Who Watches the Watchers? On the Reliability of Softwarizing Cloud Application Management

Jiawei Tyler Gu, Zhen Tang, Yiming Su, Bogdan Alexandru Stoica, Xudong Sun,
William X. Zheng, Yue Zhang[†], Akond Rahman[†], Chen Wang[‡], Tianyin Xu
University of Illinois Urbana-Champaign [†]Auburn University [‡]IBM Research

Abstract

Modern cloud applications are increasingly managed by software programs, often named "operators," which automate laborious, human-based operations. While operator programs largely prevent human mistakes, their own reliability has unprecedented impact on managed applications. This paper discusses the emerging challenges of operator program reliability on cloud-native platforms like Kubernetes. Our work is grounded in a rigorous analysis of 412 real-world failures of thirteen Kubernetes operators. We find that challenges of operator reliability come from the multifold complexity of an operator's interactions with its managed applications, environment, and user interface. Among these, operators' interactions with managed applications are the largest contributor to real-world operator failures, but they are largely overlooked—these interactions are often *ad hoc* and lack well-defined interfaces. We advocate to rethink the management interface of cloud applications and demonstrate this urgent need by showing the prevalence of defects in existing operators. Specifically, we develop a simple testing tool to exercise interactions between operators and the managed cloud applications, which discovered 86 new bugs in six popular Kubernetes operators.

1 Introduction

Modern cloud applications are increasingly managed by software programs which replace laborious, human-based operations [73, 74, 83, 100, 103]. On modern cloud platforms like Kubernetes, these management programs are commonly referred to as "operators" [1], to draw analogies with human operators [65]. Today, operators go far beyond application deployment tasks like traditional infrastructure-as-code (IaC) scripts do [66, 99]. They are long-running production services that *continuously* manage *production applications* (upgrading software versions, updating configurations, autoscaling based on workloads, handling unexpected failures, etc.).

While operator programs largely eliminate inadvertent human mistakes [55, 93–95, 97], their own correctness has unprecedented impacts. A runaway operator can directly and continuously damage bug-free applications in production. Recent studies show that software bugs in operators can lead to disastrous consequences like data loss, service unavailability, and security issues [69, 87, 106, 115], along with other significant production incidents [59, 60, 70, 81, 82, 113]. Given the trend of AI-driven operators [62, 63, 80], ensuring their reliability is crucial to prevent more frequent operator incidents.

In this paper, we discuss emerging challenges of operator reliability in the context of cloud application management. In principle, a reliable operator must (1) always reconcile the managed applications to their desired states, (2) recover applications 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 [50, 69, 87, 106, 108, 109] 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 (§2.2), 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 [115] on characterizing how *generic* software bugs manifest in operators (see §6), our work focuses on the essential complexity of softwarizing cloud application management and the fundamental challenge of ensuring correct interactions between operators and the cloud applications they manage.

Our analysis shows that reliability challenges come from the multifold complexity of operators' interactions with (1) cloud applications, (2) cloud platform like Kubernetes, (3) co-located controllers, and (4) user interface. Prior work studied (2)–(3) for testing and verification [50, 69, 87, 106, 108, 110]. Specifically, (2)–(3) are done by modeling an operator as a controller running in distributed systems with faults [106, 108], asynchrony [106, 108], and interference with other controllers [87]; (4) is done by user interface fuzzing [69] and declarative programming [110]. No prior work discussed (1).

A main finding is that *erroneous interactions with the managed applications are the dominant causes of operator failures in the field*—they contribute 42% to the studied operator failures, outnumbering other interaction issues. Unfortunately, such interactions are largely overlooked in bug studies like [115] and in testing/verification techniques. Techniques that treat operators as controllers [87, 106, 108] are inherently application agnostic. Techniques for operator programs like Acto [68, 69] have no application-specific knowledge—Acto

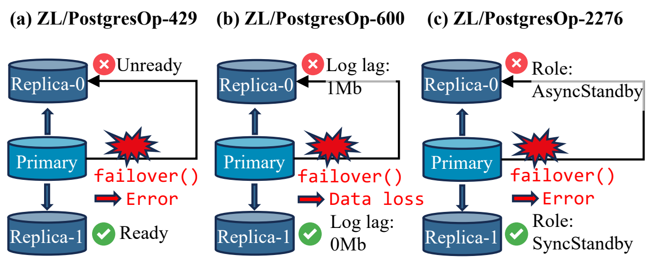


Figure 1: **Three failover operation failures of a commercial PostgreSQL operator (ZL/PostgresOp).** Each failure was caused by the operator's violating a different precondition required by PostgreSQL's failover operation.

skips application-specific properties during test generation, as it cannot infer information of such properties based on platform APIs (e.g., Kubernetes native resources).

In essence, today's cloud applications lack well-defined management interfaces for operators. However, many critical management operations have sophisticated semantics and their correctness relies on application configuration, states, execution environments, etc.; many of them are not exposed explicitly. It is challenging for operator developers to capture all fine-grained operation semantics and observe application internal states, especially considering that operator developers may not be application developers (instead, they are application users). As a result, interactions between operators and applications 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 [17] 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 [18] 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 [43] 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.

We advocate to rethink management operation interface of cloud applications and improve their *manageability*. We demonstrate the urgent need by showing the prevalence of defects residing in operator-application interactions of mature, widely used Kubernetes operators. We develop a simple testing tool named OAT to exercise how an operator manages the target cloud applications. OAT uses targeted testing policies to exercise operator-application interactions by instructing the operator to run different operations on its managed application. A key challenge is to automatically generate valid, meaningful operation commands that exercise the operator-application interface, with minimal input from developers. OAT learns such application-specific commands from examples in the operator's test suite, and relies on large language models to synthesize them when examples are unavailable. To drive the application to certain bug-triggering states, OAT also injects various faults that are expected to be handled by the operator, with policies guided by our study.

We applied OAT to six popular Kubernetes operators for managing Cassandra, Kafka, MariaDB, MinIO, MongoDB, and TiDB. OAT found 86 new bugs that have severe consequences on application availability, reliability, and security. So far, 53 of these bugs have been confirmed and 28 have been fixed. In addition, OAT revealed 13 undocumented management operation semantics through testing.

Contributions. This paper makes four main contributions:

- A discussion on emerging challenges of operator reliability for softwarized cloud application management;
- An analysis of 412 operator failures, with a focus on those where the operator failed to manage cloud applications;
- A practical testing tool, OAT, for exercising operator-application interactions, which found 86 new bugs in six popular Kubernetes operators (53 confirmed and 28 fixed);
- Artifact: https://github.com/xlab-uiuc/acto/tree/nsdi26-ae.

2 Background

Software operators are management programs running atop modern cloud platforms such as Kubernetes [57], Twine [111], and ECS [92]. They constitute the management plane of cloud applications. Different from traditional infrastructure-as-code (IaC) that are often ad hoc, one-off scripts [66, 99], operators are developed as long-running production services embodied in reusable, well-maintained system programs [1].

In modern cloud platforms like Kubernetes, operators are implemented as custom controllers [10] that continuously reconcile the application from its current states to desired states. The desired states are specified through a *declarative* interface (e.g., Custom Resource [5] in Kubernetes). In this way, users declare *what* states they want their applications to be in, and the operator addresses *how* to drive applications to reach the desired states. Operators invoke Kubernetes APIs to allocate system resources (e.g., pods and volumes), creating application execution environments. Operators also interact with the running application to update its runtime behavior.

2.1 Interactions

An operator, by design, involves complex, multi-dimensional interactions (Figure 2), which introduce unique reliability challenges beyond generic software bugs.

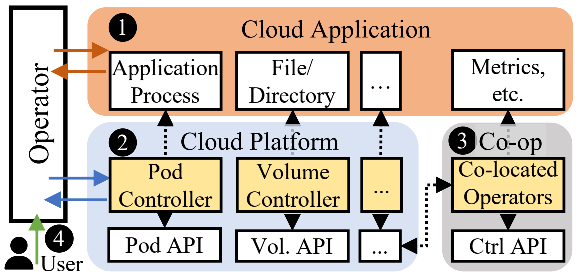


Figure 2: **Multi-dimensional interactions between an operator and its managed application, environment, and users.**

❶ Interactions with the managed application. Operators are designed for managing applications and thus must interact with them. Such interactions are often *application-specific* because applications, even when similar, have different APIs. Such interactions are embodied in different forms, including invoking application APIs, executing CLI commands in the application pod, setting environment variables or startup scripts, and changing application 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 applications. In Kubernetes, system resources are represented as API objects [9] 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 application.

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 application-level communications by injecting sidecars into application containers. The altered policy may conflict with how the operator manages the application's network.

❹ Interactions with user interface. Kubernetes operators define user interfaces in the form of Custom Resources (CRs) [5]. Here, "*users*" refer to entities, like upstream services or AI agents [80], that define the application's desired states. Each CR specifies a collection of properties describing the state of the managed application, 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.

Tool	Scope	Mechanism
Software Testing and Fault Injection		
Acto [68, 69]	Operator	Functional testing by fuzzing desired states
Sieve [106, 107]	Controller	Model-based fault injection testing
Mutiny [50]	Kubernetes	Injecting general faults to controllers
MeshTest [123]	Controller	Func. testing for ServiceMesh controllers
Formal Verification and Model Checking		
Anvil [105, 108]	Controller	Verifying liveness and safety properties
Kivi [87]	Controller	Model checking controller interactions
Programming and System Support		
DCM [110]	Controller	Declarative programming support
KEP-2340 [4]	Controller	Preventing reading stale state from caches
Transaction [15]	Controller	Transaction support for controllers

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

2.2 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 applications, but focus on controllers—in modern cloud platforms like Kubernetes, operators are implemented as custom controllers [10]. These techniques are application-agnostic and cannot address operator-application interactions (❶ in §2.1).

Table 1 does not show developer-written tests in the form of unit, integration, and system tests. However, as reported by prior work [69,106,123], 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 applications or system environments.

3 Methodology and General Findings

3.1 Methodology

We collected a dataset of 412 failures of 13 popular Kubernetes operators and conducted 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 applications, including server applications (e.g., MySQL and PostgreSQL), distributed applications (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 application. (3) they are all mature software projects developed by either the official developers of target applications or companies that provide commercial services based on the applications. The sizes of the projects are typically tens of thousands of lines of code.

For each studied operator, we randomly sampled a hundred

Operator	Application	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.

closed, fixed issues that report failure cases from its issue database and manually inspected every issue. We filter out feature requests, user questions, or issues related to building and testing. We only considered 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).

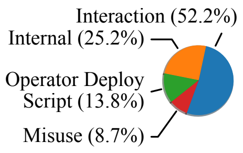
During our analysis, each failure case was analyzed by at least two authors to minimize human errors and subjectivity in the interpretation and categorization.

3.2 General Findings

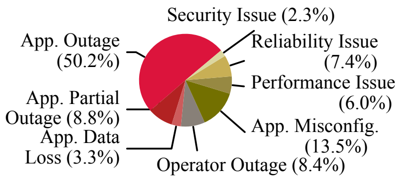
Finding 1: *The majority (52.2%) of the studied operator failures are caused by defects in operators' interactions with external entities (see §2.1), which significantly outnumber bugs in operators' internal program logic.*

Figure 3a 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 [69] 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 applications and environments). Unfortunately, as reported in [69], 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 [6]) and by misuses of the operators. While these cases are interesting, they are orthogonal to the operator's design and implementation, and thus are not the focus of this paper.



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



(b) **Consequences.**

Figure 3: **Root-cause locations of the studied 412 operator failures (3a), and the consequences of the 215 interaction-related operator failures (3b).**

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

Figure 3b shows the consequences of the 215 interaction-related operator failures. 50.2% (108/215) resulted in full outages of the managed applications. For example, MongoOp failed to reconfigure MongoDB cluster membership after changing pod IPs [41], leaving MongoDB pods unable to connect with each other and causing a full outage. 8.8% caused partial failures of the managed applications, 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 [37]. 3.3% caused silent data loss (e.g., Figure 1b).

The other 29.3% (63/215) interaction failures led to undesired application 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 application 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 applications in production, even though they result in management service unavailability (e.g., the application cannot autoscale after the operator crashes).

Finding 3: *Failure of managing applications 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). The largest category is operator-application interaction failure, i.e., the operator failed to manage applications. The operators' interactions with the cloud platform (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 applications, and the latter shows the challenge of correctly implementing complex declarative operation interfaces [110]. Operators' interaction with co-located controllers has a small percentage, as applications 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.

Operator	❶ App.	❷ Platf.	❸ Co-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.**

While the interactions with managed applications are error-prone and caused the most failures, few existing studies or tools (see §2.2) address them. Hence, in the remainder of this paper we focus on operator-application failures to understand why operators failed to manage cloud applications.

4 Failures of Managing Applications

An operator manages cloud applications through continuous *state reconciliation* [57, 69, 106, 108]. The operator observes the state of the managed application. If the current application 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 application. 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 application 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-application interactions during a management operation include:

- **Application API.** An operator calls APIs of the applications to invoke their internal procedure, e.g., calling PostgreSQL's API to start `failover` (Figure 1).
- **Application configuration.** The operator updates the application's configuration by updating configuration files, databases, or ConfigMap objects [2].
- **Execution environment.** The operator changes the execution environment of its managed applications, such as the files and environment variables.
- **Resource provisioning.** The operator allocates (and deallocates) system resources such as pods [12], volumes [11], and services [13] for the managed application based on its configuration or scaling operations.

Patterns	**Description**	**Fail. # (%)**
Semantic violations	The operator violates operation semantics required by the application (Fig. 1, 4a–4b).	58 (63.7%)
State observability	The operator fails to observe internal states of the application (Figure 4d).	15 (16.5%)
Version incompatibility	The operator fails to handle inconsistent behavior across app. versions (Figure 4e).	11 (12.1%)
Mishandling app. errors	The operator mishandles errors returned by the application (Figure 4f).	7 (7.7%)
Total		91

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

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

- *The majority (63.7%) of studied application-management failures are caused by the operator violating its managed application's operation semantics.*
- *A significant percentage (16.5%) of management failures are caused by the gaps that prevent the operator from observing the application internal states.*
- *Incompatibility between the operator and its managed application also causes a significant percentage (12.1%) of failures, triggered by upgrading application versions.*
- *The remaining cases (7.7%) are caused by the operator mishandling application errors.*

4.1 Operation Semantic Violations

The most common failure pattern is that the operators fail to satisfy the operation semantics of the applications; as a result, they issue unsafe operations that break the applications.

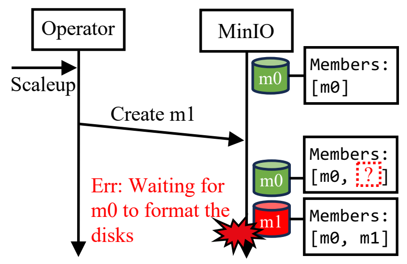
Cloud applications are large, sophisticated systems, with complex management semantics that define how they should be managed correctly. Unfortunately, in practice, such semantics are not always explicitly documented and are rarely formally specified. Hence, it is difficult for the operator to comprehend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

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

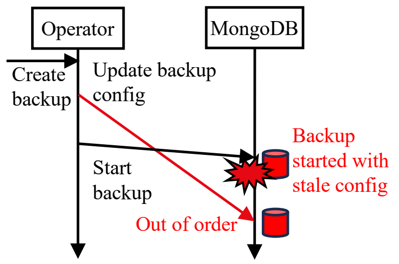
We discuss common violated application operation semantics.

4.1.1 Application Configuration

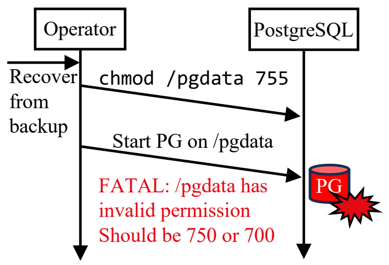
Many operations are done by changing the application's configuration at runtime, either by updating a configuration file (which requires restarting the application process) or by calling the application's configuration API/CLI. It requires operators to manage application configuration correctly. Software configuration management is a known challenge and is well studied in the literature [91, 104, 116–118, 121].



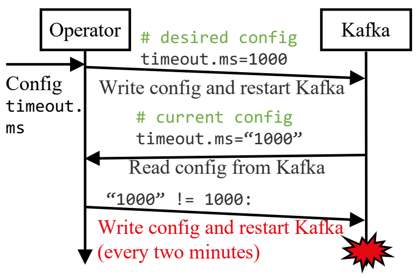
(a) **Semantic violations (app. config).** MinIOOp did not update membership configuration of `m0` when adding `m1` [34].



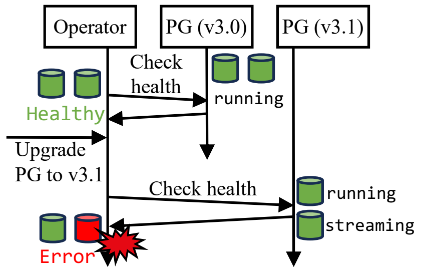
(b) **Semantic violations (order).** MongoOp did not wait for configuration updates to finish before starting the backup operation [28].



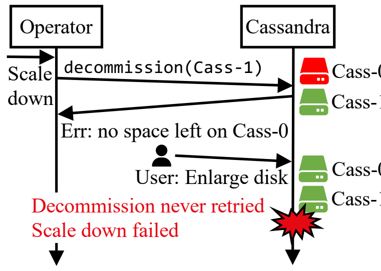
(c) **Semantic violations (env.).** CN/PostgresOp mistakenly set the mode of the data directory to be too permissive [32].



(d) **State observability.** KafkaOp fails to observe that the current configuration matches the desired one due to type mismatch, resulting in infinite restarts [20].



(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 [35].



(f) **Mishandling application errors.** CassOp never retried the failed node decommission operation, after the user resolved the error [48].

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

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 [118]. 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 application configuration are surfaced and are 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 multiple parameters (two violated order dependencies and one violated a value dependency [61]).

Finding 6: *Deep semantics of application 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 4a shows an example of cross-node misconfiguration where different MinIO replica nodes should have consistent membership lists, 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 application 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.

Patterns	**Description**	**Fail. # (%)**
Configuration	Configuration of the managed application across nodes and components (Figure 4a).	21 (36.2%)
Ordering (mul. ops)	Order dependency among multiple inter-dependent operations (Figure 4b).	18 (31.0%)
Precondition (single op.)	Preconditions of an operation in terms of application states (Figure 1).	11 (19.0%)
Environment	Execution environment of the app. (Fig. 4c).	8 (13.8%)
Total		58

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

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 [24]. When MongoDB cluster runs in sharding mode, all Mongod instances are expected to run with the `shardsvr` configuration [22].

The remaining seven failures all fall into a simple pattern—the operator failed to update the configuration of the running application. The essence is that application'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 [27, 39, 40] that was not read by the application, or the wrong database with limited scope [25, 26, 29]. In all these cases, the operator believed a successful configuration update, while the application ran with an old configuration.

4.1.2 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 4b, 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.*

4.1.3 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 application 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 [36] and data loss [31, 42, 44].

Finding 8: *Violations of preconditions on application states mostly happen 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 applications.*

4.1.4 Environment

The remaining eight failure cases were caused by the operators incorrectly preparing or managing the execution environment of the managed applications. In these cases, the operator either did not create the files or data directories expected by the applications, or misconfigured their permissions (e.g., Figure 4c). As a result, the applications 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.

4.2 State Observability

The state-reconciliation principle relies on the observability of application states. Different from the cluster states that are encoded in well-defined state objects [9], it is challenging to observe an application'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 application state matches the desired state.

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

Readiness and liveness monitors. Kubernetes allows operators to register monitors (called probes [3]) that check readiness and liveness of managed applications. Eight (out of 15) failures were caused by *ad hoc*, unreliable probes. Unreliable readiness probes may incorrectly instruct clients to connect to unready applications and fail client requests. A common pattern is to use approximate signals, such as container startup [33] and DNS resolution [46], to indicate application readiness; such signals are flaky.

Unreliable liveness probes may incorrectly instruct Kubernetes to reboot the application containers, causing disruptions. In all three cases [21, 30, 47], the liveness probes reported false alarms due to timeout of the probes when applications 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 application support, operators have to implement their own logic for interpreting the application'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 application's current state and the desired state. This causes operators to keep reconciling applications even when the application has already reached the desired state, as shown in Figure 4d. 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) [72, 76, 78, 88], despite many iterations (e.g., [23]). The deficiency of probes corroborates a recent industry report [67]. Integrating advanced observability techniques [77, 84, 85, 96] and state-encoding interface are desired but are challenging for diverse cloud applications.

4.3 Version Incompatibility

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

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

Figure 4e 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 [38], 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 impairs backup operations.

Version compatibility is a classic software reliability problem and has been recently studied in the context of software upgrades [120, 122]. However, different from traditional software compatibility, it is challenging to ensure compatibility between the operator and the managed application, without a clean management interface. Specifically, an operator is expected to work with different versions of the managed applications 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 applications cannot be systematically solved.

4.4 Mishandling Application Errors

Error handling is a long-lasting challenge and a well-studied problem [64, 71, 86, 119]. Since the operator should reconcile application to the desired state from any state (including error states), it is responsible for handling application errors. However, as mature cloud applications 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 application; however, the line is often blurry.

In 5 (out of 7) failure cases, the operator did not handle errors—when the application returned an error code, the operator chose to exit (e.g., Figure 4f). It is a simple, safe strategy, but may miss opportunities.

In the other two cases, the operator mishandled application errors. For example, TiDBOp treated all errors in the application 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.

4.5 Discussion

The above study reveals the significant challenges of correctly managing today's cloud applications using softwarized operators. Certainly, an ultimate solution is to rethink and redesign cloud applications 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 application manageability in a more evolutionary way.

4.5.1 The Case for Management Interfaces

There is an alarming lack of techniques to validate whether an operator correctly follows the application's management operation semantics. The current practice of relying on application 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 (§4.1). There is a pressing need to build management interfaces that precisely describe operation semantics for applications.

We envision that application management interfaces are much simpler programs compared to the original application but preserve its management operation semantics. For example, for an application API, the management interface should precisely describe the conditions for this API to succeed, including constraints on configurations, environments, and application states. Management interfaces should also be versioned and evolved as the applications evolve to prevent version incompatibility (§4.3).

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 an application, and validation techniques that allow operators to perform dry runs of their operations before interacting with the application. The interfaces could also enable developers to measure the interface complexity and evaluate the manageability of their applications in a similar vein as [51, 52].

Developing management interfaces is challenging as mature applications 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 [79] capture only latency-relevant behaviors.

4.5.2 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 applications. The state machine model includes actions representing all management APIs exposed by applications (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 [108] lack a systematic way to model the interactions between the operator and the application. It burdens operator developers to manually write specifications of the application 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 [108].*"

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 [54] and file systems [101] is to perform property-based testing to compare the behavior of the model and the application. There are many exciting research problems that lie in how to develop, maintain, and even synthesize formal models as management interfaces.

4.5.3 Improving Testing of Softwarized Operators

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

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

Finding 11: *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 application-specific properties.*

Existing tools for operator and controller testing [69, 106] are *application agnostic*—they only mutate system resources properties, but skip application-specific properties.

Second, the new tool must inject faults against applications.

Finding 12: *40.7% (37/91) of target failures need to be triggered by external faults that occur on the applications.*

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

5 Testing Operator-Application Interactions

Driven by the discussion in §4.5.3, we develop OAT, a simple tool for testing interactions between operators and their managed cloud applications. OAT targets bugs that manifest via different patterns of operator-application interaction failures (Table 4). Those bugs cannot be found by existing tools as they are all application agnostic (see §2.2). Due to space limit, we present the high-level implementation of OAT and the results of applying it to several Kubernetes operators in this section. More details can be found in §A and §B.

5.1 Implementation of OAT

OAT follows the end-to-end paradigm of operator/controller testing that exercises the target operator together with the application [69, 106, 123]. 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 application, until a bug is caught or a time budget is reached. Operation commands change desired application states, while faults change current states.

During a test campaign, OAT monitors the application with two key principles: (1) normal operation commands should not affect the availability of the applications 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.

OAT must address two main technical challenges:

- *How to explore the state space?* It is prohibitively expensive to enumerate all possible application states and all possible faults, not to mention their combinations.

- *How to automatically generate application-specific operations?* To detect diverse bugs, the testing tool should invoke different types of management operations.

5.1.1 Exploring State Space

OAT takes an empirical approach to explore the application's state space during its test campaign. Its testing policy is driven by our analysis (see Table 6). We separate concerns of generating operation commands that drive the operator to reconcile the application to different desired states, (discussed in §5.1.2) and injecting faults that disturb current application states. During a test campaign, OAT continuously runs tests where each test realizes one pattern of Table 6.

The faults are injected during state reconciliation using ChaosMesh [19]. For container crashes, OAT kills the application container to test failover and recovery-related operations. For network disconnection, OAT drops network requests between operator and application containers to test if the operator correctly handles returned errors. For network delay, OAT blocks network requests of the operator until it observes subsequent ones to force a different order.

Limitation. OAT's exploration strategy provides no guarantee to exhaustively exercise all possible failure cases on the operator-application interactions; thus it is not complete.

5.1.2 Synthesizing Application-Specific Properties

OAT generates operation commands by synthesizing application specific properties into a desired state declaration. The key challenge is to automatically generate different values of application-specific properties that are semantically valid and meaningful. Randomly fuzzing property values is ineffective, e.g., assigning a randomly generated string to an image ID or a ConfigMap object does not yield a valid desired state.

OAT synthesizes application-specific property values using two approaches. First, mature operator projects include abundant developer-written unit tests, where developers created operation commands that specify application-specific properties (e.g., tests need to create application configuration and specify container images). In Kubernetes, an operation command is embodied by a declaration of a desired state. A desired state is described by *properties* of Custom Resource (CR) [5]; the resource refers to the application and properties describe its container image, configuration, replica count, etc.

OAT extracts values of each property from different tests and synthesizes them into new desired state. The extraction and synthesis are done automatically, because the desired state is defined structurally in CR Definition (CRD) [5]. Note that OAT is not redundant with original unit tests: (1) the desired-state declaration is synthesized from multiple examples in different tests (using a similar idea as Frankencerts [56]), (2) the synthesized commands are not for unit testing, but for testing operator-application interactions. OAT requires at least two values for each property to drive state reconciliation.

Patterns		**Operation Commands**	**Faults**
Semantics	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 6: **OAT's test policies for different failure patterns.**

Second, when examples from existing tests are unavailable, OAT leverages large language models (LLMs) to generate values for application-specific properties. OAT uses a prompt (Figure 6) that includes the definition of the property, its description, and type information (extracted from the CRD [5]); it asks an LLM to output valid values in a YAML format.

The two techniques complement each other—developer-written examples are semantically valid and tend to exercise realistic scenarios, while the LLM fallback ensures coverage of every property. On average, 82.2% of property values of an operator can be extracted from the artifacts in the project.

Note that the synthesis is done offline before a test campaign (where OAT applies synthesized commands).

5.1.3 User Interface

With aforementioned efforts, OAT provides a fully automatic solution for operator testing. Users only need to provide three inputs: (1) a manifest for building and deploying the operator, (2) the definition of state declaration (e.g., the CRD of Kubernetes operators [5]), and (3) the operator repository (OAT scans the code and configuration files to synthesize operation commands). Note that these are standard inputs for operator testing tools [69, 104, 123].

OAT encourages users to provide additional application-specific utilities which will significantly enhance its ability:

State monitors. OAT cannot comprehend application-defined states that are not reflected or not precisely encoded in state-query API of Kubernetes. For example, in Kubernetes, application configurations are commonly encoded in a ConfigMap object which serializes the application file into a text blob. It is both ineffective and brittle to check if the application reaches desired configuration by directly comparing the blob (see §4.2). OAT thus benefits from user-defined configuration monitors that query application configuration and serialize them into a unified format for equivalence check.

Application workloads. OAT also benefits from developer-provided application workloads that can be used to measure the availability of the cloud application. We expect normal operations to not affect application availability. For tests with injected faults (we only inject common, transient faults), we expect the application availability should not drop below a predefined threshold (95% in our evaluation).

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 7: **New bugs found by OAT in evaluated operators.**

5.2 Experiment Setup

We apply OAT to six popular, mature Kubernetes operators which manage critical cloud applications (see Table 7). We select five operators from our study (Table 2) that cover different types of applications 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 7).

For each operator, we provide OAT with a state monitor for application configuration and an application workload which measures application availability (§5.1.3), 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.

5.3 Results and Experience

Finding 13: *Defects in operator-application 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 7 presents the bugs found by OAT of different patterns in each operator. OAT detected 86 *new* bugs and reported no false alarm (see §B.2). 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 applications. 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 application's management operation semantics are the largest category (46 out of 86). Among them, 33 bugs violate the semantics of application configuration; 6 bugs violate preconditions of operations; 2 bugs set up incorrect execution environment; 5 bugs violate order dependencies of multiple operations.

Finding 14: *Existing documents on application management are too vague to follow and miss important management operation semantics.*

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 [7] or experience reports [16]. 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 [14]*" 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 section [8].

Finding 15: OAT *exposed diverse patterns of code bugs at the operator-application 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 operator-application interactions without assuming particular code idioms. Therefore, it can find diverse bug patterns that manifest through the interaction failures. Among the 86 new bugs, only 37.2% (32) match the code-level patterns previously catalogued in our study (§4). The remaining 62.8% highlight that implementing operators reliably cannot be assured by targeting known faulty patterns alone. Many failures arise because the operator-application interface is under-specified and embeds implicit semantics. We summarize new bug patterns uncovered by OAT:

- *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 overwrites 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 use probes that only apply to specific configurations. 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 application components.* In MongoOp-1157, downgrading the MongoDB cluster from v7.0.8 to v6.0.15 causes the MongoOp to become 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-application interactions but stem from internal issues (Table 7). OAT exposed bugs that result in inconsistencies between the application 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 [45]). This further shows the challenges of hardening already under-specified interfaces.

Finding 16: *43.0% (37/86) of the bugs require application-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 [69]). Instead, they are manifested through inconsistencies between application configuration states and the ConfigMap object (33 bugs) and transient application 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.

6 Related Work

We discussed recent efforts on improving reliability of controllers and operators in §2.2. The most related work is Acto [68,69] which is the only technique that targets operators for cloud applications (the others all target controllers). However, Acto does not address operator-application 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 application-specific properties to exercise how the operator manages the cloud application. OAT also considers various faults, while Acto only performs functional testing without external faults.

Xu et al. [115] characterize how *generic* software bugs manifest in operators. Our work differs in three aspects. First, we focus on understanding the essential complexity of softwarizing cloud application 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 applications; 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-application interactions.

The operator-application interaction failures can be viewed as a special kind of cross-system interaction failures [112]. Operators are not involved in the control and data planes of cloud applications; they construct the management plane. They interface with cloud applications' internal management components which were studied in [112].

Our work is complementary to studies on reliability of Infrastructure-as-Code (IaC) [66,75,98,99]. Unlike IaC which mostly handles one-shot, static infrastructure deployment, operators manage the entire application lifecycle, handling continuous upgrades, runtime reconfiguration, failover, recovery, etc. This increased complexity forces operators to reason about evolving application states and understand application 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 applications in production.

Prior work validates the implementation correctness of cloud application interface semantics [89, 90]. 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 [49, 51, 52, 58, 114], especially those on management complexity from a reliability perspective. Compared with routers and switches, cloud applications are more diverse and complex, creating new challenges for management operations.

7 Concluding Remarks

We discussed the essential challenges of softwarizing cloud application management based on 412 real-world operator failures and developed OAT, a testing tool which exposed 86 new bugs in six popular Kubernetes operators, showing that reliability remains a major concern. We advocate to rethink the management interface, and envision formal specifications as interfaces to enable various systematic testing and verification techniques. As cloud applications scale and evolve, manageability must return as a first-class design principle [53], especially as operators shift toward software and AI.

Acknowledgement

We thank the anonymous reviewers, and our shepherd, Chang Lou, for their insightful comments. We thank Wentao Zhang, Thrivikraman Varadharajan, Tianyi Huang, and Zhixing (Mark) Zhang for helping study operator issues. This work was funded in part by NSF CNS-2145295, CNS-2130560, SHF-2312321, and an IIDAI Entrepreneurship grant.

References

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

[2] ConfigMaps. https://kubernetes.io/docs/concepts/configuration/configmap.

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

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

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

[6] Helm Charts. https://helm.sh/.

[7] 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.

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

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

[10] Operator Pattern. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[11] Persistent Volumes. https://kubernetes.io/docs/concepts/storage/persistent-volumes.

[12] Pods. https://kubernetes.io/docs/concepts/workloads/pods.

[13] Service. https://kubernetes.io/docs/concepts/services-networking/service.

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

[15] Why we can't have nice things: implementing transactions in Kubernetes. https://kcsna2024.sched.com/event/1nSjl.

[16] 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.

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

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

[19] Chaos mesh — a solution for system resiliency on kubernetes. https://dzone.com/articles/chaos-mesh-a-chaos-engineering-solution-for-system, 2020.

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

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

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

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

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

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

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

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

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

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

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

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

[32] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[50] Barletta, M., Cinque, M., Di Martino, C., Kalbarczyk, Z. T., and Iyer, R. K. 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).

[51] Benson, T., Akella, A., and Maltz, D. Unraveling the Complexity of Network Management. In *Proceedings of the 6th USENIX Symposium on Networked System Design and Implementation (NSDI'09)* (Apr. 2009).

[52] Benson, T., Akella, A., and Shaikh, A. 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).

[53] Bianchini, R., Martin, R. P., Nagaraja, K., Nguyen, T. D., and Oliveira, F. Human-Aware Computer System Design. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS-X)* (June 2005).

[54] Bornholt, J., Joshi, R., Astrauskas, V., Cully, B., Kragl, B., Markle, S., Sauri, K., Schleit, D., Slatton, G., Tasiran, S., Van Geffen, J., and Warfield, A. 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).

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

[56] Brubaker, C., Jana, S., Ray, B., Khurshid, S., and Shmatikov, V. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP'14)* (May 2014).

[57] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, Omega, and Kubernetes. *Communications of the ACM 59*, 5 (May 2016), 50–57.

[58] Canini, M., Venzano, D., Perešíni, P., Kostić, D., and Rexford, J. A NICE Way to Test OpenFlow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)* (Apr. 2012).

[59] Cebula, M., and Sherrod, B. 10 Weird Ways to Blow Up Your Kubernetes. In *KubeCon North America* (Nov. 2019).

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

[61] Chen, Q., Wang, T., Legunsen, O., Li, S., and Xu, T. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)* (Nov. 2020).

[62] Chen, Y., Pan, J., Clark, J., Su, Y., Zheutlin, N., Bhavya, B., Arora, R., Deng, Y., Jha, S., and Xu, T. 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).

[63] Chen, Y., Shetty, M., Somashekar, G., Ma, M., Simmhan, Y., Mace, J., Bansal, C., Wang, R., and Rajmohan, S. AIOpsLab: A Holistic Framework to Evaluate AI Agents for Enabling Autonomous Clouds. In *Proceedings of the 8th Annual Conference on Machine Learning and Systems (MLSys'25)* (May 2025).

[64] Chen, Y., Sun, X., Nath, S., Yang, Z., and Xu, T. 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).

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

[66] Drosos, G.-P., Sotiropoulos, T., Alexopoulos, G., Mitropoulos, D., and Su, Z. When Your Infrastructure Is a Buggy Program: Understanding Faults in Infrastructure as Code Ecosystems. *Proc. ACM Program. Lang.* (Oct. 2024).

[67] Ganatra, V., Parayil, A., Ghosh, S., Kang, Y., Ma, M., Bansal, C., Nath, S., and Mace, J. 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).

[68] Gu, J. T., Sun, X., Tang, Z., Wang, C., Vaziri, M., Legunsen, O., and Xu, T. Acto: Push-Button End-to-End Testing for Operation Correctness of Kubernetes Operators. *USENIX ;login:* (Aug. 2024).

[69] Gu, J. T., Sun, X., Zhang, W., Jiang, Y., Wang, C., Vaziri, M., Legunsen, O., and Xu, T. 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).

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

[71] Gunawi, H. S., Rubio-González, C., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Liblit, B. EIO: Error Handling is Occasionally Correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)* (Feb. 2008).

[72] Gunawi, H. S., Suminto, R. O., Sears, R., Golliher, C., Sundararaman, S., Lin, X., Emami, T., Sheng, W., Bidokhti, N., McCaffrey, C., Srinivasan, D., Panda, B., Baptist, A., Grider, G., Fields, P. M., Harms, K., Ross, R. B., Jacobson, A., Ricci, R., Webb, K., Alvaro, P., Runesha, H. B., Hao, M., and Li, H. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)* (Feb. 2018).

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

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

[75] Hassan, M. M., Salvador, J., Santu, S. K. K., and Rahman, A. State Reconciliation Defects in Infrastructure as Code. In *Proceedings of the ACM on Software Engineering* (July 2024).

[76] Huang, L., Magnusson, M., Muralikrishna, A. B., Estyak, S., Isaacs, R., Aghayev, A., Zhu, T., and Charapko, A. Metastable Failures in the Wild. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[77] Huang, P., Guo, C., Lorch, J. R., Zhou, L., and Dang, Y. 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).

[78] Huang, P., Guo, C., Zhou, L., Lorch, J. R., Dang, Y., Chintalapati, M., and Yao, R. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS-XVI)* (May 2017).

[79] Iyer, R., Ma, J., Argyraki, K., Candea, G., and Ratnasamy, S. The Case for Performance Interfaces for Hardware Accelerators. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS-XIX)* (2023).

[80] Jha, S., Arora, R., Watanabe, Y., Yanagawa, T., Chen, Y., Clark, J., Bhavya, B., Verma, M., Kumar, H., Kitahara, H., Zheutlin, N., Takano, S., Pathak, D., George, F., Wu, X., Turkkan, B. O., Vanloo, G., Nidd, M., Dai, T., Chatterjee, O., Gupta, P., Samanta, S., Aggarwal, P., Lee, R., Murali, P., Ahn, J.-w., Kar, D., Rahane, A., Fonseca, C., Paradkar, A., Deng, Y., Moogi, P., Mohapatra, P., Abe, N., Narayanaswami, C., Xu, T., Varshney, L. R., Mahindru, R., Sailer, A., Shwartz, L., Sow, D., Fuller, N. C. M., and Puri, R. 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).

[81] Kumar, H., and Šafránek, J. Storage on Kubernetes - Learning From Failures. In *KubeCon North America* (Nov. 2019).

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

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

[84] Leners, J. B., Gupta, T., Aguilera, M. K., and Walfish, M. 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).

[85] Leners, J. B., Wu, H., Hung, W.-L., Aguilera, M. K., and Walfish, M. 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).

[86] Li, A., Lu, S., Nath, S., Padhye, R., and Sekar, V. ExChain: Exception Dependency Analysis for Root Cause Diagnosis. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)* (Apr. 2024).

[87] Liu, B., Lim, G., Beckett, R., and Godfrey, P. B. Kivi: Verification for Cluster Management. In *Proceedings of the*

2024 USENIX Annual Technical Conference (ATC'24) (July 2024).

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

[89] LOU, C., JING, Y., AND HUANG, P. 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).

[90] LOU, C., PARIKESIT, D. S., HUANG, Y., YANG, Z., DIWANGKARA, S., JING, Y., KISTIJANTORO, A. I., YUAN, D., NATH, S., AND HUANG, P. 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).

[91] MEHTA, S., BHAGWAN, R., KUMAR, R., ASHOK, B., BANSAL, C., MADDILA, C., BIRD, C., ASTHANA, S., AND KUMAR, A. 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).

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

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

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

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

[96] PANDA, B., SRINIVASAN, D., KE, H., GUPTA, K., KHOT, V., AND GUNAWI, H. S. 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).

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

[98] RAHMAN, A., FARHANA, E., PARNIN, C., AND WILLIAMS, L. 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).

[99] RAHMAN, A., PARNIN, C., AND WILLIAMS, L. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)* (May 2019).

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

[101] RIDGE, T., SHEETS, D., TUERK, T., GIUGLIANO, A., MADHAVAPEDDY, A., AND SEWELL, P. 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).

[102] SHETTY, M., CHEN, Y., SOMASHEKAR, G., MA, M., SIMMHAN, Y., ZHANG, X., MACE, J., VANDEVOORDE, D., LAS-CASAS, P., GUPTA, S. M., NATH, S., BANSAL, C., AND RAJMOHAN, S. Building AI Agents for Autonomous Clouds: Challenges and Design Principles. In *Proceedings of 15th ACM Symposium on Cloud Computing (SoCC'24)* (Nov. 2024).

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

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

[105] SUN, X., GU, J. T., RIVERA, C., CHAJED, T., HOWELL, J., LATTUADA, A., PADON, O., SURESH, L., SZEKERES, A., AND XU, T. Anvil: Building Kubernetes Controllers That Do Not Break. *USENIX ;login:* (June 2024).

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

[107] SUN, X., LUO, W., GU, J. T., GANESAN, A., ALAGAPPAN, R., GASCH, M., SURESH, L., AND XU, T. Sieve: Chaos Testing for Kubernetes Controllers. *USENIX ;login:* (Nov. 2024).

[108] SUN, X., MA, W., GU, J. T., MA, Z., CHAJED, T., HOWELL, J., LATTUADA, A., PADON, O., SURESH, L., SZEKERES, A., AND XU, T. Anvil: Verifying Liveness of Cluster Management Controllers. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)* (July 2024).

[109] SUN, X., SURESH, L., GANESAN, A., ALAGAPPAN, R., GASCH, M., TANG, L., AND XU, T. Reasoning about mod-

ern datacenter infrastructures using partial histories. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)* (May 2021).

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

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

[112] TANG, L., BHANDARI, C., ZHANG, Y., KARANIKA, A., JI, S., GUPTA, I., AND XU, T. 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).

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

[114] XING, J., HSU, K.-F., XIA, Y., CAI, Y., LI, Y., ZHANG, Y., AND CHEN, A. Occam: A Programming System for Reliable Network Management. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys'24)* (Apr. 2024).

[115] XU, Q., GAO, Y., AND WEI, J. 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).

[116] XU, T., JIN, X., HUANG, P., ZHOU, Y., LU, S., JIN, L., AND PASUPATHY, S. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).

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

[118] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. 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).

[119] YUAN, D., LUO, Y., ZHUANG, X., RODRIGUES, G., ZHAO, X., ZHANG, Y., JAIN, P. U., AND STUMM, M. 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).

[120] ZHAI, E., CHEN, A., PISKAC, R., BALAKRISHNAN, M., TIAN, B., SONG, B., AND ZHANG, H. Check before You Change: Preventing Correlated Failures in Service Updates. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)* (Feb. 2020).

[121] ZHANG, J., RENGANARAYANA, L., ZHANG, X., GE, N., BALA, V., XU, T., AND ZHOU, Y. 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).

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

[123] ZHENG, N., QIAO, T., LIU, X., AND JIN, X. 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).

A More Details of OAT

A.1 Testing State Transitions

OAT models an operator's input as a pair of an application's existing state and its desired state [69]. 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 application 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 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 application operations (e.g., reconfiguration), OAT triggers state transitions by declaring new desired states. Based on Finding 11 (§4.5.3), the majority of application interaction failures require changing the desired states. Among these, a majority require changing application-specific properties. The key challenge is to effectively synthesize application-specific properties for generating desired state declarations.
- To test operations which are only triggered when the application needs to handle a faulty state (e.g., failover and recovery operations), OAT triggers state transitions by perturbing the existing application state via fault injection. OAT checks if the application 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 application 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 application-specific properties (§A.2). Second, it generates end-to-end tests by systematically combining these declarations with fault injection (§A.3). Finally, it executes each test in local Kubernetes clusters and validates their outcomes using automatic oracles (§A.4). OAT reports test failures along with the state transition for reproduction.

A.2 Generating State Declarations

OAT generates desired state declarations that effectively exercise operator-application interactions through diverse state transitions. Typically, application-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 application, and does not explore valid state transitions.

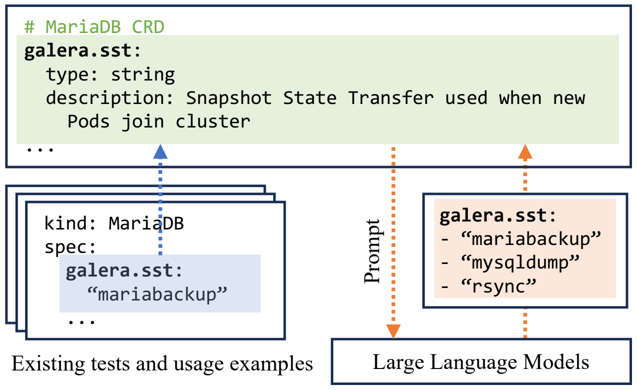


Figure 5: **Synthesizing values for the `galera.sst` property using existing tests and usage examples, and LLMs.**

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

Developer-written values. Mature operator projects already contain unit tests and usage examples that instantiate desired state declarations with meaningful property values. In Kubernetes, the desired states are described by properties of Custom Resources (CRs) [5] 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 5, 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 5, the LLM generates additional valid values for `galera.sst`, such as "`mysqldump`" and "`rsync`,", which extend coverage beyond what appears in developer tests. Figure 6 shows the LLM prompt used for the property. Although LLM-

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

generated values may occasionally be invalid, OAT filters these cases during testing: invalid values are rejected by the operator or application 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 (§B.2). OAT prioritizes developer-written values if available, and uses LLM-generated values otherwise.

A.3 Perturbing Existing Application State

Application-operator interaction failures arise not only from transitions starting from healthy states but also from error states caused by external events (Finding 12, §4.5.3). OAT injects application faults to perturb the existing state, to test whether operators can: (1) recover the application back to desired states from error states, and (2) tolerate faults that occur during state transitions and eventually drive the application 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 application container crashes to drive the existing state to an error state. OAT then removes the fault, and checks that the operator successfully reconciles the application 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 application 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 application state eventually matches the new desired state. OAT uses network delays to test operation ordering dependencies (see §4.1.2) and network partitions to test operation error handling (see §4.4).

A.4 Test Oracles

OAT employs automatic oracles to check whether the application state after the state transition matches the desired state in each test. Similar to previous works [69, 106], OAT checks for explicit or implicit state mismatches by observing the application state through the Kubernetes API objects. Kubernetes APIs provide limited visibility into application internal state and cannot detect silent failures where applications 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 application state only after the transition completes cannot detect transient availability violations. To address these limitations, OAT can leverage optional user-provided utilities that expose application internal state and monitor availability throughout transitions, significantly enhancing bug detection capability.

State monitors. OAT benefits from application state monitors which can check applications' 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.

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

A.5 Implementation Details

We implemented OAT for Kubernetes operators in approximately 1,200 lines of Python code, on top of the Acto framework [68, 69]. 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 [19] to inject faults into the applications. 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 application workloads (§5.1.3) are also provided through this interface.

B More Details on OAT Evaluation

B.1 Efficiency and Cost

Table 8 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 application-specific properties is low (Table 8). 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.

B.2 False Positives

OAT reports no false alarms. Most operation commands generated by OAT declare valid desired application states. A reliable operator must reconcile the cloud application 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 application state (which is known as *misoperations*), we expect a reliable operator not to crash or drive the cloud application to an error state (e.g., application 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 application, OAT detects a *misoperation vulnerability* [69], as discussed in §5.3. Note that misoperation vulnerabilities are considered serious reliability threats [69, 116, 117], as they are commonly introduced by human mistakes or AI hallucinations (if AI is used to interfere with these operators [62, 102]).

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

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 8: **Detailed information on the tests run by OAT.** "Prop." refers to the application-specific properties; "Config" refers to the unique application configurations OAT generated. "Test" refers to tests run by OAT, each of which realizes a failure pattern in Table 6.

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 application-specific properties and application configuration.