIs. Such interactions are embodied in different forms, including invoking system APIs, executing CLI commands in the system pod, setting environment variables or startup scripts, and changing system configuration files. Take a ZooKeeper operator as an example. To add a new ZooKeeper node, the operator must update the quorum membership. It executes CLI commands to register the new node to the quorum and sets its config startup argument to use a designated configuration file. The operator queries ZooKeeper's `ruok` API to observe its current state.

❷ **Interactions with the cloud platform.** Operators rely on the cloud platform to manage system resources (pods, data volumes, network policies, etc.) that are allocated to the managed systems. In Kubernetes, system resources are represented as API objects [102] that are reconciled by Kubernetes built-in controllers. Kubernetes operators manage system resources by creating and updating API objects with desired configurations, e.g., creating a Pod object with the desired image and resource constraints to run the system.

Tool	Scope	Mechanism
Software Testing and Fault Injection		
Acto [57, 56]	Operator	Functional testing by fuzzing desired states
MeshTest [168]	Controller	Functional testing for ServiceMesh controllers
Sieve [137, 138]	Controller	Model-based fault injection testing
Mutiny [7]	Kubernetes	Injecting general faults to controllers
Formal Verification and Model Checking		
Anvil [139, 136]	Controller	Verifying liveness and safety properties
Kivi [84]	Controller	Model checking controller interactions
Programming and System Support		
DCM [142]	Controller	Declarative programming support
Consistent read [37]	Controller	Preventing reading inconsistent state from caches
Garen [74]	Controller	Transaction support for controllers

Table 1: **Recent efforts on improving operator/controller reliability.** None of them addresses how operators interact with the managed systems (the focus of this paper).

❸ **Interactions with co-located operators/controllers.** Operators may interact with other operators or custom controllers on the cloud platform. In Kubernetes, operators usually avoid doing so directly; the interactions happen implicitly when two or more operators/controllers manage the same set of system resources. For example, Istio is a custom controller that modifies system-level communications by injecting sidecars into system containers. The altered policy may conflict with how the operator manages the system's network.

❹ **Interactions with user interface.** Kubernetes operators define user interfaces in the form of Custom Resources (CRs) [39]. Here, "*users*" refer to entities, like upstream services or AI agents [71], that define the system's desired states. Each CR specifies a collection of properties describing the state of the managed system, such as container images, configurations, replica counts, etc. Operation commands are embodied by specifying desired states. Desired states are declared by creating CRs and assigning values to their properties.

2.1.2 Existing Efforts

Given the importance of operator reliability, recent work focuses on improving Kubernetes operators. Table 1 categorizes existing efforts into (1) software testing and fault injection, (2) formal verification and model checking, and (3) programming and system support. Note that most techniques do *not* target operators for cloud systems, but focus on controllers—in modern cloud platforms like Kubernetes, operators are implemented as

custom controllers [106]. Testing tools like Acto [57] and MeshTest [168] check that controllers correctly reconcile the system to the desired state, but Acto only reasons about platform-level properties in the desired state, and MeshTest focuses specifically on service mesh controllers. Fault injection tools like Sieve [137] and Mutiny [7] test whether controllers can tolerate faults, such as stale reads and message drops, in the interaction with the cloud platform. Anvil [139] verifies general liveness properties of controllers, but abstract away system semantics. Kivi [84] uses model checking to verify the inter-controller interactions. Programming and system support like DCM [142], consistent read [37], and Garen [74] provide stronger platform guarantees for controllers. None of these efforts reason about semantics of managed systems, thus cannot address operator-system interactions.

Table 1 does not show developer-written tests in the form of unit, integration, and system tests. However, as reported by prior work [57, 137, 168], existing manually written tests rarely capture state transitions or cover failure scenarios. Most of them are unit tests that only test operator code without reasoning about the managed systems or system environments.

2.2 Methodology and General Findings

2.2.1 Methodology

We collected a dataset of 412 failure cases of 13 popular Kubernetes operators and conduct a systematic analysis. Table 2 lists the studied Kubernetes operators and their information.

The operators are selected with the following guidelines: (1) they are all open-source projects so that we can reproduce the failures and thoroughly understand their root causes. (2) they cover a diverse set of modern cloud systems, including server systems (e.g., MySQL and PostgreSQL), distributed systems (e.g., Kafka and ZooKeeper), platform runtimes and frameworks (e.g., Knative and KubeBlocks); we deliberately selected two PostgreSQL operators to compare different operators of the same system. (3) they are all mature software projects developed by either the official developers of target systems or companies that provide commercial services based on the systems. The sizes of the projects are typically tens of thousands of lines of code.

For each studied operator, we randomly sample a hundred closed, fixed issues that report failure cases from its issue database and manually inspect every issue. We filter out feature requests, user questions, or issues related to building and testing. We only consider closed issues that conclude root causes with sufficient information. Finally, we collected 412 operator failure cases (the last column in Table 2 shows the number of failures of each studied operator).

Operator	System	Dev.	# Stars	LOC	# Cases
CassOp	Cassandra	K8ssandra	176	30K	17
CN/PostgresOp	PostgreSQL	EDB	3,807	106K	37
CockroachOp	CockroachDB	Official	288	18K	14
KafkaOp	Kafka	Strimzi	4,631	195K	47
KnativeOp	Knative	Official	179	18K	13
KubeBlocks	Multiple	ApeCloud	1,781	156K	33
MinIOOp	MinIO	Official	1,133	17K	63
MongoOp	MongoDB	Percona	312	28K	30
RabbitMQOp	RabbitMQ	Official	830	15K	16
SolrOp	Solr	Official	243	21K	21
TiDBOp	TiDB	Official	1241	230K	35
ZooKeeperOp	ZooKeeper	Pravega	362	6K	22
ZL/PostgresOp	PostgreSQL	Zalando	4,133	34K	64

Table 2: **Thirteen Kubernetes operators we studied.** "# Cases" refered to the number of studied failures.

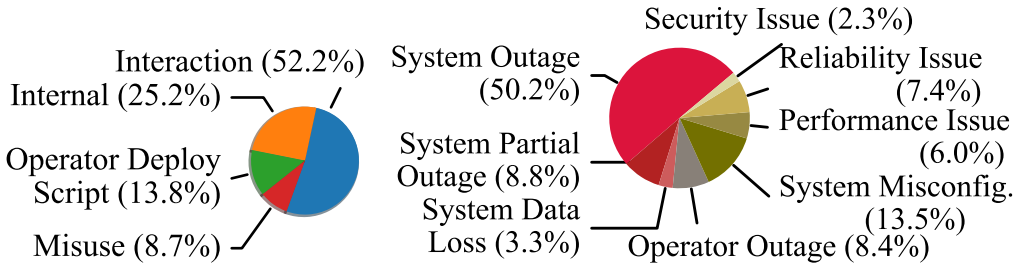
During our analysis, each failure case was analyzed by at least two authors to minimize human errors and subjectiveness in the interpretation and categorization.

2.2.2 General Findings

Finding 1: *The majority (52.2%) of the studied operator failures are caused by defects in operators' interactions with external entities (see §2.1.1), which significantly outnumbers bugs in operators' internal program logic.*

Figure 4a shows the distribution of root-cause locations of the 412 operator failures studied. The results show that defects which manifested via interactions are dominating operator failures, which are two times more prevalent than bugs in operator programs (e.g., `nil`-pointer dereference, logic flaws, etc.) referred to as "internal". This finding corroborates recent analysis [57] that unit tests are insufficient to test operator reliability, because unit tests only target program-level correctness but can hardly exercise an operator's external interactions (e.g., with the systems and environments). Unfortunately, as reported in [57], many existing operator projects heavily rely on unit-level testing for quality assurance. Therefore, in this paper we focus on interaction-related failures which are unique to operators. Bugs internal to operators are not fundamentally different from traditional software bugs which have been studied extensively in the literature.

The remaining failures were caused by defects in the deployment scripts of the operators (mostly Helm Charts [62]) and by misuses of the operators. While these cases are interesting, they are orthogonal to the operator's design and implementation, and thus



(a) **Root-cause locations.** (b) **Consequences.**

Figure 4: **Root-cause locations of the studied 412 operator failures (4a), and the consequences of the 215 interaction-related operator failures (4b).**

are not the focus of this paper.

Finding 2: *The majority (62.3%) of interaction-related operator failures had a catastrophic impact, e.g., system full outages, partial outages, and data loss.*

Figure 4b shows the consequences of the 215 interaction-related operator failures. 50.2% (108/215) resulted in full outages of the managed systems. For example, MongoOp failed to reconfigure MongoDB cluster membership after changing pod IPs [32], leaving MongoDB pods unable to connect with each other and causing a full outage. 8.8% caused partial failures of the managed systems, making certain important features unavailable such as data backup services, e.g., MongoDB lost the backup service due to inconsistent TLS configurations among the backup agents and Mongod [116]. 3.3% caused silent data loss (e.g., Figure 1b).

The other 29.3% (63/215) interaction failures led to undesired system behaviors including misconfigurations, degraded performance, reliability issues (e.g., incorrect replicas), and security risks (e.g., incorrect permissions). Although these failures did not cause explicit system outages, they are harder to detect and have severe production implications.

Only 8.4% (18/215) interaction failures affected the operator programs (e.g., crashes and hangs). For operators, such consequences are arguably the least severe as they do not directly affect managed systems in production, even though they result in management service unavailability (e.g., the system cannot autoscale after the operator crashes).

Finding 3: *Failures of managing systems is the largest category (42.3%) among all operator interaction failures.*

Table 3 shows the distribution of failures manifested on different types of interactions (see §2.1.1). The largest category is operator-system interaction failure, i.e., the operator failed to manage systems. The operators' interactions with the cloud platform

Operator	❶ System	❷ Platform	❸ Co-located Op.	❹ User	Total
CassOp	2	0	1	5	8
CN/PostgresOp	15	1	0	4	20
CockroachOp	4	6	0	2	12
KafkaOp	7	9	0	5	21
KnativeOp	0	4	1	8	13
KubeBlocks	12	7	0	1	20
MinIOOp	3	5	1	7	16
MongoOp	15	4	0	1	20
RabbitMQOp	3	10	1	0	14
SolrOp	5	2	2	2	11
TiDBOp	9	7	1	3	20
ZooKeeperOp	9	3	2	6	20
ZL/PostgresOp	7	7	0	6	20
Total	91 (42%)	65 (30%)	9 (5%)	50 (23%)	215

Table 3: **Distribution of different types of interaction failures of the studied operators.**

(Kubernetes) and user interfaces also make up a significant percentage of the studied failures. The former tells the complexity of managing system resources (e.g., pods, volumes, and networks) for systems, and the latter shows the challenge of correctly implementing complex declarative operation interfaces [142]. Operators' interaction with co-located controllers has a small percentage, as systems are often exclusively managed by one operator. The failed interactions are mostly with third-party controllers managing low-level resources such as networks and telemetry.

While the interactions with managed systems are error-prone and caused the most failures, few existing studies or tools (see §2.1.2) address them. Hence, in the remainder of this paper we focus on operator-system failures to understand why operators failed to manage cloud systems.

2.2.3 Threats to Validity

Representativeness of the selected operators. All 13 studied operators in our dataset target the Kubernetes platform. While Kubernetes is the de facto cloud platform and its state-reconciliation pattern is shared by other platforms (Twine, ECS, Borg), operators on other platforms may exhibit different failure distributions due to differences in resource abstractions, API designs, and platform guarantees. We selected operators managing a diverse set of systems, including databases, messaging systems, platform runtimes, developed by both official system teams and commercial vendors. However,

operators managing other types of systems (e.g., Machine Learning training frameworks) could have failure patterns underrepresented in our dataset.

Representativeness of selected failures. We collect failures from publicly reported, closed issues with identifiable root causes. This methodology is biased toward failures with visible symptoms, such as system outages, crashes, and explicit errors. Failures that manifest as silently incorrect behavior can be underrepresented: an operator may drive the system to an implicit error state, such as degraded durability guarantees, weakened security configurations, or suboptimal performance, without user noticing. Similarly, transient failures, for example, brief unavailability during rolling upgrades or failovers that resolve before users investigate, are less likely to be reported.

2.3 Failures of Managing Systems

An operator manages cloud systems through continuous *state reconciliation* [57, 137, 139, 16]. The operator observes the state of the managed system. If the current system state deviates from the desired state, it issues management operations to reconcile the current state to the desired state. For example, if the number of replicas in the desired state is larger than that in the current state, the operator will scale up the managed system. The scale-up operation would take a series of actions to add a new replica node, e.g., (1) allocating a new pod, (2) running a replica node using the new pod, and (3) updating the membership of the system to add the new replica. A reconciliation is invoked when users update the desired state or the current state changes (e.g., unexpected failures). Operator-system interactions during a management operation include:

- **System API.** An operator calls APIs of the systems to invoke their internal procedure, e.g., calling PostgreSQL's API to start `failover` (Figure 1).
- **System configuration.** The operator updates the system's configuration by updating configuration files, databases, or ConfigMap objects [34].
- **Execution environment.** The operator changes the execution environment of its managed systems, such as the files and environment variables.
- **Resource provisioning.** The operator allocates (and de-allocates) system resources such as pods [117], volumes [114], and services [129] for the managed system based on its configuration or scaling operations.

Finding 4: *We find four main failure patterns (Table 4):*

- *The majority (63.7%) of studied system-management failures are caused by the operator violating its managed system's operation semantics.*

Patterns	Description	Fail. # (%)
Semantic violations	The operator violates operation semantics required by the system (Fig. 1, 5a–5b).	58 (63.7%)
State observability	The operator fails to observe internal states of the system (Figure 5d).	15 (16.5%)
Version incompatibility	The operator fails to handle inconsistent behavior across app. versions (Figure 5e).	11 (12.1%)
Mishandling system errors	The operator mishandles errors returned by the system (Figure 5f).	7 (7.7%)
Total		91

Table 4: **Patterns of operator-system interaction failures.** Figure 5 provides concrete examples of each pattern.

- *A significant percentage (16.5%) of management failures are caused by the gaps that prevent the operator from observing system internal states.*
- *Incompatibility between the operator and its managed system also caused a significant percentage (12.1%) of failures, triggered by upgrading system versions.*
- *The remaining cases (7.7%) were caused by the operator mishandling system errors.*

2.3.1 Operation Semantic Violations

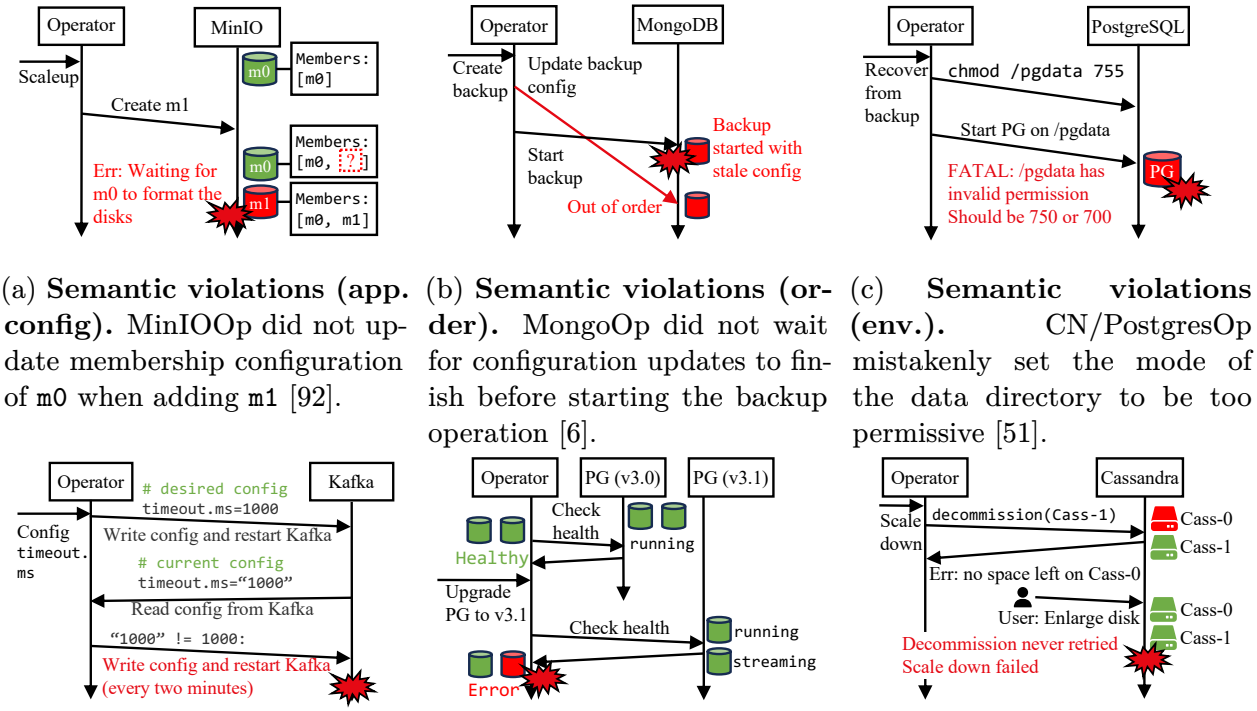
The most common failure pattern is that the operators failed to satisfy the operation semantics of the systems; as a result, they issued unsafe operations that broke the systems.

Cloud systems are large, sophisticated systems, with complex management semantics that define how they should be managed correctly. Unfortunately, in practice, such semantics are not always explicitly documented and are rarely formally specified. Hence, it is difficult for the operator to comprehend and encode a system's operation semantics in a sound and complete manner, especially when operator developers may not be system developers.

Finding 5: *Violations of system operation semantics are not accidental, but reflect the essential complexity of softwarizing cloud system management. The semantics are not explicitly documented or formally specified, and are often encoded based on experience.*

We discuss common violated system operation semantics.

2.3.1.1 System Configuration Many operations are done by changing the system's configuration at runtime, either by updating a configuration file (which requires restarting



(a) **Semantic violations (app. config).** MinIOOp did not update membership configuration of `m0` when adding `m1` [92].

(b) **Semantic violations (order).** MongoOp did not wait for configuration updates to finish before starting the backup operation [6].

(c) **Semantic violations (env.).** CN/PostgresOp mistakenly set the mode of the data directory to be too permissive [51].

(d) **State observability.** KafkaOp fails to observe that the current configuration matches the desired one due to type mismatch, resulting in infinite restarts [73].

(e) **Version incompatibility.** PostgreSQL v3.1 introduced a new healthy state "streaming," which confused ZL/PostgresOp as it only considers "running" as a healthy state [100].

(f) **Mishandling system errors.** CassOp never retried the failed node decommission operation, after the user resolved the error [105].

Figure 5: **Real-world examples of operator-system interaction failures (Tables 4 and 5 define the categories).**

the system process) or by calling the system's configuration API/CLI. It requires operators to manage system configuration correctly. Software configuration management is a known challenge and is well studied in the literature [**xu:16**, 135, 162, 163, 94, 166].

Configuration issues in the operator failures have very different patterns. Prior studies on software configuration focus on validating individual configuration values against their type, range, and system constraints, as their violations are reported to be dominating errors [163]. However, we observe no traditional configuration error (e.g., no individual parameter error). Our hunch is that the practice of automating configuration by operators minimizes inadvertent mistakes and errors.

Instead, deep semantics of system configuration are surfaced and hard for operators to capture. Among the 14 configuration-related operator failures, 11 of them involve interdependent configuration across nodes and components. *Few technique addresses them.* Only three have offending configuration in the same configuration file, but involving

Patterns	Description	Fail. # (%)
Configuration	Configuration of the managed system across nodes and components (Figure 5a).	21 (36.2%)
Ordering (multiple operations)	Order dependency among multiple inter-dependent operations (Figure 5b).	18 (31.0%)
Precondition (single operation)	Preconditions of an operation in terms of system states (Figure 1).	11 (19.0%)
Environment	Execution environment of the app. (Fig. 5c).	8 (13.8%)
Total		58

Table 5: **The types of violated operation semantics.**

multiple parameters (two violated order dependencies and one violated a value dependency [27]).

Finding 6: *Deep semantics of system configuration, including cross-parameter, cross-component, and cross-node configuration, are the sources (14/14) of violations that lead to misconfiguration-related operator failures.*

Figure 5a shows an example of cross-node misconfiguration where different MinIO replica nodes should have consistent membership list, which was violated by MinIOOp due to a bug. Such patterns are common but manifest in different forms. For example, in four cases, operators set inconsistent TLS configurations among different system components, causing interoperability issues. In several cases, the configuration involves different parameters across components, e.g., MongoOp updated MongoDB servers to `tlsRequired`, but it did not enable TLS for the MongoDB monitor.

A more subtle issue is the cross-component dependency between configuration and code. For example, when the TiDB cluster runs with TiFlash component, the Placement Driver (PD) needs to have placement-rules enabled [113]. When MongoDB cluster runs in sharding mode, all Mongod instances are expected to run with the `shardsvr` configuration [96].

The remaining seven failures all fall into a simple pattern—the operator failed to update the configuration of the running system. The essence is that system's configuration interfaces (both file, databases, and API) are overly complicated, e.g., multiple files with overwriting relationships, and multiple databases with different scopes. The operator updated the wrong file [169, 15, 14] that was not read by the system, or the wrong database with limited scope [127, 42, 118]. In all these cases, the operator believed a successful configuration update, while the system ran with an old configuration.

2.3.1.2 Ordering Due to the asynchronous nature of operations and their dependencies, we find that incorrect ordering of concurrent, interdependent operations is a frequent failure pattern. In 31.0% (18 out of 58) of operation semantic violations, the preconditions are violated due to incorrect coordination of interdependent operations issued by the operator (Table 5).

No order enforced. In 12 cases, the operator did not enforce any order against operations that have order dependencies. Due to the asynchrony of distributed operations, an operation issued early in time is not guaranteed to finish before an operation issued later. This caused two failure patterns: (1) a later operation was executed before an early operation and invalidated the early operation, as shown in Figure 5b, and (2) a later operation was executed during the execution of an early operation, causing interference. For example, to scale down a CockroachDB cluster, CockroachOp first stops the CockroachDB process and then removes the container. However, if the container is deleted during the stop operation, the operation fails, leaving the cluster in an inconsistent state.

Incorrect ordering. In the other six failures, the operator enforced a wrong order. For example, when launching a PostgreSQL cluster, the KubeBlocks starts PostgreSQL nodes one by one. A correct order is to start the leader node first and then follower nodes, because each follower must contact the leader to join the quorum. However, KubeBlocks starts nodes randomly. In KB-3485, KubeBlocks started a follower first, and the follower kept waiting for the leader to join; meanwhile, KubeBlocks also kept waiting for the follower to enter the ready state before starting the leader node, causing a deadlock.

Finding 7: *Correct (partial) ordering of management operations needs to be enforced to avoid dependency violations. Atomicity of operation execution, e.g., by lightweight transaction, may help avoid conflicts of concurrent operations.*

2.3.1.3 Preconditions A generic pattern is the violation of preconditions of an operation, which applies to individual operations. (Ordering of multiple operations can be viewed as a special case.) Figure 1 shows an example that the PostgreSQL failover operation has at least three preconditions: the target node must (1) be in a ready state, (2) not lag behind, and (3) have the role of SyncStandby. Any violations fail the failover operation.

We find that PostgreSQL failover is inherently error-prone. Both PostgreSQL operators (ZL/PostgresOp and CN/PostgresOp) introduced *multiple* failures of failover operations. In one case, CN/PostgresOp failovers a node when it has a full disk, without checking if

the target node has sufficient disk space, a precondition. The target node has the same disk capacity and is also full; failover exacerbated rather than resolved the error. In essence, the failover operation of PostgreSQL is error-prone by design, requiring operators to enumerate fine-grained preconditions on system internal states.

Some preconditions require additional operations to satisfy. For example, user traffic needs to be rerouted from a node before starting a failover operation on the node in KubeBlocks; data needs to be migrated before shutting down a node in Solr. Such preconditions are commonly violated (by multiple operators), causing partial failures [101] and data loss [141, 40, 154].

Finding 8: *Violations of preconditions on system states mostly happened to operations that handle failures (e.g., failover) and reduced capacity (e.g., downscaling); these operations tend to have complex, subtle preconditions and have major impact on systems.*

2.3.1.4 Environment The remaining eight failure cases were caused by the operators incorrectly preparing or managing the execution environment of the managed systems. In these cases, the operator either did not create the files or data directories expected by the systems, or misconfigured their permissions (e.g., Figure 5c). As a result, the systems failed to read from or write to the designated locations. We find that all these failures happened when the operators attempted to revamp an existing execution environment or restore a previously created environment from a backup, instead of a normal startup procedure from a clean-slate environment.

2.3.2 State Observability

The state-reconciliation principle relies on the observability of system states. Different from the cluster states that are encoded in well-defined state objects [102], it is challenging to observe a system's internal states which are less defined and do not have unified schemas. As the second largest causes (16.5%), the operator cannot reliably check if the current system state matches the desired state.

Finding 9: *All the observability-related failures are caused by ad hoc monitors of the system's internal state or ad hoc encoding of the system state.*

Readiness and liveness monitors. Kubernetes allows operators to register monitors (called probes [35]) that check readiness and liveness of managed systems. Eight (out of 15) failures were caused by *ad hoc*, unreliable probes. Unreliable readiness probes may incorrectly instruct clients to connect to unready systems and fail client requests. A

common pattern is to use approximate signals, such as container startup [68] and DNS resolution [24], to indicate system readiness; such signals are flaky.

Unreliable liveness probes may incorrectly instruct Kubernetes to reboot the system containers, causing disruptions. In all three cases [153, 90, 85], the liveness probes reported false alarms due to timeout of the probes when systems were running slow. The three issues were resolved by enlarging the timeout and reducing runtime probing overhead, which are workarounds rather than fundamental resolutions.

Encoding states. Without system support, operators have to implement their own logic for interpreting the system's current status and encoding them in a way that can be compared with the desired state. In 7 (out of 15) cases, the operator failed to check the semantic equivalence of the system's current state and the desired state. This causes operators to keep reconciling systems even when the system has already reached the desired state, as shown in Figure 5d. The encoding is nontrivial, e.g., CN/PostgresOp checks if a PostgreSQL instance is stopped based on the text output of the `pg_ctl status` command, which is brittle as the output is different when PostgreSQL runs in different locales.

We inspected the probes implemented by the studied operators; most of them use simple, brittle checks (e.g., dummy client requests), which are unreliable and cannot address real-world failure modes (e.g., slow, gray, partial, and metastable failures) [65, 67, 86, 60], despite many iterations (e.g., [107]). The deficiency of probes corroborates a recent industry report [52]. Integrating advanced observability techniques [66, 82, 111, 81] and state-encoding interface are desired but are challenging for diverse cloud systems.

2.3.3 Version Incompatibility

Version incompatibility is the third largest root causes (12.1%) of operator failures. The failures were manifested when the operator upgraded the managed system—the new system version is incompatible with the operator.

Finding 10: *Version incompatibility issues are often rooted in ad hoc, brittle assumptions made by the operators.*

Figure 5e shows an example where ZL/PostgresOp assumed statuses other than "running" to be erroneous, which is broken when PostgreSQL introduces a new healthy state. In another case [131], CN/PostgresOp checks if WAL archive is available based on the existence of a successful archive. This works in the old versions of PostgreSQL that always creates WAL archives periodically. This assumption is broken when the new version of PostgreSQL changed its behavior—when there is no database activity, WAL archive will be skipped. This new behavior makes CN/PostgresOp believe the WAL archive is never

available and impair backup operations.

Version compatibility is a classic software reliability problem and has been recently studied in the context of software upgrades [167, 165]. However, different from traditional software compatibility, it is challenging to ensure compatibility between the operator and the managed system, without a clean management interface. Specifically, an operator is expected to work with different versions of the managed systems as software upgrading is a basic feature of all the studied operators (Table 2). We assert that, without a well-defined management interface, compatibility between an operator and its managed systems cannot be systematically solved.

2.3.4 Mishandling System Errors

Error handling is a long-lasting challenge and a well-studied problem [164, 59, 29, 83]. Since the operator should reconcile system to the desired state from any state (including error states), it is responsible for handling system errors. However, as mature cloud systems already implement extensive error handling, an interesting question is—what kinds of errors should and should not be handled by the operators? In principle, an operator should handle errors that cannot be handled by the system; however, the line is often blurry.

In 5 (out of 7) failure cases, the operator did not handle errors—when the system returned an error code, the operator chose to exit (e.g., Figure 5f). It is a simple, safe strategy, but may miss opportunities.

In the other two cases, the operator mishandled system errors. For example, TiDBOp treated all errors in the system pod to be transient, and waited for them to recover before issuing any other operations. However, permanent errors can cause the operator to hang, preventing critical operations like upscaling that could mitigate failures. The fix is to prioritize upscaling operations over waiting for pod recovery.

2.3.5 Discussion

The above study reveals the significant challenges of correctly managing today's cloud systems using softwarized operators. Certainly, an ultimate solution is to rethink and redesign cloud systems that are fully autonomous, eliminating the needs of external management. While such solutions are revolutionary and fundamental, they may not be realized in a short term. Below, we discuss potential directions to improve cloud systems manageability in a more evolutionary way.

2.3.5.1 The Case for Management Interfaces

There is an alarming lack of techniques to validate whether an operator correctly follows system's management operation semantics. The current practice of relying on system documentation to implement correct operations is error-prone—documentation is often incomplete and vague, and sometimes even wrong. As evidence, operation semantic violations are prevalent and have severe consequences (§2.3.1). There is a pressing need to build management interfaces that precisely describe operation semantics for systems.

We envision that system management interfaces are much simpler programs compared to the original system but preserve its management operation semantics. For example, for an system API, the management interface should precisely describe the conditions for this API to succeed, including constraints on configurations, environments, and system states. Management interfaces should also be versioned and evolved as the systems evolve to prevent version incompatibility (§2.3.3).

Management interfaces can be used for different purposes. We envision testing techniques that use management interfaces as mocks to check if an operator correctly interacts with a system, and validation techniques that allow operators to perform dry runs of its operations before interacting with the system. The interfaces could also enable developers to measure the interface complexity and evaluate the manageability of their systems in a similar vein as [9, 10].

Developing management interfaces is challenging as mature systems tend to have complex management operation semantics. We envision that management interface for each API could be derived by preserving only the conditions that appear along the path of the API invocation and abstracting away other details, similar to how performance interfaces [70] capture only latency-relevant behaviors.

2.3.5.2 Formal Model

We envision that the formal model of a management interface is written as a state machine and is used as an executable specification. The state machine model naturally captures asynchronous and concurrent interactions between operators and systems. The state machine model includes actions representing all management APIs exposed by systems (e.g., PostgreSQL's API to start `failover`), as well as background actions (e.g., a PostgreSQL node gets in a ready state). The model should also define a collection of bad states which are reachable by invalid operations, such as starting `failover` on an unready PostgreSQL node.

The model could enable formal verification of operator programs. Existing verification frameworks like Anvil [139] lack a systematic way to model the interactions between the operator and the system. It burdens operator developers to manually write specifications

of the system APIs, which is ad hoc and hard to be complete. In fact, a bug was found in a ZooKeeper operator verified by Anvil, which was caused by "*an incomplete specification of a trusted ZooKeeper API that did not cover ZooKeeper misconfigurations [139].*"

The model could also enable model checking of the operator and runtime monitoring/verification. For example, model checking the operator and the model together can detect occurrence of bad states (safety violations) caused by buggy operations. Runtime verification with the model can also report and block buggy operations that violate preconditions.

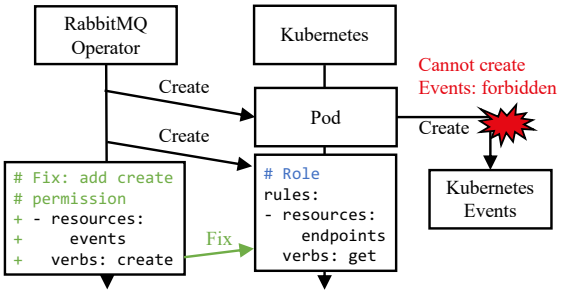
The model is only useful if it accurately captures the management operation semantics. A potential solution that has been proven effective for key-value stores [12] and file systems [126] is to perform property-based testing to compare the behavior of the model and the system. There are many exciting research problems that lie in how to develop, maintain, and even synthesize formal models as management interfaces.

2.4 Failures of Platform Interaction

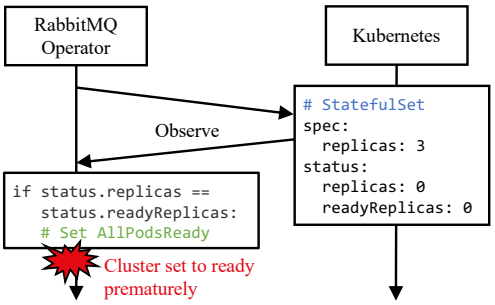
Operators, as custom controllers, are essentially clients of the cloud platform. They must continuously interact with the platform to allocate, manage, and observe the system resources (e.g., pods [117], volumes [114], network [129]) that constitute the system's environment. Modern cloud platforms like Kubernetes follow the state-reconciliation principles, where the desired states and current states of the system resources are represented as a set of API objects. Operators (and controllers) manage the system resources by creating and updating the API objects, and interact with the system resources created by the platform to satisfy system management requirements.

Finding 11: *There are four primary failure patterns in operator-platform interactions (Table 6):*

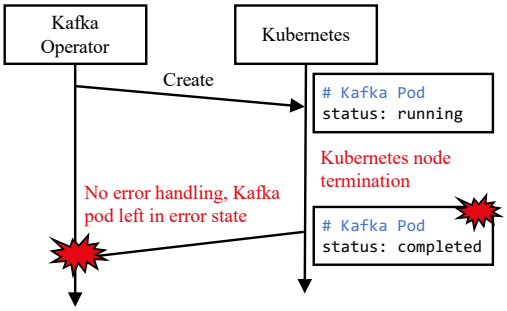
- *About half (46.2%) of the failures are caused by violating the requirements for creating and using the platform resources via the API objects.*
- *A significant percentage (24.6%) of the studied failures are caused by operators' observability issues, where the operators misinterpret the current state of the platform resources or observe a stale state of the resource.*
- *Incorrect error handling also caused a significant portion (16.9%) of the studied failures, where the operators failed to handle platform errors.*
- *The rest (12.3%) are due to operators making assumptions that are not portable across different platform implementations.*



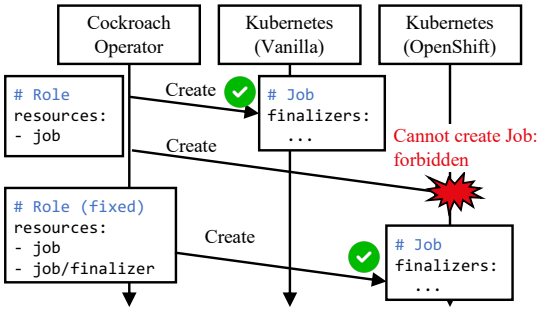
(a) **Resource requirement violation.** RabbitMQOp misconfigured the RBAC role for pods, missing the permission to create events [119].



(b) **State observability.** RabbitMQOp misinterprets the StatefulSet API object, reporting a healthy status when no pod is created [2].



(c) **Mishandling platform errors.** KafkaOp unable to handle failed pods caused by node failures, leaving the Kafka in a degraded state [134].



(d) **Platform compatibility.** CockroachOp failed to create jobs with finalizers on OpenShift due to stricter security policies [123].

Figure 6: **Real-world examples of operator-platform interaction failures. Tables 6 defines the patterns.**

2.4.1 Resource Requirement Violation

The most common failure pattern is the violation of requirements for managing the platform API objects. Platforms like Kubernetes provide a declarative interface, where operators manage the resources by declaring their desired state configurations without worrying about how to create and update them. However, the underlying resources often have implicit operational requirements that are not enforced by the platform's schema validation.

2.4.1.1 Misconfiguration. Operators manage system resources through the platform's declarative interface by specifying the desired state of platform API objects—a process that resembles traditional system configuration. In 18 cases, operators violated configuration constraints of platform API objects, and these failures exhibit characteristics different from traditional misconfiguration issues studied in prior work [**xu:16**, 135, 162,

Patterns	Description	Fail # (%)
Violating resource requirements	Operators violate requirements for managing platform resources (Figure 6a).	30 (46.2%)
State observability	Operators incorrectly observe and interpret the state of platform resources (Figure 6b).	16 (24.6%)
Mishandling platform errors	Operators fail to handle platform errors. (Figure 6c).	11 (16.9%)
Platform incompatibility	Incompatibility across different platform implementations (Figure 6d).	8 (12.3%)
Total		**65**

Table 6: **Patterns of platform interaction failures.** Figure 6 provides examples of each pattern.

163, 94, 166]. None of the 18 cases involved violations of basic structure or type constraints, likely because Kubernetes enforces structure and type constraints through its strongly-typed API schema definitions. Instead, operator misconfigurations violate higher-level semantic constraints-rules about how multiple API objects must relate to each other, or implicit requirements for specific API objects.

In 11 cases, misconfigurations violated constraints spanning multiple API objects. A recurring issue (six cases) involved permission misconfiguration in the RBAC (Role-Based Access Control) API object. Figure 6a shows a failure where the operator omitted the "create" permission for Event resources in the RBAC role created for the RabbitMQ server pods. While the RBAC object itself is syntactically correct, the missing permission violates the requirement of the pod and causes loss of observability. Another recurring pattern (four cases) involved name collisions between multiple platform objects. Since Kubernetes objects are uniquely identified by the combination of name and namespace, creating two objects with identical names causes them to be treated as the same object by the platform. This leads to one object overwriting another.

The remaining seven misconfigurations violated implicit constraints of a single platform object. In SolrOp-84, the operator violated an implicit requirement for Kubernetes headless Service [129], where the `port` and `targetPort` properties must be identical. The operator set them to different values, breaking service discovery. This shows how operator misconfigurations often involve subtle semantic rules not enforced by schema validation.

While several analysis tools have been developed for checking Kubernetes misconfigurations [77, 76, 36], a significant gap remains to apply them to operators. Operators dynamically construct resource configurations at runtime based on the user

inputs, cluster state, and complex reconciliation logic, making it challenging to statically analyze all possible configurations they might generate. This dynamic nature means misconfiguration bugs may only manifest under specific input combinations or cluster conditions that are not exercised during development testing.

2.4.1.2 Order Dependencies. Operators need to configure multiple platform API objects (e.g., creating a pod along with its required ConfigMap) or configure the same API object multiple times as its state evolves (e.g., updating a StatefulSet's replica count, then its image). Operators must correctly order configuration operations to satisfy these dependencies. Similar to order dependencies in system interactions, we observed two types of ordering violations: incorrect ordering and lack of enforced ordering.

In four cases, operators enforced incorrect ordering, creating circular dependencies with the platform's reconciliation logic. Although the declarative interface conceptually allows operators to specify desired states in any order, in reality, there are implicit ordering requirements due to resource dependencies. In KubeBlocks-4115, the operator attempted to read an updated configuration from a ConfigMap inside its pods' PostStartHook, which executes after the container starts. However, Kubernetes follows a specific order to only updates mounted ConfigMap after the PostStartHook has completed. This created a circular dependency—the PostStartHook needed the updated ConfigMap to execute, but the ConfigMap update required the PostStartHook to finish first.

In three cases, operators failed to enforce proper ordering. Interaction with the platform resources is inherently asynchronous: the platform's built-in controllers asynchronously observe the desired states from operators and reconcile the underlying system resources. Operators need to handle the unready system state where the current system state does not match the desired state yet. For example, in ZL/PostgresOp-713, the ZL/PostgresOp deletes and recreates the service object when the desired service type is changed. ZL/PostgresOp did not wait for the service resource to be fully deleted before recreating it, causing the service creation to fail.

2.4.1.3 Resource Misuse. In five cases, operators correctly configured the system resources but referenced or interacted with them incorrectly. Most cases (4/5) are due to missing namespace qualifiers in the reference, which implicitly resolve to the default namespace. For example, in CockroachOp-906, CockroachOp referenced a service resource without specifying the namespace. When the CockroachDB cluster was deployed in a non-default namespace, the reference incorrectly resolved to the default namespace, causing connection failures. The fifth case tried to write to a file projected by ConfigMap

resource, which is immutable.

2.4.2 Observability Issues

In 16 cases, operators failed to correctly observe or interpret the state of platform API objects.

2.4.2.1 State Mismatch. In 11 cases, operators incorrectly concluded that the current state did not match the desired state, triggering unnecessary reconciliation loops that disrupted service availability. These failures reveal two patterns.

In nine cases, operators failed to check for semantic equivalence of the platform API objects, instead, they rely on shallow equality checks that require exact matches of the state, including semantically irrelevant details like ordering or formatting. In other words, there are multiple states of API objects which satisfy the desired state, but operators' brittle comparisons recognize only a narrow subset as matching, rejecting semantically equivalent but syntactically different states. In KafkaOp-1958, KafkaOp fails to correctly compare the current environment variables in the StatefulSet versus the desired ones. The current environment variables are listed in a different order than the computed desired ones, and KafkaOp considers the current state as not matched with the desired state and keeps updating and restarting the StatefulSet.

In two cases, operators misinterpreted the status fields of platform API objects, misjudging when resources had reached their desired state. Figure 6b shows an example where RabbitMQOp misinterpreted the status of the StatefulSet API object. The operator determines of the RabbitMQ cluster is fully ready by checking if the `status.replicas` equals to `status.readyReplicas`. However, because `status.replicas` represents the number of pods currently created by StatefulSet instead of the desired count, the operator may report the cluster as ready even when no pod has been created.

2.4.2.2 Stale State. Operators often operate on stale states, due to asynchrony and the extensive uses of caches for performance and scalability. Operators are expected to tolerate stale views that lag behind the latest states maintained in the data store. In five cases, the operators fail to handle observing stale states of the platform API objects.

For example, in MongoOp-430, the operator observed a stale StatefulSet marked with a deletion timestamp from the old MongoDB cluster and incorrectly interpreted this as the intent to delete the current MongoDB cluster. The operator then proceeded to delete both the new StatefulSet and its associated PersistentVolumeClaims, causing service disruption

and permanent data loss. Such issues can be avoided by checking the UID of the StatefulSet on deletion.

2.4.3 Mishandling Platform Errors

In 11 cases, operators fail to handle errors from the platform. Just like any services, requests to the platform can fail and operators are required to gracefully handle platform errors. The majority cases (eight cases) are similar to traditional error handling, for example, KafkaOp does not correctly handle pod failures caused by platform node termination (Figure 6c). When a machine running Kafka pods is terminated (e.g., due to spot instance preemption), the pods enter a failed state but the operator does not delete and restart it, leaving the system in a degraded state.

The more complex challenge (three cases) appears when the errors occur to the operator pods, where operators crashed mid-reconciliation and could not correctly resume from the resulted intermediate states. Kubernetes' reconciliation model requires operators to safely restart from any intermediate state. When an operator crashes between steps, it must continue from the intermediate state and reach to the desired end state. For example, in KafkaOp-8401, KafkaOp performs a three-step CA rotation: (1) establish trust to new CA while keeping old certificates, (2) deploy new certificates signed by new CA, (3) remove trust in old CA. If the operator crashes before finishing the step 2, it incorrectly mark the CA rotation as completed by updating the CA generation ID annotations, and never retries to deploy the new certificates. Kafka nodes continue using old certificates signed by the old CA. Failures in this pattern can benefit from transaction support provided by the platform. For example, the KafkaOp can write the three-step CA rotation in one transaction, so that either all or none of them succeed.

2.4.4 Platform Incompatibility

While the Kubernetes API establishes a standardized interface, different implementations, ranging from upstream Kubernetes to managed services like GKE [53] and EKS [3], or enterprise distributions like OpenShift [124], exhibit divergent runtime behaviors. We found 8 failures (12.3%) where operators made assumptions that were not portable across these implementations.

Figure 6d shows a failure due to the divergent security policies between the vanilla Kubernetes and the OpenShift. In CockroachOp-780, the operator successfully created the jobs with finalizers on the vanilla Kubernetes platform, but failed on OpenShift. This failure is because OpenShift requires explicit permission for the finalizers, while other

implementations are more permissive.

This illustrates a critical compatibility challenge: operators validated on a single Kubernetes distribution may have latent portability bugs that only manifest when deployed on other implementations. These incompatibilities fundamentally arise from the Kubernetes API's ambiguous specification. While the API rigorously defines the structure and schema of the API objects (i.e., how desired states should be declared), the runtime behavior of the created system resources remains underspecified. This vague definition causes the semantic divergence across different implementations for the same APIs.

2.5 Failures of Interacting with Co-located Operators

One operator not only interacts with the built-in Kubernetes controllers, but also other custom operators which extends the Kubernetes control plane.

Finding 12: *In most co-located operator interaction failures (8/9 cases), operators interact with other custom operators indirectly. Failures arise implicitly when operators violate undocumented behavioral requirements imposed by co-located operators managing the same resources.*

We identified nine failures caused by problematic interactions with co-located operators. Notably, only one case involved direct interaction where the SolrOp utilizes ZooKeeperOp to manage ZooKeeper clusters [170]. The remaining eight cases have indirectly erroneous interactions: operators created or managed resources in ways that violated implicit expectations of other custom operators which manage low-level resources (e.g., changing network routing between pods). These failures reveal that custom operators add implicit requirements for operators to follow such as expectations about resource naming, labeling, annotation semantics. These requirements are rarely documented in interface specifications or enforced through validation mechanisms, creating a source of brittle integration points.

Figure 2 shows an example that ZooKeeperOp created services with port names that did not satisfy the Istio's expected conventions to have network protocols as the prefixes. Note that both ZooKeeperOp and Istio functioned correctly in isolation, and the failure emerged only when they are deployed together.

The failures caused by broken conventions are similar to the cross-system interaction failures [145]. While cross-system testing can detect such failures, due to the large number of operators in the Kubernetes community, exhaustively performing testing with all other operators is unrealistic. These failures point to a fundamental gap in the Kubernetes ecosystem—the lack of mechanisms for explicitly declaring and enforcing inter-operator contracts. Custom operator developers can specify the assumed conventions and check

them to detect incompatibilities early. For example, Istio can write validation rules to check if the network port names satisfy its naming convention, so that such problems can be detected before RabbitMQ is deployed.

2.6 Failures of Interfacing Users

Operators define declarative interfaces (CRs [39] in Kubernetes) consisting of hundreds or thousands of properties. Different properties describe different aspects of the system state, e.g., replica number, size of storage. Unlike traditional API interfaces where each exposed method corresponds to an explicit function implementation, operator interfaces are loosely coupled to their implementations: operators consume the entire desired state specification and handle all properties within a single reconciliation function.

We identified 50 interface failures (23.3% of all failures) caused by inconsistencies between what the interface exposes and what operators actually implemented.

Finding 13: *Most (43/50) of the operation interface failures are due to missing or incomplete implementations for interface properties. The rest 7 cases are unexpected behaviors of the interface properties due to the ambiguous interface design.*

2.6.1 Missing or Incomplete Implementation.

In 43 user interface failures, the interface properties are either unimplemented or partially implemented. We found three patterns:

- **Partial state transition (22 cases):** The exposed property does not support all possible state transitions. A core requirement for the declarative interface is to support transition from *any* starting state to the desired state. Operators take different actions depending on the difference between the existing system state and the desired state. For example, in MongoOp-895, the operator only partially implements the interface for exposing the cluster. However, it lacks the implementation for turning the expose feature from on to off.
- **Unimplemented properties (14 cases):** The operator is missing the implementation for certain properties exposed in the interface. Operators' operation interface consists of thousands or tens of thousands of properties, making it hard to maintain. This problem is exacerbated by interface and code evolution. For example, in ZooKeeperOp-108, the ZooKeeperOp does not implement for the exposed property `spec.size`. This discrepancy was introduced due to interface evolution, where the developers introduced a new property, `spec.replicas`, to replace the `spec.size` property, but did not remove it from the interface.

- **Zero-value handling (7 cases):** The operators fail to handle or support the zero-values of the properties, e.g., empty array, 0 for integer types. For example, in KafkaOp-1817, when users intended to remove all liveness probes by setting the "livenessProbes" property to empty array, the operator interpreted the empty array as unspecified and applied the default liveness probe configuration. These failures stem from a semantic ambiguity in interface specifications: does an empty value (e.g., empty array, zero, empty string) represent an intentional choice by the user, or does it indicate the property was left unspecified? In Go, which is widely used for operators, distinguishing between "explicitly set to zero" and "unspecified" requires using pointer types where "nil" represents unspecified. However, developers often fail to implement this pattern consistently across all properties.

2.6.2 Unexpected Behavior

In seven failures, operator's implementation for an interface property does not align with users' intention. For example, in MinIOOp-1606, users specified a desired security context for containers and expected it to be applied to all containers created by the operator. However, MinIOOp only applied the security context for the main MinIO containers, but left out the initialization containers and sidecar containers. The unexpected behaviors are fundamentally due to the lack of formal specification for the operators' user interface, where users infer the behavior of the interfaces based on their names and natural language description. A specification which describes how each property affects the system state could serve as the explicit documentation for users to avoid unexpected behaviors.

2.7 Existing Solutions

We analyze the existing solutions for operator reliability challenges and evaluate their coverage of the failure patterns from the study (Table 7). We find there are significant gaps: while existing testing, verification, and platform solutions address some platform, co-located, and user interactions, no existing work targets the system interaction, which is the dominant failure pattern.

2.7.1 Automatic Testing

Automatic testing techniques have been developed to find defects in existing operator implementations by executing them under various scenarios and checking for incorrect behaviors. The key strength of automatic testing is its practicality: it requires little manual effort to apply and can effectively explore large testing space and uncover bugs.

Patterns	Acto	MeshTest	Sieve	Mutiny	Anvil	Kivi	Consistent read	Garen
System (all)	-	-	-	-	-	-	-	-
Platform								
Operation requirement	◐	◐	-	-	●	-	-	-
Observability issue	◐	◐	◐	-	●	-	◐	-
Mishandling errors	-	-	◐	◐	●	-	-	◐
Platform incompatibility	-	-	-	-	-	-	-	-
Co-located op. (all)	-	-	-	-	-	●	-	-
User								
Unimplemented interface	●	-	-	-	●	-	-	-
Undesired behavior	-	-	-	-	●	-	-	-

Table 7: **Coverage of existing solutions for the operator failure patterns.** ● denotes the pattern is addressed, ◐ denotes specific subpatterns are addressed, and "-" denotes the pattern is not addressed. The tools are listed in Table 1. DCM is omitted as it does not address operator interaction failures.

However, testing tools provide no completeness guarantees—they only show the presence of bugs but not their absence.

2.7.1.1 Functional Testing Functional testing validates that operators correctly implement their intended behavior under normal operating conditions without external events such as faults. Our previous work Acto [57] is an automatic end-to-end testing tool which generates desired states covering all properties in the operators' interface and checks if operators always correctly reconcile the systems to match the desired states. MeshTest [168] is another operator testing tool which targets specifically service mesh operators, testing their functionality correctness under different service flow configurations.

Both Acto and MeshTest address the resource requirement violations and the observability issues in the operator-platform interaction (§2.4) by systematically generating different desired states and checking whether operators create correct platform resources. The automation enables comprehensive coverage of the desired state space with minimal manual effort, thus can detect the incomplete implementation in the user interface (§2.6.1). However, they do not systematically explore the space of faults and timing of events, thus cannot address failures involving ordering dependencies and error handling.

2.7.1.2 Fault-injection Testing Fault-injection testing tools complement functional testing by introducing controlled faults and timing events to expose operator reliability issues. Unlike functional testing which does not control the environment or the timing,

fault injection deliberately creates faulty conditions that operators must handle gracefully. This different exploration strategy enables fault-injection tools to discover operator bugs that functional testing cannot reach.

Our prior work, Sieve [137], tests operators by perturbing operators' view of the cluster state, directly targeting the observability issues (§2.4.2.2) and error handling (§2.4.3) when interacting with the platform. Specifically, Sieve forces the operators to read stale states from the cloud platform to test operators' ability to tolerate stale states. Sieve also crashes the operators at different points in the reconciliation loop and checks if operators can correctly resume from the intermediate state.

Mutiny [7] complements Sieve by targeting the error handling issues (§2.4.3) in the platform interaction. Mutiny injects low-level faults, such as bit-flips and message drops, into the network communication between the operator and the cloud platform. This approach tests operators' ability to gracefully handle transient platform errors and data corruption.

2.7.2 Verification

Unlike testing which finds bugs in existing implementations, formal verification proves correctness during operator development through mathematical proofs that implementations satisfy specified properties for all possible executions. Despite the strong correctness guarantee, formal verification requires significant development effort. Operator developers need to write formal specifications and mathematical proofs in addition to the implementation. Additionally, the strong correctness guarantee depends on the trusted model of the external components, such as the cloud platform and the managed system.

Our previous work, Anvil [139], is a verification framework for Kubernetes operators. Operators implemented with Anvil are formally verified to satisfy the eventually stable reconciliation property, which is a general liveness property stating that operators eventually reconcile the system to the desired state, and then always keep the cluster in the desired state. Thus, operators implemented using Anvil are free of platform interaction failures (§2.4) and unimplemented user interface issues (§2.6.1). Anvil requires developers to write a specification, thus the operators are free of undesired behavior issues (§2.6.2). Currently, Anvil cannot verify the interaction with the system (§2.3) and co-located operators (§2.5), due to the lack of accurate models for them. Anvil also cannot address platform compatibility issues (§2.4.4) because it only verifies operators with the vanilla Kubernetes model. Reasoning about compatibility across different platform distributions requires maintaining separate models for each Kubernetes distribution which scales poorly in practice.

2.7.3 Model Checking

Model checking sits in the middle between testing and formal verification in the spectrum of reliability techniques. Unlike testing which directly executes existing implementations or verification which requires mathematical proofs, model checking operates on abstract system models (e.g., finite-state machines) that capture essential behaviors while omitting implementation details. Model checking can analyze *existing* operators by extracting models from their implementations, or it can help developers reason about *system designs* before implementation by modeling state transitions and interactions. Model checking requires less effort than formal verification because it does not demand mathematical proofs, yet it is more systematic than testing because it exhaustively explores all possible states and transitions in the model. However, the abstraction process introduces a fundamental challenge: models must be small enough for tractable analysis (avoiding state-space explosion) yet accurate enough to capture real bugs. Details omitted during abstraction may leave out bugs that model checking cannot detect.

Kivi [84] applies model checking to verify properties about interactions between Kubernetes operators. It does so by manually translating operator implementations into models and verify that the properties always hold for all traces of the system execution. By explicitly modeling multiple operators, Kivi can reason about co-located operator interactions (§2.5). For example, Kivi can check that two operators managing overlapping resources do not create conflicting configurations or interfere with each other's reconciliation logic.

However, Kivi requires non-trivial manual effort to build accurate operator models. Developers must translate operator implementations into state machines while preserving the interaction logics, requiring expertise in both operators and formal modeling. Like other existing techniques, Kivi cannot address system interaction failures as it does not model the systems. Applying model checking to system interaction failures faces fundamental challenges: cloud systems like databases have complex internal states. Accurately modeling the systems while keeping the state space tractable remains an open problem.

2.7.4 Programming and System Support

Programming and system-level support addresses operator reliability issues by providing primitives and guarantees directly from the platform. Unlike testing and model checking which find bugs in existing code, platform support changes the programming model to make certain failure patterns impossible or easier to handle. These solutions require

minimal developer effort but can only address specific, well-understood failure patterns that the platform designers anticipated.

2.7.4.1 Consistent Read When interacting with the cloud platforms, operators often observe stale states due to asynchrony and the extensive use of cache. Failing to tolerate stale states leads to wrong reconciliation actions and causes failures (§2.4.2.2). Tolerating the stale states observed from the cloud platform is challenging. For example, the observed states may "time travel" to a state in the past. Operators are expected to recognize the stale state instead of treating it as a new, unseen state. To address this challenge, Kubernetes introduced the KEP-2340 [37] feature which allows operators to perform consistent read from the Kubernetes API servers without overloading the centralized data store. By ensuring that reads are linearizable, this feature guarantees the operators' view of the cluster never "travel back" in time, thus fundamentally eliminating stale state bugs.

2.7.4.2 Garen (Transaction Support) Operators must handle the intermediate states resulted from crashing in the middle of its reconciliation loop (§2.4.3). Garen [74] proposed Atomic State Reconciliation to provide the transaction support from the platform. Transaction support provides a clean way to eliminate intermediate state vulnerabilities by ensuring all or none updates in one transaction succeed. However, transactions require developers to correctly group updates into transactions: developers need to ensure the related updates are in one transaction and not to group too many update in one transaction which can hurt performance. More fundamentally, Garen's transaction support only applies to platform states. It cannot provide atomicity guarantees for external state changes, such as the updates to the systems' internal state.

2.8 Related Work

We discussed recent efforts on improving reliability of controllers and operators in §2.1.2. The most related work is Acto [57, 56] which is the only technique that targets operators for cloud systems (the others all target controllers). However, Acto does not address operator-system interactions, which, as shown by our study, is a major cause of operator failures. Oat is inspired by Acto and adopts its end-to-end testing paradigm. Unlike Acto, Oat focuses on system-specific properties to exercise how the operator manages the cloud system. Oat also consider various faults, while Acto only performs functional testing without external faults.

Xu et al. [161] characterizes how *generic* software bugs manifest in operators (see §2.8). Our work differs in three aspects. First, we focus on understanding the essential

complexity of softwarizing cloud system management rather than bug characterization. Second, their study characterizes generic software bugs in operators, whereas we focus on operator failures manifested through the interactions between operators and cloud systems; we find interaction failures are dominant root causes but largely overlooked. Third, we emphasize potential solutions in addition to bug characteristics— advocating for rethinking management interfaces and developing OAT to test operator-system interactions.

The operator-system interaction failures can be viewed as a special kind of cross-system interaction failures [145]. Operators are not involved in the control and data planes of cloud systems; they construct the management plane. They interface with cloud systems' internal management components which were studied in [145].

Our work is complementary to studies on reliability of Infrastructure-as-Code (IaC) [44, 121, 122, 61]. Unlike IaC which mostly handles one-shot, static infrastructure deployment, operators manage the entire system lifecycle, handling continuous upgrades, runtime reconfiguration, failover, recovery, etc. This increased complexity forces operators to reason about evolving system states and understand system management operation semantics, which IaC scripts do not commonly address. Consequently, bugs in IaC scripts typically surface at first deployment, whereas operator bugs emerge during reconciliation of systems in production.

Prior work validates the implementation correctness of cloud system interface semantics [87, 88]. In contrast, we focus on operation semantics correctness: *misuses* of the management interface that violate operation requirements.

Our work is also inspired by work on network management analysis [18, 10, 9, 1, 160], especially those on management complexity from a reliability perspective. Compared with routers and switches, cloud systems are more diverse and complex, creating new challenges for management operations.

Previous work [8, 120] studied the inconsistencies between the configuration interface and the implementation, similar to the missing implementation issues in the user interaction failures (§2.6.1). They focus on traditional configuration interfaces where the parameters are completely unimplemented, however, the majority of the operator interface failures involve partially implemented properties and remain unaddressed.

Chen et al. [26] studied and identified security vulnerabilities in Kubernetes operators where their elevated privileges enable attacks that bypass Kubernetes' namespace isolation. In contrast, our study on platform interaction failures focuses on reliability issues under non-adversarial conditions, analyzing how operators incorrectly manage platform resources in normal operations.

2.9 Summary

This chapter presented a systematic study of 412 real-world operator failures across 13 popular Kubernetes operators. Our analysis shows that the fundamental reliability challenge of operators lies in the complexity of their interactions with external entities: the managed system, the cloud platform, co-located operators, and the user interface.

One important finding is that interactions with the managed system are the dominant source of operator failures (42%), due to the lack of well-defined management interfaces of systems. Operator developers must encode complex operation semantics in an ad-hoc manner, resulting in a fragile, error-prone interaction. The remaining interactions present distinct challenges: platform interactions (30%) involve implicit resource requirements and stale state observation, user interface failures (23%) are dominated by missing or incomplete implementations, and co-located operator failures (5%) expose the lack of inter-operator contracts.

Our analysis of existing techniques (Table 7) shows that while tools like Acto, Sieve, Anvil, and platform-level solutions partially address platform, user interface, and co-located operator interaction failures, no existing technique targets operator-system interactions—the largest category. This gap motivates the development of Oat (Chapter 4). More broadly, our findings point to the need for well-defined management interfaces that describe the operation semantics of cloud systems, which remains an open and important research challenge.

Chapter 3 Acto: Automatic End-to-End Testing for Operators

Our empirical study (Chapter 2) establishes that operator failures are dominated by their erroneous interactions with external entities. In practice, operator developers primarily rely on unit tests and integration tests for quality assurance. Unit tests verify individual functions in isolation and cannot check whether an operator reconciles the managed system to its desired state. Integration tests typically run the operator against a mocked or simulated cluster environment, but do not validate end-to-end operation correctness. As we show in a motivating study of 50 open-source Kubernetes operators (Section 3.2), the few manually written end-to-end tests cover only a small fraction of interface properties and test almost, test from the default initial state, and assert on a negligible portion of system state fields.

This chapter presents Acto, fully automatic end-to-end testing technique and tool for cloud system operators and can be readily applied to any types of operators for managing different systems. Unfortunately, existing automated test generation techniques like fuzzing [91] or symbolic execution [17] cannot effectively test operators end to end, since they neither reason about the semantics of operations nor check the system states. In particular, operator bugs do not commonly manifest as crashes, but drive systems into undesired states (Section 3.6.1).

Acto automatically generates end-to-end tests that check three operation correctness requirements: (1) the operator always reconciles the managed system to the desired state; (2) the operator recovers the managed system from error states by rolling back to a previous good state; and (3) the operator is resilient to misoperations by rejecting erroneous inputs from driving the system into error states. To effectively explore the large state space, Acto employs three test strategies: single-operation, operation-sequence, and error-state recovery. It also uses two principled automated oracles to detect both explicit errors and silent state mismatches.

Key results. We implemented Acto for Kubernetes operators. It works in two modes: a blackbox mode (Acto-■) that only requires the operator's interface specification (custom resource definition of Kubernetes operators) and a whitebox mode (Acto-□) that additionally takes the operator's source code for semantic 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 42 have been confirmed and 30 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

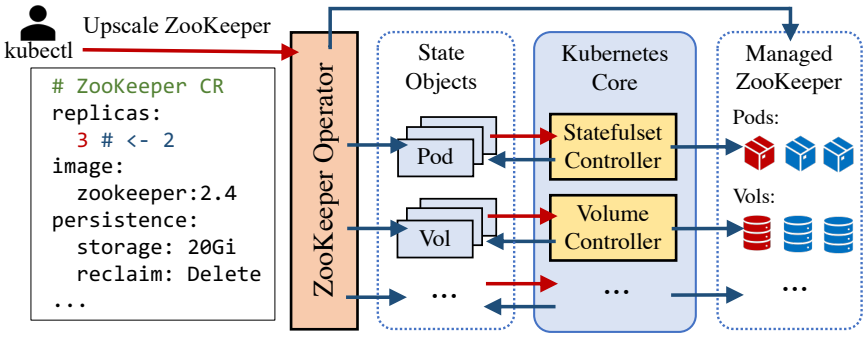


Figure 7: **Scaling up a ZooKeeper system (from 2 to 3 replicas) with a new desired-state declaration (CR).**

security of the managed systems. Lastly, Acto finds many 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 has few false positives: Acto-□ reports no false alarm and Acto-■ has a 0.19% false alarm rate.

This chapter makes four main contributions:

- We present the first fully automatic end-to-end testing technique that checks operation correctness for cloud system operators using a state-centric approach.
- We develop 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.
- Acto is released as an open-source project and is hosted at `https://github.com/xlab-uiuc/acto`, where the sosp-ae branch includes detailed instructions on reproducing the results in this paper.

3.1 Background

Operation programs (i.e., operators) for modern cloud management platforms like Kubernetes [16], Twine [144], and ECS [95] follow a declarative, state-reconciliation design 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. We give a brief overview of the pattern, using Kubernetes [16] as an example.

Declarative operation interface. The declarative interface greatly simplifies the operation complexity on cloud platforms: users specify what state the system should be in, and the operator determines how to reach that state from whatever the current state is. This design is important for reliable cluster management at scale. It allows the operator to continuously compares the current state against the desired state and acts on the divergence, instead of reacting to individual events. This means the operator converges to the same desired state regardless of the starting state, and missed or duplicate events cannot cause the system to drift.

In Kubernetes, operators expose a declarative interface in the form of *custom resources (CRs)* [39]. A CR defines a system resource and its properties that can be modified to manage that resource. A state declaration specifies property values in a CR. Figure 7 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. Management operations are expressed by changing one or more property values in a CR.

Kubernetes operators maintain CR definitions in the OpenAPISchema format [104], 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 [16, 159, 144, 128, 95, 151]. 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 [49]). 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 highly interpretable and extensible [16].

Figure 7 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 stateful-set 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. 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 [162, 108, 54, 99, 103, 13, 11]. We discuss systematic mitigations for misoperation vulnerabilities with developers.

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

3.2 Motivating Study

To understand the kinds of test cases (i.e., tests) that operator developers write and the limitations of their current testing practices, we study 50 open-source 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 and integration tests, which only check individual methods in operator code or test the operator against a mocked environment. Mocks of the managed systems are inherently limited by the accuracy of their assumptions about system behavior. They must encode the management semantics which are under-specified (Section 2.3). End-to-end (e2e) tests provide full fidelity by validating operation correctness against the actual managed system.

While improving the fidelity of mocked environments could catch more bugs than current unit tests, mocks are inherently limited by the accuracy of their assumptions about system behavior — they must encode the very management semantics that are under-specified and cause real failures. End-to-end testing against the actual managed system provides full fidelity.

Typically, an e2e test first causes an operator to carry out an operation, for example, to

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

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 around the managed systems. These four operators contain 7–31 e2e tests; PCN/MongoOp relies only on e2e tests (no unit tests). Table 11 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 8 shows that existing e2e tests change very few properties when testing operation correctness in these four Kubernetes operators that we study. 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 efficiently helps test more operations in multiple configurations.

Operators are long-running processes that 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 8 (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.*

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 9: **Three types of assertions in existing e2e tests.**

Given the enormous state space, developers likely find it tedious to write assertions on many state-object fields. Table 9 shows a breakdown of three kinds of assertions that we observe in existing e2e tests. These tests 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 either only check a small part of the system state or only check the availability of system services.

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 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 11, §3.6). Our results suggest that manually writing end-to-end (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 system operators.

3.3 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 (§3.1) 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 modern cloud management platforms based on state reconciliation 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 (e.g., the custom resource definition of Kubernetes operators). Both inputs are abundant in mature operator projects; they are widely used for operator development and deployment. 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 for root cause analysis, and minimized test code that reproduces detected failures.

3.3.1 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., $S^D \vDash D$. 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 $S^D \vDash D$.

If an operation fails (e.g., due to bugs in operator code), the system enters an error state, $S^e \nvDash D$, i.e., S^e does not satisfy the desired state. When $S^e \nvDash D$, 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 $\breve{D}$, where S^C is the set of all possible valid system states ($S^c \in S^C$), S^E is the set of all possible error states ($S^e \in S^E$), and $\breve{D}$ is the set of all possible declarations of desired state ($D \in \breve{D}$). There can be a large number of values for different properties that constitute the system state. Exhaustive testing could be prohibitively expensive and any practical testing approach can only exercise a part of the state space, i.e., $S \times \breve{D}$.

3.3.2 Test Strategy

Acto systematically explores the state space using the following three test strategies (Figures 8a–c).

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 $S^D \vDash D$. 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 §3.2; it is easy to implement and reason about. The key challenge is how to explore an effective and representative subset of $\breve{D}$.

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 §3.2, 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

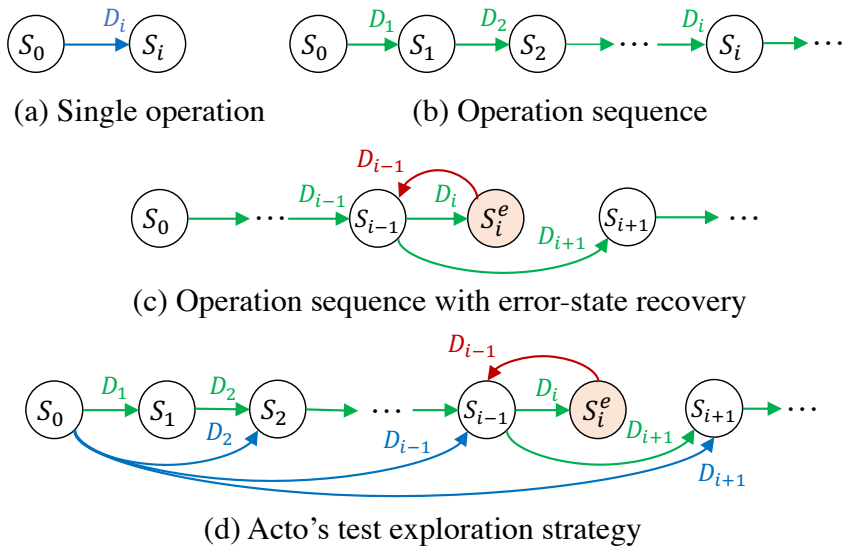


Figure 8: **State transitions of different test strategies.**

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 8b. The result is a sequence of state transitions, $S_0 \xrightarrow{D_1} S_1 \xrightarrow{D_2} ... \xrightarrow{D_i} S_i \xrightarrow{D_{i+1}} ...$; Acto checks whether each $S_i \vDash D_i$, where $i \neq 0$.

Error-state recovery. The operation-sequence strategy does not test whether or not 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 8c. Error states can be reached because of operator bugs that reconcile the system to a state $S^e \nvDash D$, or misoperations—semantic errors in D that escape syntactic validation against the interface specification.

Acto combines these three test exploration strategies (Figures 8a–c) to realize the state transition sequences in one test campaign, as shown in Figure 8d.

3.4 Design

This subsection describes the main components of Acto and how we implement them. These components embody Acto's state-centric testing technique (§3.3); they generate declarations of desired system states, execute test campaigns, and check reconciled states

using automated test oracles.

3.4.1 Realizing State Transitions

During a test campaign (Figure 8d), 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 systematically exercises *all* the properties that are defined in the operation interface. Each new D_{i+1} changes one property in the current state S_i and any other properties that are needed to satisfy predicates on property relationships (§3.4.2.4). Specifically, Acto selects a previously untested property and uses it to declare a new desired state. The end state after one transition, becomes the start state for the next transition (Figure 8b). 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, §3.4.2.2). Table 10 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. This scenario-driven approach allows Acto to focus on a small number of representative states, instead of the very large set of all possible property values. We implement scenarios as plugins that can be extended or customized; users of Acto 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 10). Acto uses misoperations to check if an operator (1) is resilient to operation errors, and (2) can recover from undesired or error states. Acto's oracles (§3.4.3) 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 (see §3.4.2.1). A correct operator should not carry out an erroneous operation or at least should be able to 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 10: **Examples of built-in scenarios of Acto to generate new state declarations and trigger state transitions.** Scenarios are created based on property semantics inferred by Acto and they can be extended or customized.

3.4.2 Generating State Declarations

Acto generates desired-state declarations, $D \in \breve{D}$, that are syntactically valid (§3.4.2.1), resemble real-world scenarios (§3.4.2.2, §3.4.2.3), and satisfy predicates on property relationships (§3.4.2.4). Such desired states improve the effectiveness 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.

3.4.2.1 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 sub-properties. For primitive properties, Acto uses constraints like the type, min/max values (for numeric types), length (for string type), regular-expression patterns, etc.

3.4.2.2 Inferring Property Semantics To exercise different scenarios (§3.4.1), 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

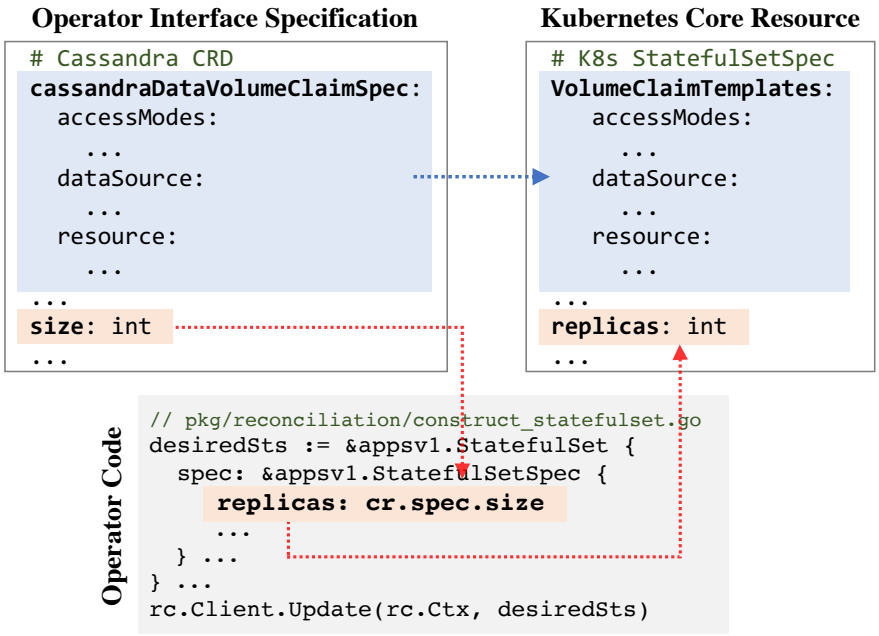


Figure 9: **Semantic analysis maps the properties in the operation interface to the properties of a Kubernetes core resource.**

resource specification. Specifically, all Kubernetes core resource types have unique structures. Figure 9 exemplifies how Acto infers semantics from the property structure: CassOp has a `cassandraDataVolumeClaimSpec` property with the same structure as the `VolumeClaimTemplates` property in Kubernetes' StatefulSet resource. Therefore, 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 9 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 the 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 9.

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 9) and the taints are propagated via data-flow dependencies. The analysis is field sensitive—to track each primitive (sub-)property in the declaration—, inter-procedural and context sensitive.

3.4.2.3 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 generators focus on high-level semantics to exercise different scenarios (Table 10). Each generator creates property values to realize a scenario. 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 (§3.4.2.1). Acto only mutates primitive sub-properties of composite properties. Acto's mutation ensures syntactic validity but does not guarantee representativeness. Mutated values that are not representative help check for vulnerabilities to misoperations. Our manual inspection during Acto evaluation (§3.6) shows that 80+% of mutations are semantically meaningful.

3.4.2.4 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 [162, 27].

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).

3.4.3 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 if $S_i \vDash D_i$ for each state transition $S_{i-1} \xrightarrow{D_i} S_i$. Checking $S_i \vDash D_i$ is challenging. First, S_i and D_i are represented differently: D_i is a specification [39] and S_i is embodied in state objects [156]. Second, satisfiability ($\vDash$) is domain-specific; its semantics may not be obvious. Acto uses two types of oracles to detect implicit-state mismatch:

- *Consistency oracle* (§3.4.3.1). Acto checks whether $S_i \vDash D_i$ from the operator and the management platform (e.g., Kubernetes) views. A buggy operator's view may show $S_i \vDash D_i$ while the management view shows $S_i \nvDash D_i$. Such view inconsistencies likely indicate the presence of bugs.
- *Differential oracle* (§3.4.3.2). This oracle leverages the level-triggering principle [63] 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

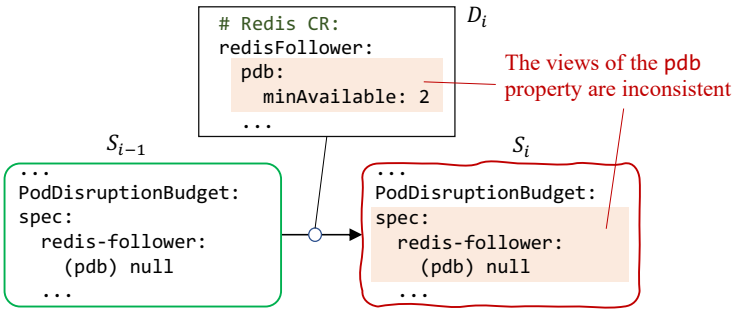


Figure 10: **An OCK/RedisOp bug detected by Acto's consistency oracle [133].** The `PodDisruptionBudget` state object has a `null` `pdb`, inconsistent with the `pdb` declared in D_i.

> 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.

In addition to the automated built-in oracles, Acto also has an interface to allow users to add custom oracles, e.g., domain-specific oracles to check managed systems.

3.4.3.1 Consistency Oracle Some bugs occur if an operator stops reconciliation because the system is in state $S_i \vDash D$ in the operator's view, but $S_i \nvDash D$ 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` subsections 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 the corresponding `spec` fields in the state objects. If a match is found, it indicates that the management platform agrees with the operator. Otherwise, Acto raises an alarm.

Figure 10 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 [132]). 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 [8].

Acto uses property structure analysis (§3.4.2.2) to infer correspondences between fields in the `spec` subsection 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

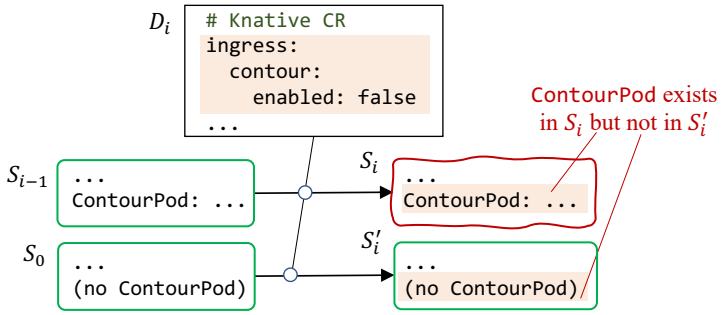


Figure 11: **A KnativeOp bug that is detected by Acto's differential oracle [38].** Contour continues to manage ingress after an operation explicitly disables it.

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 change to matched field values.

3.4.3.2 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 11 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 8d). 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 (§3.2). 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 ports 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 deterministic.

3.4.4 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 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 [41, 48] can also be used. Acto users can suppress alarms by writing annotations.

3.5 Implementation

We implement Acto for Kubernetes operators. Acto-■ has 12,100 lines of Python code. Roughly 9,000 of those lines implement generic test logic (e.g., input generation, test execution, and oracles). Kubernetes-specific semantic inference and value generation take ∼2K lines. If new Kubernetes resources are introduced in the future, we will need to extend Acto to add new value generators for the associated properties (§3.4.2.3). The remaining lines of Acto-■ code implement utilities: environment setup, state analysis, etc. Acto-□ is built on top of Acto-■ using an additional 5,700 lines of Go code for program analysis. We currently support operators written in Go, the most popular language among operators. Acto runs tests on virtualized Kubernetes clusters. It supports three backends, Kind [75], Minikube [97], and K3d [72].

Static analysis in Acto-□. We use `ssa` [110] which provides intra-procedural static single-assignment (SSA) representation. We use `pointer` [109] for alias analysis, which implements the Andersen-style point-to analysis [4].

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.

Operator	System	Dev.	# Stars	LOC	# E2E Tests
CassOp	Cassandra	K8ssandra	148	23.1K	48
CockroachOp	CockroachDB	Official	238	17.4K	21
KnativeOp	Knative	Official	157	16.3K	7
OCK/RedisOp	Redis	OCK	531	2.5K	0
OFC/MongoOp	MongoDB	Official	977	17.1K	62
PCN/MongoOp	MongoDB	Percona	268	15.0K	31
RabbitMQOp	RabbitMQ	Official	669	14.7K	8
SAH/RedisOp	Redis	Spotahome	1303	10.5K	1
TiDBOp	TiDB	Official	1130	132.8K	131
XtraDBOp	XtraDB	Percona	448	15.5K	37
ZooKeeperOp	ZooKeeper	Pravega	332	5.5K	8

Table 11: **The Kubernetes operators that we evaluate.**

Test parallelization. To speed up testing, Acto partitions its operation sequences, $[(S_0, D_1), (S_1, D_2), ..., (S_x, D_{x+1})]$, into multiple tests and runs them in parallel. To run three partitions of this sequence in parallel, Acto creates three tests corresponding to 1) $S_0 \xrightarrow{D_1} S_1 \xrightarrow{D_2}, ..., \xrightarrow{D_i} S_i$, 2) $S_0 \xrightarrow{D_i} S_i \xrightarrow{D_{i+1}}, ..., \xrightarrow{D_n} S_n$, and 3) $S_0 \xrightarrow{D_n} S_n \xrightarrow{D_{n+1}}, ..., \xrightarrow{D_x} S_x$. 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.

3.6 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 11). 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 §3.2.

Our main evaluation results are summarized as follows:

- Acto finds 56 new bugs in eleven operators; 42 bugs in the operators have been confirmed; 30 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%.

3.6.1 Finding New Bugs and Vulnerabilities

Acto finds previously unknown bugs in all evaluated operators, 56 bugs in total (Table 12). We reported all these bugs. So far, 42 were confirmed and 30 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 tests to reproduce all 56 bugs that it detects; developers can add these e2e tests to their regression test suite (§3.4.4). In fact, for six bug fixes, developers added regression tests that perform the same state transition generated by Acto. Our experience tells that the generated e2e tests are invaluable for debugging and validating bug fixes.

Many bugs detected by Acto have severe consequences: managed-system failures, reliability issues, and security issues (Table 13). Estimating the likelihood of encountering each bug "in the field" is hard—the data for such estimation is not publicly available. However, a bug detected by Acto was also encountered by a real user after we reported it [20]. Also, some previously reported bugs are similar to those that Acto detects (e.g., [31]). Note that the evaluated operators are popular open-source projects (GitHub "#Stars" in Table 11), 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 [147], incompatibility between declarations and API-server validation [5], crashes due to Go's generated shared object [33], etc. All these six bugs were confirmed or fixed after we reported them.

Acto also detects 630 misoperation vulnerabilities (§3.6.1.2). Each vulnerability corresponds to a unique misoperation that drives the managed system into an error state.

3.6.1.1 Bugs Detected by Acto Acto detects bugs that violate the first two operation correctness requirements: (1) driving managed systems into undesired or error states, or (2) 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 latent and hard to observe (e.g., security vulnerabilities).

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 12: **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.

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 10 and 11. These bugs show the importance of modeling operations as state transitions and testing different state transitions to the same declared states (§3.3.1).

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. In another example [149], 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 [148]. The crash brought down the webhook service [46] 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. 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

Consequence	Example	# Bugs
System failure	MongoDB is down and cannot recover [98]	5
Reliability issue	Redis is not protected by disruption budget [133]	15
Security issue	CockroachDB uses outdated secrets [157]	2
Resource issue	Redis runs with no resource guarantee [125]	9
Operation outage	CockroachOp crashes and cannot recover [148]	18
Misconfiguration	Ingress controller cannot be disabled [38]	15

Table 13: **Consequences of the 56 detected bugs in Table 12.** One bug can have multiple consequences.

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.

3.6.1.2 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 10). *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 [162, 108, 54, 99, 103, 13, 11].

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 [142]. 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 §3.6.1.1.

3.6.1.3 Effectiveness of Different Oracles Acto's consistency and differential oracles catch 43 of the 56 bugs (Table 14). The consistency oracle detects 23 bugs by matching and comparing properties in state declarations to the `spec` subsections in state objects (§3.4.3.1). The differential oracle catches ten more bugs that are triggered during

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., exception, error code)	14 (25.00%)

Table 14: **Breakdown of the number of bugs detected by the oracles.** Same bug can be detected by multiple oracles.

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). Compared with state-based assertions in existing tests that only cover 0.24%–10.9% of state-object fields (Table 9), Acto's oracles systematically check *all* related fields. For example, the differential oracle compares all state-object fields that are deterministic (71.4%–80.5% of all fields across evaluated operators) through different transitions to the same end state.

3.6.1.4 Coverage Acto achieves 100% property coverage for every operator—Acto generates at least one operation for each property (§3.4.1). Acto's effectiveness over manually-written tests (§3.2) comes from its ability to cover more properties and their values, and more transitions from different states (including error states). In 38 of 56 detected bugs, the related property is uncovered 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 [79] are correctly added to pods, but Acto detects a bug [155] that can only be triggered when pod labels are deleted.

3.6.1.5 Bug Fixes We reported all 56 bugs that Acto finds to the developers of the respective operators; 42 have been confirmed and 30 of those have been fixed. Developers typically fix these reported bugs by improving reconciliation logic for the bug-triggering transitions generated by Acto, and adding validation logic before reconciling on each state declaration to prevent error conditions. Fixing bugs in failure-recovery logic usually requires more effort, because it needs domain knowledge to differentiate permanent error states from transient unstable states. For example, the TiDBOp bug [150] has been confirmed, but the developers cannot easily fix it because the operator cannot reliably detect liveness violations—the pod migration will never succeed in the future—by observing the current state.

3.6.1.6 Tradeoffs between Acto-■ and Acto-□. We expect Acto-□ to be more beneficial than Acto-■ for operators that heavily use primitive-typed properties or do not follow naming conventions for property dependencies. In the evaluated operators, most properties have composite type with clear structure features and they follow naming conventions. Hence, the benefit of Acto-□ over Acto-■ is small in our evaluation. Note that Acto-□ is language specific—it currently only supports operators written in Go. Acto-■ is language agnostic and can apply to operators written in languages other than Go, and proprietary, close-sourced operators.

3.6.2 Test Efficiency

Table 15 shows machine hours Acto-□'s test campaigns take per operator and the number of operations in each test campaign ("#Ops"). The longest campaign (XtraDBOp) had 1,950 operations. Acto stops generating operations when a campaign covers all properties and corresponding scenarios.

All experiments are run on Cloudlab [45] 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 (§3.5). 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 (§3.4.2.2) 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 (§3.4.2.3).

3.6.3 False Positives

Acto's 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

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 15: **Acto-□ test campaign time per operator.**

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 [115, 152], but it is now beyond the scope of testing.

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 §3.4.2.4, 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 (§3.4.2.4).

3.7 Implications and Discussion

We reflect on our experience on finding root causes of detected bugs and vulnerabilities, and discuss 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 (§3.3.1). Declarative programming [142] may make operator testing less error-prone.

As for testing, Acto uses property coverage to drive state transitions in the test campaigns (§3.4.1). The rationale is to achieve high coverage of desired states, as state transitions are triggered by changing property values via the operation interface. Traditional coverage metrics like code coverage are insufficient because they are not concerned with system states: tests that are adequate for the code in one state may not be adequate in a different state. Code coverage may not help test all properties either, e.g., an operator that is missing code to handle transition-triggering property changes could have high code coverage. Acto can find bugs due to missing code if the end state does not match the desired state.

Reducing risks. Operations can pose new reliability risks to managed systems—what happens if an operation fails during execution? An operation can span a series of procedures. For example, we observe that existing Kubernetes operators commonly implement reconfiguration operations in two stages: (1) stopping the current running node (with the old configuration); then (2) starting a new node (with new configuration). In such implementations, failure in either stage is risky. First, such a failure could leave the operator in intermediate states which are nontrivial to recover from [140, 137]. Acto's results show that recovery failures are common (§3.6.1.1). Second, in such implementations, the first step can open a small window of downtime (e.g., due to stopping the current node). That downtime would be magnified if a new node fails to start. So, it is safer to turn down 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 [167, 89]. For example, a ZooKeeper cluster cannot have two leaders at the same time, to avoid a split brain. So, a reconfiguration operation must first stop the old leader node before starting the new one to avoid a split brain. System support for speculative execution or emulation can help.

Closing the knowledge gaps. Operations must also respect the constraints of the managed system. Otherwise, an operation can harm the managed system. The TiDBOp bug described in §3.6.1.1 is one example. Also, many vulnerabilities to misoperations that Acto detects are rooted in the essential cross-system interaction challenge [145]—it is hard for an operator to comprehensively check a requested operation's *semantic* validity if the semantics are not defined inside the operator code but in the managed system or the underlying management framework (e.g., Kubernetes). One potential solution is to

replicate the validity checks of the relevant components in the operator. (Prior work showed the promise of automatically extracting configuration checks [**xu:16**].) In essence, the knowledge 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.

3.8 Summary

This chapter presented Acto, the first fully automatic end-to-end testing tool for cloud system operators. Acto takes a state-centric approach, modeling operations as state transitions and systematically exploring the space of desired states, starting states, and error states. Its two automated oracles, consistency oracle and differential oracle, detect both explicit errors and silent state mismatches.

Our evaluation on eleven popular Kubernetes operators demonstrates that Acto is effective, efficient, and practical. Acto found 56 previously unknown operator bugs and six bugs in Kubernetes and the Go runtime, with no false alarms in whitebox mode and few false alarms in blackbox mode. Many of these bugs have severe consequences, including system failures, security vulnerabilities, and unrecoverable error states. Acto's test campaigns complete within a nightly run, and developers have already used its generated test cases to fix bugs and build regression suites.

However, Acto's broad applicability comes with an inherent limitation: its system-agnostic design cannot exercise system-specific operations, which our empirical study identifies as the dominant source of operator failures. The next chapter presents Oat, which builds on top of Acto's end-to-end testing framework to target this critical gap.

Chapter 4 OAT: Automatic Testing of Operator-system Interaction

Our empirical study (Chapter 2) identifies the interaction with the managed system as the largest source of operator failures, accounting for 42% of all interaction-related failures. None of the existing solutions address this gap, including Acto.

Acto (Chapter 3) provides the first automatic end-to-end testing framework for operators, but its system-agnostic design limits its ability to detect this dominant class of bugs. Specifically, Acto has two limitations in this regard. First, Acto only mutates generic system resource properties and cannot generate operations that exercise system-specific behavior, such as reconfiguring a database's replication method. Second, Acto does not inject faults against the managed system, so it cannot test critical operator responsibilities like failover, recovery, and fault tolerance during operations.

This chapter presents OAT, a testing tool that builds on Acto's end-to-end framework to specifically target operator-system interaction bugs. OAT advances Acto in three directions. First, OAT generates semantically meaningful values for system-specific properties to trigger diverse state transitions, using two complementary techniques: mining developer-written tests and usage examples, and querying a large language model for properties. Second, OAT introduces fault injection against the managed system, including container crashes, network delays, and network partitions, to test failover, recovery, and fault tolerance during operations. Third, OAT provides enhanced oracles, including user-provided state monitors that expose system-internal state and system workloads that continuously monitor availability throughout state transitions.

We evaluate OAT on six popular, mature Kubernetes operators managing critical cloud systems. OAT detected 86 new bugs across all six operators with no false alarms, finding bugs in every tested operator and in all studied failure patterns. Notably, 62.8% of the bugs manifest through patterns that do not appear in our empirical study, showing the challenges of implementing reliable operators and the breadth of bug spectrum.

4.1 Study

Despite the prevalence and the importance of the operator-system interaction failures, no existing technique addresses this challenge. Automatic testing tools like Acto and Sieve are system-agnostic. Verification frameworks like Anvil lack models for system management interface to reason about the system interactions.

The importance of operator reliability demands *automatic* testing techniques for detecting defects in operator-system interactions and preventing interaction failures. Such testing must advance existing techniques in two aspects:

First, the tool must systematically exercise the operator-system interactions. Specifically, it must generate operation commands that can mutate system-specific properties through the declarative user interface.

Finding 14: *70.3% (64/91) of target failures must be triggered by operation commands that change the desired states through the declarative user interface; among them, 71.9% (46/64) need to specify system-specific properties.*

Existing tools for operator and controller testing [137, 57] are *system agnostic*—they only mutate system resources properties, but skip system-specific properties.

Second, the new tool must inject faults against systems.

Finding 15: *40.7% (37/91) of target failures need to be triggered by external faults that occur on the systems.*

No existing testing tool (Table 1) consider faults that happen to the managed systems. They only reason about faults on the cloud platform or the operators.

Driven by the above two findings, we develop Oat for testing the interaction between the operators and their managed systems (Chapter 4).

4.2 Design

Driven by the discussion in Section 4.1, we develop Oat, a simple tool for testing interactions between operators and their managed cloud systems. Oat targets bugs manifested via different patterns of operator-system interaction failures (Table 4). Those bugs cannot be found by existing tools as they are all system agnostic (see Section 2.1.2).

Oat follows the end-to-end paradigm of operator (or controller) testing that exercises the target operator together with the system [57, 168, 137]. It organizes tests into *test campaigns*. In each campaign, Oat keeps generating new operation commands and/or injecting faults which drive the operator to continuously reconcile the system, until a bug is caught or a time budget is reached. Operation commands change desired system states, while faults change current states.

During a test campaign, Oat monitors the system with two key principles: (1) normal operation commands should not affect the availability of the systems in production, and (2) operators should handle external events correctly; Oat only injects transient faults (a node crash, a network delay, or a connection timeout) that are common in real-world deployment and are expected to be handled by the operators.

4.2.1 Testing State Transitions

OAT models an operator's input as a pair of a system's existing state and its desired state [57]. Operators monitor and reconcile any divergence between the existing and desired state. When a mismatch occurs, the operator initiates a state transition to drive the system from its existing state to a new state, to match the desired state.

Given an operator, OAT automatically generates end-to-end tests which effectively explore different types of state transitions. Each test starts in a consistent state, where the existing state matches the desired state. The test then triggers a state transition by creating a divergence between the existing and desired states. OAT takes an empirical approach to explore the system's state space during its test campaign. Its testing policy is driven by our analysis (see Table 16). OAT explores the space of possible state transitions by employing three strategies: (1) declaring new desired states, (2) perturbing the existing states, and (3) both, described as follows:

- To test normal system operations (e.g., reconfiguration), OAT triggers state transitions by declaring new desired states. Based on Finding 11 (§4.1), the majority of system interaction failures require changing the desired states. Among these, a majority require changing system-specific properties. The key challenge is to effectively synthesize system-specific properties for generating desired state declarations.
- To test operations which are only triggered when the system needs to handle a faulty state (e.g., failover and recovery operations), OAT triggers state transitions by perturbing the existing system state via fault injection. OAT checks if the system state is reconciled back to the desired state after the transient faults are removed.
- To test operators' ability to handle errors during operations, OAT both declares new desired states and perturbs the existing states in a test. We expect operators to be able to tolerate transient faults happening in the middle of the state transition (e.g., handle operation errors), and eventually drive the system to the declared desired state.

OAT workflow. OAT tests operators with the strategies mentioned above in three phases. First, it generates valid desired state declarations with semantically meaningful values for system-specific properties (§4.2.2). Second, it generates end-to-end tests by systematically combining these declarations with fault injection (§4.2.3). Finally, it executes each test in local Kubernetes clusters and validates their outcomes using automatic oracles (§4.2.4). OAT reports test failures along with the state transition for reproduction.

Patterns		Operation Commands	Faults
Seman-tics	Configuration	Update app configuration	N/A
	Ordering	Update app-specific properties	Delay operations
	State	Update app-specific properties	Crash app container
	Environment	Update app security context	N/A
State observability		Update app-specific properties	N/A
Error handling		Update app-specific properties	Operation timeout
Incompatibility		Update app image versions	N/A

Table 16: **OAT's test policies for different failure patterns.**

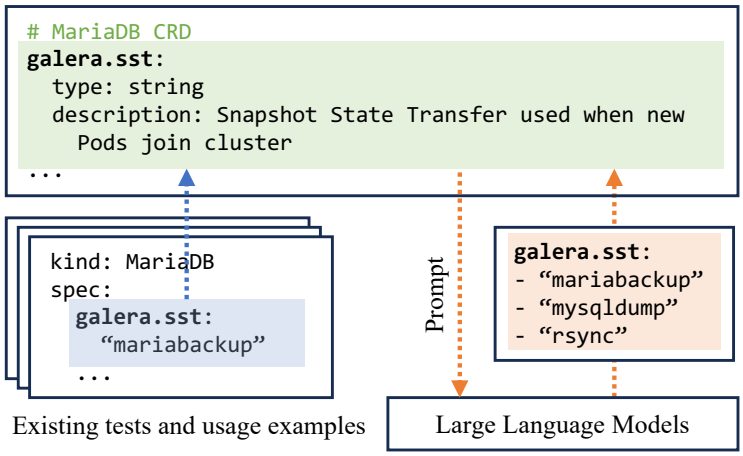


Figure 12: **Synthesizing values for the `galera.sst` property using existing tests and usage examples, and LLMs.**

4.2.2 Generating State Declarations

OAT generates desired state declarations that effectively exercise operator-system interactions through diverse state transitions. Typically, system-specific properties accept only a narrow, specific set of values. The key challenge is to automatically synthesize a wide array of such semantically meaningful values. For example, MariaDB's state snapshot transfer method accepts few values, including "`mariabackup`", "`mysqldump`", and "`rsync`." Testing with different values exercises the operator's ability to reconfigure this property correctly across state transitions. Randomly fuzzing property values is ineffective: it generates mostly invalid values that are rejected by the operator and system, and does not explore valid state transitions.

OAT synthesizes system-specific property values using (1) developer-written values or (2) large language models.

Developer-written values. Mature operator projects already contain unit tests and

```
Context:
You are a expert of the mariadb-operator of the
Kubernetes ecosystem. You are tasked with providing
values for properties of the MariaDB CRD.

Prompt:
Here is the property that need values:
- name: spec.galera.sst
- description: SST is the Snapshot State Transfer
    used when new Pods join the cluster.
- type: string

The property has a datatype and description
provided above, please make sure the generated
value satisfies the datatype and description.

Provide two values for the property and please
follow the YAML format. Directly give me the YAML
object without any other message, for example:
---
spec:
  galera:
    sst: value
---
spec:
  galera:
    sst: value
```

Figure 13: **An example LLM prompt for synthesizing values for the `galera.sst` property for MariaDBOp.**

usage examples that instantiate desired state declarations with meaningful property values. In Kubernetes, the desired states are described by properties of Custom Resources (CRs) [39] and are structured according to the Custom Resource Definition (CRD). Oat thus can automatically parse the desired state declarations based on the CRD to extract property values. It then combines values across different examples to construct new desired states. This process ensures that synthesized declarations are realistic, while also exploring new operations not covered in existing tests. For example, in Figure 12, for the MariaDBOp, Oat identifies "`mariabackup`" as a valid value for the `galera.sst` property by mining existing CR examples. To trigger state transitions, Oat requires at least two values per property.

Large Language Models (LLMs). Not all properties are covered by existing unit tests and usage examples. When no example is found, Oat queries a LLM with a structured prompt. The prompt includes the property's definition, natural language description, and type information from the CRD, and asks the model to produce candidate values in YAML format. In Figure 12, the LLM generates additional valid values for `galera.sst`, such as

"`mysqldump`" and "`rsync`,", which extend coverage beyond what appears in developer tests. Figure 13 shows the LLM prompt used for the property. Although LLM-generated values may occasionally be invalid, Oat filters these cases during testing: invalid values are rejected by the operator or system and discarded automatically.

Tradeoffs. Developer-written values are semantically valid and exercise realistic scenarios, yet are not always available (82% property coverage in our evaluation). LLMs can synthesize values for uncovered properties, but may generate invalid ones due to hallucination. We find 78% of LLM-generated values to be valid (§4.4.3). Oat prioritizes developer-written values if available, and uses LLM-generated values otherwise.

4.2.3 Perturbing Existing System State

System-operator interaction failures arise not only from transitions starting from healthy states but also from error states caused by external events (Finding 12, §4.1). Oat injects system faults to perturb the existing state, to test whether operators can: (1) recover the managed system back to desired states from error states, and (2) tolerate faults that occur during state transitions and eventually drive the managed system to the desired state. Oat injects three fault types that commonly occur in production environments and expose operator bugs: container crashes, network delays, and network partitions.

To test the correctness of the failover and recovery operations of operators, Oat injects transient system container crashes to drive the existing state to an error state. Oat then removes the fault, and checks that the operator successfully reconciles the system back to the desired state.

To test if operators can tolerate faults occurring during operations, Oat combines fault injection with new desired states. Specifically, Oat introduces a persistent fault to the system and then declares a new desired state to trigger reconciliation. With the persistent fault, the operator cannot successfully complete reconciliation. Oat then removes the fault and checks if the system state eventually matches the new desired state. Oat uses network delays to test operation ordering dependencies (see §2.3.1.2) and network partitions to test operation error handling (see §2.3.4).

4.2.4 Test Oracles

Oat employs automatic oracles to check whether the system state after the state transition matches the desired state in each test. Similar to previous works [57, 137], Oat checks for explicit or implicit state mismatches by observing the system state through the Kubernetes API objects. Kubernetes APIs provide limited visibility into system internal state and cannot detect silent failures where systems are running in incorrect states (e.g., running

with old configuration values). Additionally, operators are required to maintain high availability during state transitions, and checking system state only after the transition completes cannot detect transient availability violations. To address these limitations, Oat can leverage optional user-provided utilities that expose system internal state and monitor availability throughout transitions, significantly enhancing bug detection capability.

State monitors. Oat benefits from system state monitors which can check systems' internal state against the desired state. Specifically, we found that configuration state monitors are easy to implement, but effective at detecting configuration-related operation semantic violations. For example, a MariaDB configuration state monitor can be implemented by (1) getting the current configuration from MariaDB by running a `SHOW VARIABLES` command, (2) parsing the MariaDB configuration into key-value pairs, and (3) comparing them with the declared desired configuration.

System workload. Operators are required to maintain high availability for distributed systems during normal operations, e.g., through careful rolling upgrades and leader re-election before decommissioning a system instance. Such failures cannot be detected by checking the system state after the operation finishes; instead, it requires continuously monitoring the system availability during the operation. Oat benefits from user-provided system workloads (e.g., periodic read and write requests) and uses them to monitor the system availability by checking the success rate of system requests during state transitions.

4.3 Implementation

We implemented Oat for Kubernetes operators in approximately 1,200 lines of Python code, on top of the Acto framework [57, 56]. We reuse helpful utilities from Acto, such as setting up Kubernetes clusters and test parallelization. As Acto does not consider faults, we implemented new fault injection logic for Oat in about 1,000 lines of code. Specifically, Oat uses the ChaosMesh [23] to inject faults into the systems. Value synthesis, including collecting developer-written values and prompts for LLM (GPT-4o), takes about 200 lines.

Oat exposes a simple programming interface for custom test oracles. Users implement custom oracles as Python functions that take a runtime context as the argument, allowing them to query the current system state and return the test result. Oat loads user-provided functions and invokes them during each test to validate outcomes. The user-provided state monitors and system (§4.2.4) workloads are also provided through this interface.

4.4 Evaluation

We apply OAT to six popular, mature Kubernetes operators which manage critical cloud systems (see Table 17). We select five operators from our study (Table 2) that cover different types of systems with different management requirements. We would like to check whether bugs with similar patterns still exist in the latest versions of these operators. We also select a new operator MariaDBOp to check if our work can generalize. We test the latest versions of these six operators (the version is hyperlinked in Table 17).

For each operator, we provide OAT with a state monitor for system configuration and a system workload which measures system availability (§4.2.4), implemented in 88–208 lines of Python code. In our experience, porting a new operator takes less than eight developer-hours.

All tests are run on CloudLab Clemson c6420 machines with 2 Intel Xeon Gold 6142 CPUs (16 cores) and 376 GB of memory, with Ubuntu 22.04 LTS. OAT generates 339–2480 unique operation commands and takes 6.2–63.2 machine hours to finish the test campaigns for each operator.

4.4.1 Results and Experience

Finding 16: *Defects in operator-system interactions are still prevalent, which are significant threats to operator reliability. OAT found 86 new bugs in the six evaluated operators; 53 were confirmed and 28 were fixed.*

Table 17 presents the bugs found by OAT of different patterns in each operator. OAT detected 86 *new* bugs and reported no false alarm (see §4.4.3). OAT found bugs in every tested operator and bugs in all studied patterns. The result shows that operator reliability is a significant concern—existing software engineering practices used by the operator projects cannot effectively prevent defects in operators in terms of their interactions with the cloud systems. Note that all the studied operator projects have extensive unit and integration tests.

The failure patterns of detected bugs match our expected distribution (see Table 4). Violations of the system's management operation semantics are the largest category (46 out of 86). Among them, 33 bugs violate the semantics of system configuration; 6 bugs violate preconditions of operations; 2 bugs set up incorrect execution environment; 5 bugs violate order dependencies of multiple operations.

Finding 17: *Existing documents on system management are too vague to follow and miss important management operation semantics.*

Operator	Operation Semantics	State Observ.	Version Compat.	Error Handling	Internal	Total
CassOp	7	0	1	0	2	10
KafkaOp	2	1	0	0	0	3
MariaDBOp	9	1	0	1	16	27
MinIOOp	1	0	0	0	1	2
MongoOp	18	2	1	3	2	26
TiDBOp	9	2	1	0	6	18
Total	46	6	3	4	27	86

Table 17: **New bugs found by Oat in evaluated operators.**

For 13 (out of 46) operation semantic violations found by Oat, no document describes the semantics. In CassOp-695, the requirements for changing `num_tokens` on an existing Cassandra cluster are not specified in the official Cassandra document, but only exist in online blog posts [64] or experience reports [21]. The CassOp did not implement the semantic requirements for changing `num_tokens`, causing the Cassandra cluster to crash after the reconfiguration.

The rest 33 operation semantic violations have documents, but were incorrectly implemented by the operators. We found that some of the operation semantics are too vague to rigorously follow. In MariaDBOp-1226, the MariaDB document specifies the precondition for restarting a node as "*transfer all client connections from the node you are about to upgrade to the other nodes [158]*" without specifying the concrete operations needed to transfer the client connections. The more concrete precondition is to perform primary stepdown before restarting the primary. Some operation semantics are scattered around the document which are hard to find. For example, one of the preconditions for the MariaDB recovery operation is specified in the known issue subsection [93].

Finding 18: *Oat exposed diverse patterns of code bugs at the operation-system boundary. In particular, 62.8% of the bugs found manifest through code-level patterns that do not appear in our study, showing the challenges of implementing reliable operators and the breadth of bug spectrum.*

Oat tests operators-system interactions without assuming particular code idioms. Therefore, it can find diverse bug patterns that manifest through the interaction failures. Among the 86 new bugs Oat detected, only 37.2% (32) match the code-level patterns previously catalogued in our study (§2.3). The remaining 62.8% highlight that implementing operators reliably cannot be assured by targeting known faulty patterns alone. Many failures arise because the operator-system interface is under-specified and

embeds implicit semantics. We summarize the new bug patterns uncovered by Oat below:

- *Silent configuration overruling.* Configuration updates are silently overruled by the operator due to buggy logic when merging updated with existing values. In MongoOp-1335, MongoOp hardcoded the `tlsMode` argument which overwrite any changes of the `net.tls.mode` parameter.
- *Configuration-operation dependency.* Some configuration changes require operations beyond updating parameter values. For example, updating `directoryperdb` for MongoDB requires creating a backup, stopping the MongoDB instance, updating the `directoryperdb` value, restarting the MongoDB instance, and restoring from the backup. In MongoOp-1241, MongoOp changes `directoryperdb` without these operations, causing MongoDB to crash.
- *Brittle observability.* Operators used probes that only apply to specific configuration. In MariaDBOp-1096, MariaDBOp uses a prepared query statement as the liveness probe. This query statement is broken when MariaDB is configured with `max_prepared_stmt_count=1`, causing MariaDBOp to keep restarting healthy MariaDB instances.
- *Version incompatibility between system components.* In MongoOp-1157, downgrading the MongoDB cluster from v7.0.8 to v6.0.15 causes the MongoOp to stuck in an intermediate state where mongos instances are running in v7.0.8, and mongod is running in v6.0.15. The incompatibility causes the mongos instances to become unhealthy, while MongoOp waits for them to become ready.
- *Internal bugs.* 27 bugs are triggered by operator-system interactions but stem from internal issues (Table 17). Oat exposed bugs that result in inconsistencies between the system interface and its implementation. In MDEV-35754, MariaDB's configuration interface specifies 512 MB max for `transaction_prealloc_size` but code allows only 128 MB. Oat also exposed operator's internal bugs (e.g., nil pointer dereference [19]). This further shows the challenges of hardening already under-specified interfaces.

Finding 19: *43.0% (37/86) of the bugs require system-specific state monitors and workloads to capture.*

These bugs cannot be captured by regular oracles that check crashing behavior, error logs, or state objects (used by prior work [57]). Instead, they are manifested through inconsistencies between system configuration states and the ConfigMap object (33 bugs) and transient system unavailability (4 bugs). In MongoOp-1334, MongoOp uses TCP connection success as the readiness probe for MongoDB. During a rolling upgrade, MongoOp restarts each MongoDB instance only after confirming the previous instance to

be fully ready to ensure the majority quorum. However, MongoDB may successfully establish TCP connection but in the booting phase.

4.4.2 Efficiency and Cost

Table 18 shows the number of tests and machine hours Oat takes to test each operator. All experiments run on CloudLab Clemson c6420 machines equipped with two 16-core Intel Xeon Gold 6142 CPUs and 376 GB of memory running Ubuntu 22.04 LTS. Oat generates 339–2,480 tests across the tested operators, with 24–259 faults injected. It takes 6.2–63.2 machine hours to run these tests across the operators.

The cost of using GPT-4o to generate values for system-specific properties is low (Table 18). Oat consumed 11,761 tokens (including prompt, input, and output tokens) on average for each operator. Currently, the monetary cost of generating values using GPT-4o API is about 0.005 USD per operator.

4.4.3 False Positives

Oat reports no false alarms. Most operation commands generated by Oat declare valid desired system states. A reliable operator must reconcile the cloud system to these valid states. If the operator fails to reconcile or crashes, then Oat catches a true failure. We then inspect each failure and identify the underlying root causes in the operator programs.

If the generated operation commands declare an invalid system state (which is known as *misoperations*), we expect a reliable operator not to crash or drive the cloud system to an error state (e.g., system outages or partial failures). In other words, we expect reliable operators to prevent misoperations from failing their managed operations. If a misoperation fails the operator or its managed system, Oat detects a *misoperation vulnerability* [57], as discussed in §4.4.1. Note that misoperation vulnerabilities are considered serious reliability threats [**xu:16**, 57, 162], as they are commonly introduced by human mistakes or AI hallucinations (if AI is used to interfere with these operators [130, 28]).

In summary, every alarm reported by Oat is either an operator bug or a misconfiguration vulnerability. Oat reported 1094 alarms in total for the six evaluated operators. 384 alarms are caused by 86 bugs in operators and 710 alarms are caused by the 396 misoperation vulnerabilities. These misconfiguration vulnerabilities are mainly caused by LLM-generated property values and configuration parameter values. GPT-4o generated semantically meaningful values for 78% of system-specific properties and system configuration.

Operator	# Prop.	# Config	# Tests	# Faults	Time (hrs)
CassOp	8	229	1,846	70	63.2
KafkaOp	92	233	2,164	432	37.2
MariaDBOp	35	259	2,480	212	56.7
MinIOOp	19	24	339	122	6.2
MongoOp	53	93	1,585	326	57.5
TiDBOp	76	126	1,468	544	51.2

Table 18: **Detailed information on the tests run by Oat.** "Prop." refers to the system-specific properties; "Config" refers to the unique system configurations Oat generated. "Test" refers to tests run by Oat, each of which realizes a failure pattern in Table 16.

4.5 Limitations

Like other testing tools, Oat is neither sound nor complete. Oat cannot guarantee complete coverage of operator-system interaction failures due to several fundamental limitations in its exploration strategy.

Oat's state exploration strategy may not explore all system interactions, leading to false negatives. It triggers the operators to interact with the system indirectly by exercising the system-specific properties with few representative valid values. However, some interactions may only get triggered with specific values. Additionally, Oat relies on the developer-written values and LLMs to synthesize values, which may lead to invalid values thus ineffective tests.

Furthermore, Oat does not fully explore the fault space. It injects generic types of faults at random timing during each test. Thus, Oat may miss system interaction failures which requires specific timing or specific type of faults.

Oat's automatic oracles rely on user-written state monitors to observe the system internal state. Without user-provided monitors, Oat cannot detect silent failures where systems appear healthy but in an undesired state. Future work may explore integrating Oat with automatically generated monitors [86, 87, 88] for detecting such silent failures.

4.6 Summary

This chapter presented Oat, a testing tool that targets the most prevalent yet previously unaddressed pattern of operator bugs: erroneous operator-system interactions. By building on top of Acto's end-to-end testing framework and extending it with system-specific value synthesis, fault injection against managed systems, and enhanced oracles, Oat addresses the critical gap identified by our empirical study.

Our evaluation demonstrates that operator-system interaction defects remain pervasive even in mature, well-tested operator projects. Oat found 86 new bugs across six popular

operators with no false alarms. The detected bugs span all failure patterns identified in our study—operation semantic violations, state observability gaps, version incompatibility, and error handling failures—confirming that these patterns are not historical artifacts but ongoing threats. The finding that 62.8% of bugs exhibit previously uncatalogued code-level patterns underscores that reliable operator-system interaction cannot be achieved by targeting known bug patterns alone, and that automated testing tools like Oat are essential for uncovering the long tail of interaction defects.

Our experience also reinforces the broader argument of this dissertation: the root cause of operator-system interaction failures is the absence of well-defined management interfaces. Oat's bug-finding effectiveness comes precisely from exercising the under-specified boundary between operators and systems—a boundary that, as our results show, remains a significant source of reliability risk in production cloud systems.

Chapter 5 Toward Formal Management Interfaces for Cloud Systems

An important finding throughout this dissertation is that operator-system interaction failures, the dominant root cause of operator failures, stem from the absence of well-defined system management interfaces. Testing tools like Acto and Oat can detect these bugs, but cannot guarantee their absence. This chapter discusses a more fundamental approach: formalizing cloud system management interfaces and using them to enable both model checking and formal verification of operator correctness.

We envision a management interface model as a state machine that captures the management operation semantics of a cloud system. The model defines the states a system can be in (e.g., a PostgreSQL node being in a bootstrapping, ready, or failed state), actions representing management APIs (e.g., failover), and external events such as faults. Transitions between states are guarded by conditions on the system's current state. For example, a successful failover transition may only be taken when the target node is ready, has up-to-date logs, and holds the `SyncStandby` role. The model also defines bad states that are reachable when these guards are violated.

Such management interface models enable two key use cases. First, model checking can verify existing operator correctness by composing an abstract operator model (extracted from the operator's implementation) with the system management interface model and exhaustively exploring all interleavings of operator actions, system state changes, and external events. This would detect operation semantic violations, ordering errors, and precondition violations without relying on sampled test executions.

Second, the management interface model can serve as a trusted specification against which the operator's actual implementation is proven correct. Existing verification frameworks like Anvil relies on trusted specification of the system APIs which are overly simplified—a bug was found in an Anvil-verified ZooKeeper operator due to an incomplete API specification [139]. A standardized, versioned management interface model would provide a reusable specification that multiple operators for the same system can share, and an operator verified against such a model would carry a stronger correctness guarantee than what testing or model checking alone can provide.

Realizing this vision requires addressing several challenges: constructing accurate models for systems with large management surfaces, evolving models alongside system versions to prevent the incompatibility issues identified in our study (Section 2.3.3), and validating that models faithfully capture real system behavior. Despite these challenges, formalizing management interfaces represents a promising next step beyond the testing tools developed in this dissertation: from detecting bugs to preventing them by design.

Chapter 6 Conclusion

This dissertation addresses the reliability of cloud system operators—the management programs that continuously reconcile cloud systems to desired states. As operators become the dominant paradigm for cloud system management, their correctness has unprecedented impact: a buggy operator can directly and continuously damage bug-free systems in production. This dissertation enhances operator reliability through an empirical understanding of how operators fail and practical testing tools that automatically detect serious bugs in them.

Our empirical study of 412 real-world operator failures reveals that the fundamental challenge of operator correctness lies in the multifold complexity of their interactions with managed systems, the cloud platform, co-located operators, and the user interface. One important finding is that interactions with managed systems are the dominant source of failures (42%), due to the lack of well-defined management interfaces between operators and the systems they manage. This finding reframes operator reliability as fundamentally an interface problem: operators fail not because their internal logic is unusually buggy, but because the semantics of the operations they must perform are under-specified, undocumented, and scattered across system code, configuration files, and tribal knowledge.

Guided by this understanding, we built two testing tools. Acto is the first fully automatic end-to-end testing tool for cloud system operators. It found 56 previously unknown bugs across eleven popular operators, with no false alarms in whitebox mode. Acto demonstrates that automatic end-to-end testing of operators is both viable and effective, and that the declarative, state-reconciliation paradigm of modern operators creates opportunities for principled test generation and oracle design. Oat builds on Acto's framework to target the operator-system interaction. By generating semantically meaningful values for system-specific properties, injecting faults against managed systems, and providing enhanced oracles that expose system-internal state, Oat detected 86 new bugs across six mature operators with no false alarms. The finding that 62.8% of these bugs exhibit previously uncatalogued code-level patterns confirms that this class of defects cannot be addressed by targeting known bug patterns alone.

Taken together, these contributions validate the thesis that a deep empirical understanding of operator failures enables the design of practical testing tools that automatically detect serious, previously unknown bugs in cloud system operators. The 142 new bugs found by Acto and Oat across 14 popular operators demonstrate that operator reliability remains an underaddressed concern, even in mature, well-tested projects.

References

[1] Aditya Akella and Ratul Mahajan. "A Call to Arms for Management Plane Analytics". In: *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets-XIII)*. 2014.

[2] *AllReplicasReady outputs True although no pod created.* `https://github.com/rabbitmq/cluster-operator/issues/261`. 2020.

[3] *Amazon Elastic Kubernetes Service.* `https://aws.amazon.com/eks/`. 2026.

[4] Lars Ole Andersen. "Program Analysis and Specialization for the C Programming Language". PhD thesis. DIKU, University of Copenhagen, May 1994.

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

[6] *Backup doesn't start with current main images.* `https://perconadev.atlassian.net/browse/K8SPSMDB-584`. 2021.

[7] Marco Barletta, Marcello Cinque, Catello Di Martino, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. " Mutiny! How Does Kubernetes Fail, and What Can We Do About It? " In: *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'24)*. June 2024.

[8] Farnaz Behrang, Myra B. Cohen, and Alessandro Orso. "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.

[9] Theophilus Benson, Aditya Akella, and David Maltz. "Unraveling the Complexity of Network Management". In: *Proceedings of the 6th USENIX Symposium on Networked System Design and Implementation (NSDI'09)*. Apr. 2009.

[10] Theophilus Benson, Aditya Akella, and Aman Shaikh. "Demystifying Configuration Challenges and Trade-Offs in Network-based ISP Services". In: *Proceedings of 2011 Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'11)*. Aug. 2011.

[11] Ricardo Bianchini, Richard P. Martin, Kiran Nagaraja, Thu D. Nguyen, and Fabio Oliveira. "Human-Aware Computer System Design". In: *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)*. June 2005.

[12] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. "Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3". In: *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*. Oct. 2021.

[13] Aaron B. Brown and David A. Patterson. "Undo for Operators: Building an Undoable E-mail Store". In: *Proceedings of the 2003 USENIX Annual Technical Conference (ATC'03)*. June 2003.

[14] *Bug: Cluster unrecoverable because of incorrect primary_slot_name.* `https://github.com/cloudnative-pg/cloudnative-pg/issues/3588`. 2023.

[15] *BUG: pg_hba.conf is not valid for postgresql-patroni-ha.* `https://github.com/apecloud/kubeblocks/issues/2184`. 2023.

[16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, Omega, and Kubernetes". In: *Communications of the ACM* 59.5 (May 2016), pp. 50–57.

[17] Cristian Cadar and Koushik Sen. "Symbolic Execution For Software Testing: Three Decades Later". In: *Communications of the ACM* 56.2 (Feb. 2013), pp. 82–90.

[18] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. "A NICE Way to Test OpenFlow Applications". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. Apr. 2012.

[19] *Cass-operator crashes when using configSecret for Cassandra configuration.* `https://github.com/k8ssandra/cass-operator/issues/705`. 2024.

[20] *Cassandra operator becomes partially inoperable if replaceNodes has a wrong pod name (issue comment).* `https://github.com/k8ssandra/cass-operator/issues/315#issuecomment-1090149844`. 2022.

[21] *Cassandra vnodes: can I lower the number on slower nodes and expect rebalancing to occur automatically?* `https://stackoverflow.com/questions/32416642/cassandra-vnodes-can-i-lower-the-number-on-slower-nodes-and-expect-rebalancing/32419325#32419325`. 2015.

[22] Melanie Cebula and Bruce Sherrod. "10 Weird Ways to Blow Up Your Kubernetes". In: *KubeCon North America*. Nov. 2019.

[23] *Chaos Mesh — A Solution for System Resiliency on Kubernetes.* https://dzone.com/articles/chaos-mesh-a-chaos-engineering-solution-for-system. 2020.

[24] *Check if the CLUSTER_JOIN endpoint is ready instead of just resolving it.* https://github.com/apecloud/kubeblocks/issues/6390. 2024.

[25] Illya Chekrygin. "Keep the Space Shuttle Flying: Writing Robust Operators". In: *KubeCon Europe*. May 2019.

[26] Andong Chen, Ziyi Guo, Zhaoxuan Jin, Zhenyuan Li, and Yan Chen. "Breaking the Bulkhead: Demystifying Cross-Namespace Reference Vulnerabilities in Kubernetes Operators". In: *Proceedings of the 33rd Annual Network and Distributed System Security Symposium (NDSS'26)*. Feb. 2026.

[27] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. "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.

[28] Yinfang Chen, Jiaqi Pan, Jackson Clark, Yiming Su, Noah Zheutlin, Bhavya Bhavya, Rohan Arora, Yu Deng, Saurabh Jha, and Tianyin Xu. "Stratus: A Multi-agent System for Autonomous Reliability Engineering of Modern Clouds". In: *Proceedings of the 39th Annual Conference on Neural Information Processing Systems (NeurIPS'25)*. Dec. 2025.

[29] Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. "Push-Button Reliability Testing for Cloud-Backed Applications with Rainmaker". In: *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*. Apr. 2023.

[30] *Cloud Native Computing Foundation Operator White Paper.* https://www.cncf.io/wp-content/uploads/2021/07/CNCF_Operator_WhitePaper.pdf. 2025.

[31] *CLOUDP-116155 Initial bootup with arbiters.* https://github.com/mongodb/mongodb-kubernetes-operator/pull/1024. 2022.

[32] *Cluster unready after switching from expose LoadBalancer to ClusterIP.* https://perconadev.atlassian.net/browse/K8SPSMDB-841. 2023.

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

[34] *ConfigMaps*. https://kubernetes.io/docs/concepts/configuration/configmap. 2025.

[35] *Configure Liveness, Readiness and Startup Probes*. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes. 2025.

[36] *Conftest*. https://github.com/open-policy-agent/conftest. 2025.

[37] *Consistent Reads from Cache*. https://github.com/kubernetes/enhancements/blob/77044f023b737d42d30d4d99015a12556ea099a1/keps/sig-api-machinery/2340-Consistent-reads-from-cache/README.md. June 2024.

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

[39] *Custom Resources*. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/. 2025.

[40] *Data lost when inject network delay fault to Redis cluster*. https://github.com/apecloud/kubeblocks/issues/5107. 2023.

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

[42] *Default privileges are not set in public schema*. https://github.com/zalando/postgres-operator/issues/1420. 2021.

[43] Jason Dobies and Joshua Wood. *Kubernetes Operators: Automating the Container Orchestration Platform*. O'Reilly Media, Inc., 2020.

[44] Georgios-Petros Drosos, Thodoris Sotiropoulos, Georgios Alexopoulos, Dimitris Mitropoulos, and Zhendong Su. "When Your Infrastructure Is a Buggy Program: Understanding Faults in Infrastructure as Code Ecosystems". In: *Proc. ACM Program. Lang.* (Oct. 2024).

[45] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. "The Design and Operation of CloudLab". In: *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19)*. July 2019.

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

[47] *Ensure operator moves pods from a decommissioned node*. https://github.com/zalando/postgres-operator/issues/429. 2018.

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

[49] *etcd*. https://etcd.io/.

[50] *Failover fails with synchronous_mode and 2 instances*. https://github.com/zalando/postgres-operator/issues/2276. 2023.

[51] *FATAL: data directory "/var/lib/postgresql/data/pgdata" has invalid permissions' when bootstrapping cluster from backup*. https://github.com/cloudnative-pg/cloudnative-pg/issues/625. 2022.

[52] Vaibhav Ganatra, Anjaly Parayil, Supriyo Ghosh, Yu Kang, Minghua Ma, Chetan Bansal, Suman Nath, and Jonathan Mace. "Detection Is Better Than Cure: A Cloud Incidents Perspective". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'23)*. Nov. 2023.

[53] *Google Kubernetes Engine*. https://cloud.google.com/kubernetes-engine. 2026.

[54] Jim Gray. "Why Do Computers Stop and What Can Be Done About It?" In: *Tandem Technical Report 85.7* (June 1985).

[55] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, and Chunqiang Tang. "Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta". In: *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*. July 2023.

[56] Jiawei Tyler Gu, Xudong Sun, Zhen Tang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. "Acto: Push-Button End-to-End Testing for Operation Correctness of Kubernetes Operators". In: *USENIX ;login:* (Aug. 2024).

[57] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. "Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management". In: *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*. Oct. 2023.

[58] Sebastien Guilloux. “Writing a Kubernetes Operator: the Hard Parts”. In: *KubeCon North America*. Nov. 2019.

[59] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. “EIO: Error Handling is Occasionally Correct”. In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST’08)*. Feb. 2008.

[60] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. “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.

[61] Md Mahadi Hassan, John Salvador, Shubhra Kanti Karmaker Santu, and Akond Rahman. “State Reconciliation Defects in Infrastructure as Code”. In: *Proceedings of the ACM on Software Engineering*. July 2024.

[62] *Helm Charts*. `https://helm.sh/`. 2025.

[63] Tim Hockin. *Kubernetes: Edge vs. Level Triggered Logic*. `https://speakerdeck.com/thockin/edge-vs-level-triggered-logic`. June 2017.

[64] *How to tweak the number of num_tokens (vnodes) in live Cassandra cluster*. `https://www.pythian.com/blog/technical-track/tweak-number-of-num_tokens-vnodes-in-live-cassandra-cluster`. 2025.

[65] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. “Metastable Failures in the Wild”. In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI’22)*. July 2022.

[66] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. “Capturing and Enhancing In Situ System Observability for Failure Detection”. In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI’18)*. Oct. 2018.

[67] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. "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.

[68] *Improve components' readiness check mechanisms.* `https://github.com/pingcap/tidb-operator/issues/4760`. 2022.

[69] *Improve failover during rolling updates.* `https://github.com/zalando/postgres-operator/issues/600`. 2019.

[70] Rishabh Iyer, Jiacheng Ma, Katerina Argyraki, George Candea, and Sylvia Ratnasamy. "The Case for Performance Interfaces for Hardware Accelerators". In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS-XIX)*. 2023.

[71] Saurabh Jha, Rohan Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, Noah Zheutlin, Saki Takano, Divya Pathak, Felix George, Xinbo Wu, Bekir O. Turkkan, Gerard Vanloo, Michael Nidd, Ting Dai, Oishik Chatterjee, Pranjal Gupta, Suranjana Samanta, Pooja Aggarwal, Rong Lee, Pavankumar Murali, Jae-wook Ahn, Debanjana Kar, Ameet Rahane, Carlos Fonseca, Amit Paradkar, Yu Deng, Pratibha Moogi, Prateeti Mohapatra, Naoki Abe, Chandrasekhar Narayanaswami, Tianyin Xu, Lav R. Varshney, Ruchi Mahindru, Anca Sailer, Laura Shwartz, Daby Sow, Nicholas C. M. Fuller, and Ruchir Puri. "ITBench: Evaluating AI Agents across Diverse Real-World IT Automation Tasks". In: *Proceedings of the 42th International Conference on Machine Learning (ICML'25)*. July 2025.

[72] *K3d.* `https://github.com/k3d-io/k3d`.

[73] *KafkaConnector resources restarting every 2 minutes.* `https://github.com/strimzi/strimzi-kafka-operator/issues/2981`. 2020.

[74] Mingi Kim, Ahnjae Shin, Jaewoo Maeng, Myeongjae Jeon, and Byung-Gon Chun. "Garen: Reliable Cluster Management with Atomic State Reconciliation". In: *Proceedings of the 21st ACM European Conference on Computer Systems (EuroSys'26)*. Apr. 2026.

[75] *Kind.* `https://kind.sigs.k8s.io/`.

[76] *Kubeconform.* `https://github.com/yannh/kubeconform`. 2025.

[77] *Kubeval.* `https://github.com/instrumenta/kubeval`. 2025.

[78] Hemant Kumar and Jan Šafránek. “Storage on Kubernetes - Learning From Failures”. In: *KubeCon North America*. Nov. 2019.

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

[80] Maxime Lagresle. “Moving to Kubernetes: the Bad and the Ugly”. In: *ContainerDays*. June 2019.

[81] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. “Improving Availability in Distributed Systems with Failure Informers”. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI’13)*. Apr. 2013.

[82] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. “Detecting failures in distributed systems with the Falcon spy network”. In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP’11)*. Oct. 2011.

[83] Ao Li, Shan Lu, Suman Nath, Rohan Padhye, and Vyas Sekar. “ExChain: Exception Dependency Analysis for Root Cause Diagnosis”. In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI’24)*. Apr. 2024.

[84] Bingzhe Liu, Gangmuk Lim, Ryan Beckett, and P. Brighten Godfrey. “Kivi: Verification for Cluster Management”. In: *Proceedings of the 2024 USENIX Annual Technical Conference (ATC’24)*. July 2024.

[85] *Liveness probe failing for Prometheus Exporter connected to a large SolrCloud*. `https://github.com/apache/solr-operator/issues/693`. 2024.

[86] Chang Lou, Peng Huang, and Scott Smith. “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.

[87] Chang Lou, Yuzhuo Jing, and Peng Huang. “Demystifying and Checking Silent Semantic Violations in Large Distributed Systems”. In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI’22)*. July 2022.

[88] Chang Lou, Dimas Shidqi Parikesit, Yujin Huang, Zhewen Yang, Senapati Diwangkara, Yuzhuo Jing, Achmad Imam Kistijantoro, Ding Yuan, Suman Nath, and Peng Huang. "Deriving Semantic Checkers from Tests to Detect Silent Failures in Production Distributed Systems". In: *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI'25)*. July 2025.

[89] Sixiang Ma, Fang Zhou, Mike D. Bond, and Yang Wang. "Finding Heterogeneous-Unsafe Configuration Parameters in Cloud Systems". In: *Proceedings of the 16th ACM European Conference on Computer Systems (EuroSys'21)*. Apr. 2021.

[90] *Make LivenessProbe,ReadinessProbe initialdelayseconds,timeout configurable through CRD*. `https://github.com/pravega/zookeeper-operator/issues/275`. 2020.

[91] Valentin J.M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. "The Art, Science, and Engineering of Fuzzing: A Survey". In: *IEEE Transactions on Software Engineering* 47.11 (Nov. 2021), pp. 2312–2331.

[92] *Manual action required to expand MinIO tenant*. `https://github.com/minio/operator/issues/995`. 2022.

[93] *MariaDB Known Issues: You Must Enable Exactly N Storage Engines*. `https://mariadb.com/kb/en/transaction-coordinator-log-overview/#known-issues`. 2025.

[94] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, B. Ashok, Chetan Bansal, Chandra Maddila, Christian Bird, Sumit Asthana, and Aditya Kumar. "Rex: Preventing Bugs and Misconfiguration in Large Services using Correlated Change Analysis". In: *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. Feb. 2020.

[95] Themis Melissaris, Kunal Nabar, Rares Radut, Samir Rehmtulla, Arthur Shi, Samartha Chandrashekar, and Ioannis Papapanagiotou. "Elastic Cloud Services: Scaling Snowflake's Control Plane". In: *Proceedings of the 13th ACM Symposium on Cloud Computing (SOCC'22)*. Nov. 2022.

[96] *Migration from sharding to replica set doesn't work in some cases*. `https://perconadev.atlassian.net/browse/K8SPSMDB-345`. 2020.

[97] *Minikube*. `https://minikube.sigs.k8s.io/`.

[98] *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.

[99] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. "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.

[100] *No switchover candidate found.* https://github.com/zalando/postgres-operator/issues/1992. 2022.

[101] *Number of client connections not reevaluated dynamically in the teardown script.* https://github.com/pravega/zookeeper-operator/issues/482. 2022.

[102] *Objects In Kubernetes.* https://kubernetes.io/docs/concepts/overview/working-with-objects. 2025.

[103] Fábio Oliveira, Andrew Tjang, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. "Barricade: Defending Systems Against Operator Mistakes". In: *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. Apr. 2010.

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

[105] *Operator doesn't revalidate cluster state if node decommission failed due to disk size check.* https://github.com/k8ssandra/cass-operator/issues/639. 2024.

[106] *Operator Pattern.* https://kubernetes.io/docs/concepts/extend-kubernetes/operator/. 2025.

[107] *Operator status must reflect the status of mongos.* https://perconadev.atlassian.net/browse/K8SPSMDB-302. 2020.

[108] David Oppenheimer, Archana Ganapathi, and David A. Patterson. "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.

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

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

[111] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi. "IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services". In: *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19)*. July 2019.

[112] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. *Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*. Tech. rep. UCB//CSD-02-1175. University of California Berkeley, Mar. 2002.

[113] *PD placement rules should be enabled if using TiFlash*. `https://github.com/pingcap/tidb-operator/issues/2219`. 2020.

[114] *Persistent Volumes*. `https://kubernetes.io/docs/concepts/storage/persistent-volumes`. 2025.

[115] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. "Bucketing Failing Tests via Symbolic Analysis". In: *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering (FASE'17)*. Apr. 2017.

[116] *PMM client cannot connect to mongodb when requireTLS mode activated*. `https://perconadev.atlassian.net/browse/K8SPSMDB-765`. 2022.

[117] *Pods*. `https://kubernetes.io/docs/concepts/workloads/pods`. 2025.

[118] *Pooler issue after upgrade to 1.5. With "error: unexpected response from login query"*. `https://github.com/zalando/postgres-operator/issues/1060`. 2020.

[119] *RabbitMQ server missing role rule to create events*. `https://github.com/rabbitmq/cluster-operator/issues/264`. 2020.

[120] Ariel Rabkin and Randy Katz. "Static Extraction of Program Configuration Options". In: *Proceedings of the 33th International Conference on Software Engineering (ICSE'11)*. May 2011.

[121] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. "Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Oct. 2020.

[122] Akond Rahman, Chris Parnin, and Laurie Williams. "The Seven Sins: Security Smells in Infrastructure as Code Scripts". In: *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. May 2019.

[123] *RBAC permissions issue in OpenShift*. `https://github.com/cockroachdb/cockroach-operator/issues/780`. 2021.

[124] *Red Hat OpenShift*. `https://docs.openshift.com/`. 2021.

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

[126] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. "SibylFS: Formal Specification and Oracle-based Testing for POSIX and Real-world File Systems". In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)*. Oct. 2015.

[127] *Schema creation failed with permission error in preparedDatabases*. `https://github.com/zalando/postgres-operator/issues/1130`. 2020.

[128] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: Flexible, Scalable Schedulers for Large Compute Clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. Apr. 2013.

[129] *Service*. `https://kubernetes.io/docs/concepts/services-networking/service`. 2025.

[130] Manish Shetty, Yinfang Chen, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Xuchao Zhang, Jonathan Mace, Dax Vandevoorde, Pedro Las-Casas, Shachee Mishra Gupta, Suman Nath, Chetan Bansal, and Saravan Rajmohan. "Building AI Agents for Autonomous Clouds: Challenges and Design Principles". In: *Proceedings of 15th ACM Symposium on Cloud Computing (SoCC'24)*. Nov. 2024.

[131] *Single node PG 15 cluster stuck in Taking first backup*. `https://github.com/cloudnative-pg/cloudnative-pg/issues/896`. 2022.

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

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

[134] *Strimzi not able to rotate kafka/zookeeper pods when they are put in failed state*. `https://github.com/strimzi/strimzi-kafka-operator/issues/7290`. 2022.

[135] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. "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.

[136] Xudong Sun, Jiawei Tyler Gu, Cody Rivera, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. "Anvil: Building Kubernetes Controllers That Do Not Break". In: *USENIX ;login:* (June 2024).

[137] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. "Automatic Reliability Testing for Cluster Management Controllers". In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. July 2022.

[138] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. "Sieve: Chaos Testing for Kubernetes Controllers". In: *USENIX ;login:* (Nov. 2024).

[139] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. "Anvil: Verifying Liveness of Cluster Management Controllers". In: *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*. July 2024.

[140] Xudong Sun, Lalith Suresh, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lilia Tang, and Tianyin Xu. "Reasoning about modern datacenter infrastructures using partial histories". In: *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)*. May 2021.

[141] *Support managed scale down of SolrClouds*. `https://github.com/apache/solr-operator/issues/559`. 2023.

[142] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. "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.

[143] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. "Holistic Configuration Management at Facebook". In: *Proceedings of the 25th ACM Symposium on Operating System Principles (SOSP'15)*. Oct. 2015.

[144] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. "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.

[145] Lilia Tang, Chaitanya Bhandari, Yongle Zhang, Anna Karanika, Shuyang Ji, Indranil Gupta, and Tianyin Xu. "Fail through the Cracks: Cross-System Interaction Failures in Modern Cloud Systems". In: *Proceedings of the 18th European Conference on Computer Systems (EuroSys'23)*. May 2023.

[146] Guy Templeton and Stuart Davidson. "How a Couple of Characters (and GitOps) Brought Down Our Site". In: *KubeCon Europe*. May 2022.

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

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

[149] *TiDB crash loop when enabling binlog.* `https://github.com/pingcap/tidb-operator/issues/4945`. 2023.

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

[151] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. "Borg: The Next Generation". In: *Proceedings of the 15th ACM European Conference on Computer Systems (EuroSys'20)*. Apr. 2020.

[152] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. "Semantic Crash Bucketing". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. Sept. 2018.

[153] *Topic Operator failing to start with io.vertx.core.VertxException: Thread blocked.* https://github.com/strimzi/strimzi-kafka-operator/issues/6046. 2021.

[154] *Try to behave better when upgrading ephemeral SolrClouds.* https://github.com/apache/solr-operator/issues/365. 2021.

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

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

[157] *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.

[158] *Upgrade Galera Cluster.* https://mariadb.com/docs/galera-cluster/galera-management/upgrading-galera-cluster. 2025.

[159] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-Scale Cluster Management at Google with Borg". In: *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*. Apr. 2015.

[160] Jiarong Xing, Kuo-Feng Hsu, Yiting Xia, Yan Cai, Yanping Li, Ying Zhang, and Ang Chen. "Occam: A Programming System for Reliable Network Management". In: *Proceedings of the 19th European Conference on Computer Systems (EuroSys'24)*. Apr. 2024.

[161] Qingxin Xu, Yu Gao, and Jun Wei. "An Empirical Study on Kubernetes Operator Bugs". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'24)*. Sept. 2024.

[162] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. "Do Not Blame Users for Misconfigurations". In: *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)*. Nov. 2013.

[163] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. "An Empirical Study on Configuration Errors in Commercial and Open Source Systems". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. Oct. 2011.

[164] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. Oct. 2014.

[165] Ennan Zhai, Ang Chen, Ruzica Piskac, Mahesh Balakrishnan, Bingchuan Tian, Bo Song, and Haoliang Zhang. "Check before You Change: Preventing Correlated Failures in Service Updates". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. Feb. 2020.

[166] Jiaqi Zhang, Lakshmi Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. "EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection". In: *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)*. Mar. 2014.

[167] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. "Understanding and Detecting Software Upgrade Failures in Distributed Systems". In: *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)*. Oct. 2021.

[168] Naiqian Zheng, Tianshuo Qiao, Xuanzhe Liu, and Xin Jin. "MeshTest: End-to-End Testing for Service Mesh Traffic Management". In: *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'25)*. Apr. 2025.

[169] *zoo.cfg updated parameters are not picked up during rolling restarts.* `https://github.com/pravega/zookeeper-operator/issues/222`. 2020.

[170] *Zookeeper pod label customization not working.* `https://github.com/apache/solr-operator/issues/490`. 2022.

[171] *Zookeeper service ports should be prefixed with "tcp".* `https://github.com/pravega/zookeeper-operator/issues/183`. 2020.