Automatic Reliability Testing for Cluster Management Controllers

Xudong Sun[†], Wenqing Luo[†], Jiawei Tyler Gu[†], Aishwarya Ganesan[‡], Ramnatthan Alagappan[‡], Michael Gasch[‡], Lalith Suresh[‡], Tianyin Xu[†]

[†]University of Illinois at Urbana-Champaign [‡]VMware

Abstract

Modern cluster managers like Borg, Omega and Kubernetes rely on the *state-reconciliation* principle to be highly resilient and extensible. In these systems, all cluster-management logic is embedded in a loosely coupled collection of microservices called *controllers*. Each controller independently observes the current cluster state and issues corrective actions to converge the cluster to a desired state. However, the complex distributed nature of the overall system makes it hard to build reliable and correct controllers – we find that controllers face myriad reliability issues that lead to severe consequences like data loss, security vulnerabilities, and resource leaks.

We present Sieve, the first automatic reliability-testing tool for cluster-management controllers. Sieve drives controllers to their potentially buggy corners by systematically and extensively perturbing the controller's view of the current cluster state in ways it is expected to tolerate. It then compares the cluster state's evolution with and without perturbations to detect safety and liveness issues. Sieve's design is powered by a fundamental opportunity in state-reconciliation systems – these systems are based on state-centric interfaces between the controllers and the cluster state; such interfaces are highly transparent and thereby enable fully-automated reliability testing. To date, Sieve has efficiently found 46 serious safety and liveness bugs (35 confirmed and 22 fixed) in ten popular controllers with a low false-positive rate of 3.5%.

1 Introduction

Modern cluster managers like Kubernetes [11], Borg [80], Twine [77], Omega [72], and vSphere [20] break down cluster-management logic into a fleet of microservices, called *controllers* [27]. For example, in Kubernetes, *all* the cluster-management logic is encoded in different controllers. Today, thousands of controllers are implemented by commercial vendors and open-source communities to extend Kubernetes with new capabilities [42, 68, 74, 78]. Controllers manage everything from application lifecycles (e.g., provisioning, upgrades, autoscaling) to stateful services, storage, networking, and integrations with cloud providers [41, 53, 57, 60, 71].

These cluster managers follow the *state-reconciliation principle* for resilience and extensibility [7, 27]. In this design, each controller continuously monitors a subset of the cluster state and reconciles the *current* state of the cluster to match a *desired* state. The cluster state is typically hosted in a logically centralized, highly available data store (e.g., etcd or ZooKeeper). In Kubernetes, entities like pods, nodes, volumes, and application instances are represented as objects in the cluster state. An auto-scaling controller might thereby monitor an application-group object for the number of currently active replicas and scale it to match the desired replica count. The design allows cluster managers to be 1) *resilient*: controllers can independently fail and pick up from where they left off, and 2) *extensible*: supporting a new feature or application is a matter of adding a custom controller that manages a set of custom objects as part of the cluster state.

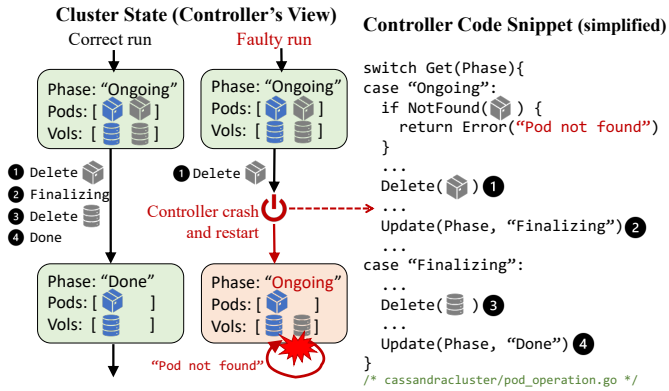

Figure 1: **A bug in a Cassandra controller detected by our tool, Sieve [30].** The controller cannot recover from an intermediate state introduced by Sieve using a crash. As a consequence, the controller cannot auto-scale the Cassandra cluster and leaks storage resources. The bug has been fixed. The code snippet is significantly simplified for clarity; the real code spans 70+ functions and 2,000+ lines of Go.

Despite the importance and prevalence of custom controllers, ensuring their reliability is challenging. Controllers run within complex, dynamic, and distributed environments. They must safely drive the system to desired states while tolerating unexpected failures, network interruptions, and asynchrony issues. If controllers are not robust to these circumstances, they lead to severe consequences such as application outages, data loss, and security issues. Buggy controllers have indeed caused many real-world problems [31, 38, 51, 52].

For example, Figure 1 shows a bug in a Kubernetes controller for managing Cassandra [30]. The bug prevents the Cassandra cluster from auto-scaling and leaks storage resources (decommissioned volumes in gray are never deleted). This is because the controller lacks crash safety – it fails to recover from an intermediate state due to a crash between deleting a Cassandra pod and updating the `Finalizing` phase.

The above crash-safety bug is only one of the myriad kinds of reliability issues that affect controllers. We find that controllers also experience bugs caused by state inconsistencies due to asynchrony effects and bugs caused by uncoordinated concurrent interactions between controllers. Existing testing techniques are either too specialized for certain types of bugs or require expert guidance in the form of formal specifications or carefully-crafted test inputs (§8). We instead seek a solution that is broadly applicable across controllers and is capable of *automatically* detecting a wide range of bugs.

Contributions. In this paper, we present Sieve, the first automatic reliability-testing technique for cluster-management controllers. Sieve drives unmodified controllers to their potentially buggy corners by systematically and extensively perturbing the controller's view of the cluster state in ways it is expected to tolerate. Sieve then compares the cluster state's evolution with and without perturbations to automatically detect safety and liveness issues.

Sieve is highly usable. It does not require 1) formal specifications of the controller or the cluster manager, 2) hypotheses about vulnerable regions in the code where bugs may lie, or 3) highly specialized test inputs. It does not rely on expert-written assertions either. Sieve requires only a manifest for building the controller image and basic test workloads. Sieve's testing is then fully automatic. This degree of usability is key to making reliability testing broadly accessible to the rapidly increasing number of custom controllers.

Sieve is powered by a fundamental opportunity in state-reconciliation systems – controllers interact with the cluster state via *state-centric interfaces*. State-centric interfaces perform semantically simple operations on the cluster state (e.g., reads and writes) and deliver notifications about cluster-state changes; the objects that flow through the interfaces typically have a uniform schema. Therefore, state-centric interfaces are highly introspectable and hence an ideal vantage point to observe and perturb a controller's view of the cluster state.

Sieve leverages the fact that a controller's actions are strictly a function of its view of the current cluster state. We thus test a controller by exhaustively introducing state perturbations through failures, delays, and reconfigurations. These are circumstances that reliable controllers are expected to tolerate. Currently, Sieve supports three typical perturbation patterns that expose controllers to 1) intermediate states (Figure 1), 2) stale states (or past cluster states), and 3) unobserved states due to missing some cluster state transitions (§3.1).

For each pattern, Sieve automatically generates test plans that cover all possible perturbations during an execution of the controller under test. Test-plan generation is based on analyzing a controller's behavior and the cluster-state evolution during reference executions. Sieve effectively avoids redundant and futile test plans to maximize test efficiency.

Sieve automatically detects buggy controller behavior using differential test oracles that compare the cluster-state transitions with and without perturbations. This comparison is feasible because a controller's behavior is reflected in the sequence of cluster-state transitions. The differential oracles are often more effective than searching for errors in logs and more comprehensive than human-written assertions (§3.6).

Key results. We implemented Sieve for Kubernetes controllers. Sieve requires only a manifest for building the controller image and basic workloads (e.g., a scale-up-and-down workload for an autoscaling feature). Sieve's testing is then fully automatic. We evaluated Sieve on ten popular open-source controllers of various kinds, from either commercial vendors or official projects. Sieve found 46 new bugs in total, among which 35 have been confirmed (22 fixed) after we reported them. Notably, these are deep semantic bugs that Sieve detected without any expert guidance. The bugs have severe consequences, including application outages, security vulnerabilities, resource leaks, and data loss. Sieve is highly efficient—all controllers could be tested in under seven hours on a cluster of 11 machines, representing a typical nightly test. Sieve also has a very low false-positive rate of 3.5%, making its testing results trustworthy.

Summary. The paper makes four main contributions:

- We present the first automatic reliability-testing technique for state-reconciliation systems: exhaustively perturbing the controller's view of cluster states and using differential oracles on the cluster state evolution to detect bugs.
- We design and implement Sieve, a system that uses our proposed technique to automatically test *unmodified* cluster-management controllers in Kubernetes.
- Sieve has already improved reliability for ten popular open-source controllers by virtue of bugs it found that were then fixed by developers. It is practical to run Sieve regularly.
- We have made Sieve publicly available at `https://github.com/sieve-project/sieve`, with instructions to reproduce all discovered bugs.

2 Background and Motivation

Modern cluster management and control plane designs follow the state-reconciliation pattern, where control loops reconcile the current state of the cluster to conform to a desired state. Kubernetes, like its predecessors Borg and Omega, follows the idea of reconciliation control loops for resiliency [27]. Similarly, vSphere [81] and NSX [82] continuously monitor and correct deviations from declaratively-specified desired

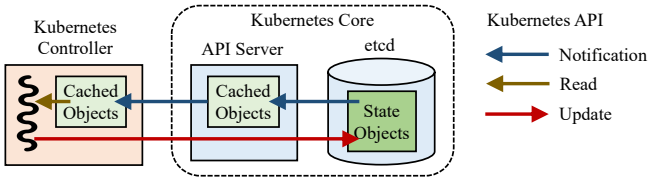

Figure 2: **Interactions between a controller and other Kubernetes components and the state-centric APIs.**

states to manage virtual machines and networks. These systems rely on a clean separation between the cluster state and the control plane logic [27]; the state is represented as mere data (e.g., JSON objects), and the control plane logic queries and manipulates the state programmatically.

We now give a brief overview of state reconciliation and cluster-management controllers. We also present the urgent need for automated reliability testing and our insights.

State reconciliation by example. We use Kubernetes as a representative example to present the basics of state reconciliation. Figure 2 illustrates Kubernetes' architecture. Kubernetes' core comprises an ensemble of *API servers* and a highly available, strongly consistent data store (etcd) that houses the *cluster state*. The cluster state is represented by a collection of *objects*. Every entity in the cluster has a corresponding object in the cluster state, including pods, volumes, nodes, and groups of applications. All other components in Kubernetes interact with the cluster state via API servers.

All cluster-management logic is encoded in *controllers* that are clients of the API servers. The controllers continuously monitor a part of the cluster state and perform state reconciliation whenever the current state does not match the desired state. The controllers perform reconciliation by querying and manipulating the state objects via a client library that exposes a state-centric interface. This interface provides notifications, reads, and writes involving the cluster state objects.

This design enables Kubernetes to be highly extensible: supporting a new application or feature is a matter of adding a new controller and a corresponding set of custom object types to the cluster state; it does not require changes to the client library or interface. The design also allows controllers to be loosely coupled, which improves resilience: controllers can independently fail and new controller instances can resume reconciliation without fail-over logic.

Figure 3 shows how a collection of controllers coordinate in a loosely coupled manner. To deploy a ZooKeeper cluster running on Kubernetes, the user creates a ZooKeeper object which specifies the desired state of the ZooKeeper cluster (e.g., replica count, version, storage size) via the Kubernetes command-line tool. The ZooKeeper controller receives a notification that a ZooKeeper object was created. To drive the system to the desired state, it updates the cluster state by creating a StatefulSet object (an abstraction to run stateful applications). Then, a StatefulSet controller is subsequently

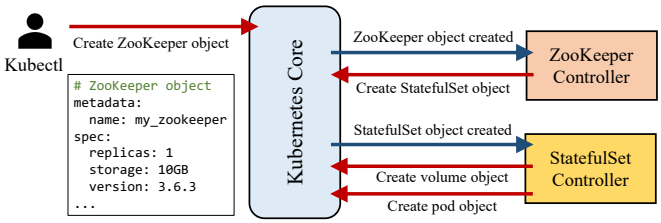

Figure 3: **The workflow of deploying ZooKeeper on Kubernetes using a ZooKeeper controller.**

notified about the StatefulSet object being created, which in turn creates pod and volume objects to run the containerized ZooKeeper nodes. While not shown in the figure for brevity, this subsequently leads to more controllers like a scheduler, a storage controller, and worker nodes being activated to bring up the actual containers and volumes. Similarly, if the user then edits the desired state of the ZooKeeper object (e.g., the number of replicas), it triggers a similar sequence of reconciliations by different controllers, as each tries to make minor adjustments to get to its appropriate desired state.

The need for automated reliability testing. Kubernetes' extensibility has led to a thriving ecosystem with *thousands* of domain-specific controllers implemented by commercial vendors and open-source communities [41, 42, 53, 60, 68, 71, 74]. For example, OpenShift, an enterprise Kubernetes platform from Red Hat, provides 130+ custom controllers that extend Kubernetes [17]. All these controllers represent critical infrastructure, making their correctness paramount. As shown by many real-world problems [31, 38, 51, 52] and our evaluation results, designing and implementing reliable controllers is challenging – many popular, mature controllers misbehave under faults, delays, and asynchrony with severe consequences.

However, controller reliability is notoriously hard to ensure. A developer faces the fundamental challenge of 1) anticipating all possible *views* of the cluster state at the controller (compounded by asynchrony) and 2) safely reconciling to the required desired states from any of these points. We observe that manually-written test suites do not sufficiently test a controller's reliability (§6).

Unfortunately, existing testing techniques are either too specialized for certain bug types (e.g., crash-recovery bugs or concurrency bugs) and cannot address the broad range of controller bugs; or require expert guidance in terms of formal specifications of the system, crafted heuristics, or hypotheses on vulnerable code regions (§8). We seek a solution that is easy to use and broadly applicable to unmodified controllers.

Our insight. To overcome the above challenges, we 1) automatically and extensively perturb an unmodified controller's view of the cluster states in ways it is expected to tolerate, and 2) automatically flag safety and liveness issues using differential oracles that compare the evolution of cluster states with and without perturbations. This degree of automation

and unintrusiveness is enabled by the fundamental nature of state-reconciliation systems. That is, these systems often have a simple and highly introspectable state-centric interface with which controllers interact with the cluster state. Such interfaces essentially do no more than reads and writes, or receive notifications regarding state-object changes. All objects share a common schema, which makes any arbitrary object highly introspectable. For example, all objects in Kubernetes have an identical set of fields representing their metadata. This enables a degree of automation that is hard to achieve otherwise.

3 Sieve Design

Sieve is an automatic reliability testing tool for cluster management controllers. It checks whether the controllers under test can correctly operate the system under common perturbations (due to unexpected faults and inherent asynchrony) and detects bugs that lead to safety and liveness issues at the development time. Sieve is automatic—it tests unmodified controllers and does not rely on formal specifications or controller-specific assertions. Sieve is effective—it focuses on well-defined, highly-targeted perturbations that reliable implementations are required to tolerate.

Sieve perturbs the controller's view of the cluster state based on three broad patterns that expose the controller to 1) intermediate states, 2) stale states, and 3) unobserved states. We discuss the three patterns and their rationales in §3.1. Note that these are not the only patterns in which faults can occur, but cover a broad range of faults that a component in a distributed system is expected to handle gracefully. Sieve can be extended to incorporate other patterns in the future.

Sieve tests controllers with the following workflow:

- *Collecting reference traces* (§3.2). Sieve starts by learning how a controller behaves in the absence of faults (under test workloads) and records the state transitions in reference traces. To do so, it instruments the state-centric interfaces used by the controller to interact with the cluster state.
- *Generating test plans* (§3.3). Sieve then analyzes the reference traces to generate *test plans*. A test plan describes a concrete perturbation. The test plan specifies *what* faults to inject and *when* to inject them to effectively drive the controller to see a target cluster state.
- *Avoiding ineffective test plans* (§3.4). To achieve high test efficiency, Sieve prunes redundant or futile test plans. For example, it avoids a test plan if it is clear that it cannot causally lead to a target cluster state.
- *Executing test plans* (§3.5). Sieve executes each test plan using a test coordinator. The test coordinator monitors the cluster-state transitions during testing and injects the specified faults according to the test plan's specification.
- *Checking test results* (§3.6). Sieve has generic, effective, differential oracles to automatically check test results. The oracles detect buggy controller behavior by comparing the cluster-state evolution between the reference and test runs.

Sieve deals with non-deterministic elements of the cluster state during testing to minimize their impact on test plan generation and test oracles (§3.7). Specifically, Sieve identifies non-deterministic state objects and fields and excludes them.

Usage. To use Sieve, one needs to provide two inputs: 1) a manifest that specifies how to build and deploy the controller under test, and 2) a set of test workloads that exercise end-to-end behavior of the controller under test. The two inputs are mostly available in mature controller projects, as they are needed for controller development and deployment. In our experience, finding them is straightforward.

3.1 Perturbing A Controller's View of The State

Sieve operates under the assumption that a controller follows the state-reconciliation principle, which receives a sequence of *notifications* about the changes to the cluster states and outputs a corresponding sequence of *updates* to the cluster states. Sieve aims to affect the outputs of a controller by perturbing its view of the cluster state. These perturbations are produced by injecting targeted faults (e.g., crashes, delays, and connection changes) when specific cluster-state changes (*triggering conditions*) happen.

Notably, the perturbation strategy allows Sieve to *decouple policy from mechanism*. The decoupling makes it easy to extend existing policies or add new policies by orchestrating the underlying perturbation mechanisms. Specifically, a policy defines a view Sieve exposes to the controller at a particular condition, while the mechanism specifies how to inject faults to create the view. Sieve automatically generates test plans for each policy; each test plan introduces a concrete perturbation based on a specification of a triggering condition and a fault to inject when that condition happens.

Sieve currently supports three patterns to perturb a controller's view. Crucially, these perturbations drive a controller to states that it is expected to tolerate. They represent valid inconsistencies in the view that a controller could see due to common faults as well as the inherent asynchrony of the overall distributed system. Over time, we hope to add more perturbation patterns.

Intermediate states. Intermediate states occur when controllers fail in the middle of a reconciliation before finishing all the state updates they would have otherwise issued. After recovery (e.g., Kubernetes automatically starts a new instance of a crashed controller), the controller needs to resume reconciliation from the intermediate state left behind.

Figure 4 illustrates how Sieve tests the official RabbitMQ controller with intermediate-state perturbations and reveals a new bug. The test workload attempts to resize the storage volume from 10GB to 15GB. The resizing is implemented with two updates: 1) updating `VolCur` to 15GB; 2) updating

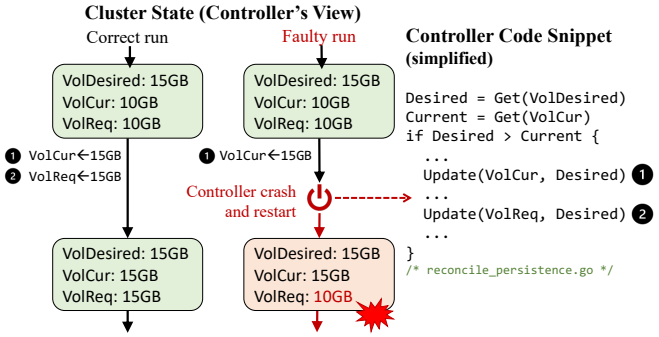

Figure 4: **An intermediate-state bug in a RabbitMQ controller detected by Sieve [70].** The controller fails to recover from the intermediate state introduced by Sieve; the controller does not successfully resize the storage volume.

`VolReq` to 15GB which triggers Kubernetes to resize the volume. The controller issues updates when `VolCur` is smaller than the desired volume size. During testing, Sieve crashes the controller between the two updates, which creates an intermediate state where `VolCur` is updated, but `VolReq` is not. The controller cannot recover from the intermediate state and the resizing never succeeds. The bug has been fixed with 700+ lines of Go code to revamp the volume resizing logic. In addition, the developers added eight new tests along with the fix to exercise how the controller handles different intermediate states, which is what Sieve performs automatically.

Stale states. Controllers often operate on stale states, due to asynchrony and the extensive uses of caches for performance and scalability [26]. As shown in Figure 2, controllers do not directly interact with the strongly consistent data stores, but are connected with API servers. The states cached at API servers could be stale due to delayed notifications. Controllers are expected to tolerate stale views that lag behind the latest states maintained in the data store.

Tolerating stale views correctly is nontrivial. For example, a Kubernetes controller's view may "time travel" to a state it observed in the past. Time traveling occurs when there are multiple API servers operating in a high-availability setup, when the controller reconnects to a stale API server that has not yet seen some updates to the cluster state. The reconnection can be triggered by failover, load balancing, or reconfigurations. Controllers are expected to recognize the stale state [18], instead of treating it as a new, unseen state.

Figure 5 illustrates how Sieve tests Percona's MongoDB controller with stale-state perturbation and reveals a new bug that leads to both application outages and data loss. To support graceful MongoDB cluster shutdowns, the controller waits to see a non-nil deletion timestamp (`DeletionTS`) field attached to the state object representing the MongoDB cluster (a common practice to give systems time to react to an impending deletion [23]). When the controller sees this change, it deletes all the pods and volumes of the MongoDB cluster.

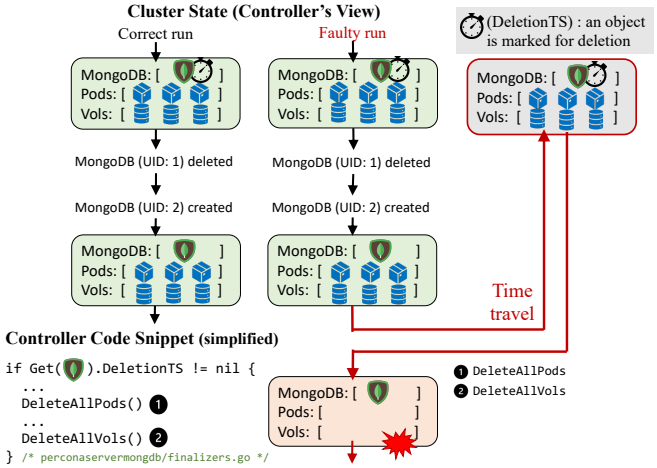

Figure 5: **A stale-state bug in a MongoDB controller detected by Sieve [43].** The controller experiences a "time-travel" and observes a stale state. It makes wrong reconciliation action based on the stale state (deleting all the pods and volumes) which leads to application outages and data loss.

Sieve drives the controller to mistakenly delete a live MongoDB cluster by introducing a time-travel perturbation. With a workload that first shuts down a MongoDB cluster and then recreates a new instance of the same cluster, Sieve waits till the cluster is recreated and then introduces a time-travel perturbation. The perturbation causes the controller to see the deletion timestamp being applied to the *already-deleted cluster*. Consequently, the controller mistakenly shuts down the newly created cluster. This revealed that the controller should be checking for the UIDs of clusters, not just their names.

Unobserved states. By design, controllers may not observe every cluster-state change in the system. The full history of changes made to the cluster state is prohibitively expensive to maintain and expose to clients [76]. Controllers are hence expected to be designed as *level-triggered* systems (opposed to being *edge-triggered*), i.e., a controller's decision must be based on the currently observable cluster state (level) [21], not on seeing every single change to the cluster state (edge).

Figure 6 illustrates how Sieve tests Instaclustr's Cassandra controller using unobserved-state perturbations and reveals a new bug that leads to resource leaks and service failures. The test workload first scales down and then scales up storage volumes of the Cassandra cluster. During scale-down, the controller removes volumes when it learns that the corresponding pods were marked for deletion (a non-nil deletion timestamp field is set on the pod object, similar to the previous example). The pods' lifecycles (including deletions) are managed by a built-in controller called a StatefulSet controller. Sieve pauses notifications to the Cassandra controller for a window such that it does not see these deletion marking events by the StatefulSet controller. This causes the Cassandra controller to not delete the corresponding volumes even though it has the

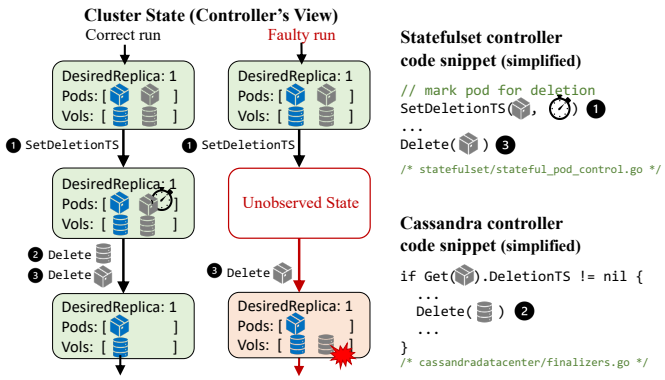

Figure 6: **An unobserved-state bug in a Cassandra controller detected by Sieve [29].** The controller misses a transient state where the pod has a non-nil deletion timestamp. It thus fails to delete the volumes, leaking storage resources. The bug also prevents new Cassandra pods from rejoining.

right information to make that call (i.e., its view has volumes created by it that do not have pods attached to them).

Hence, the volume never gets deleted, leaking the storage resource. The bug also prevents the controller from scaling the Cassandra cluster – newly-created pods try to reuse the dangling volumes and cannot rejoin using the cluster metadata already in them (as it represents a node that was decommissioned). The bug has been fixed by adding a pre-deletion hook – a coordination mechanism in Kubernetes that allows the Cassandra controller to complete the required cleanup operations before the pods can be deleted [9].

3.2 Collecting Reference Traces

Sieve starts by learning how a controller behaves in the absence of faults. To do so, Sieve interposes around the state-centric interfaces used by the controllers to interact with the cluster state. All modern cluster managers have unified, well-defined client libraries based on state-centric interfaces. Taking Kubernetes as an example, any interaction with the cluster state (exposed by the API servers) goes through a small, well-defined set of client APIs that read, modify, or receive notifications about state objects. They are used by every controller that interacts with the Kubernetes API servers. To support Kubernetes controllers, Sieve decorates 10 functions in the client library and this interposition is fully automated (§4).

With the interposition in place, Sieve learns every cluster-state change notification that the controller receives, as well as any reads and writes attempted by the controller to the cluster state or to the local cache of the cluster state maintained by the client. Sieve then runs each test workload supplied by the developer and collects the following two reference traces:

- *Controller trace.* A series of events observed via the interposition of client APIs, including notifications about state changes, entry and exits of each reconciliation cycle, and client-API invocation by the controller and their arguments.
- *Cluster state trace.* The initial cluster state and the sequence of state changes (object creations, modifications, and deletions), collected using public APIs of the cluster manager.

The controller trace is used for generating test plans (§3.3) and the cluster-state trace is used by test oracles (§3.6).

3.3 Test Plan Generation

Sieve generates a set of test plans for each test workload for which it has collected reference traces. Each test plan specifies a perturbation to inject during the workload.

A test plan is represented by a self-contained file that describes a test workload, a list of faults to inject during the workload run, and the triggering condition for when to inject each fault. Sieve currently supports several primitives that test plans can compose to introduce complex faults: 1) crash/restart a controller, 2) disconnect/reconnect a controller to an API server, 3) block/unblock a controller from processing events, and 4) block/unblock an API server from processing events. When an executed test plan reveals a bug, the test-plan file is sufficient to reproduce the bug.

Figure 7 shows a simplified test-plan file generated by Sieve. Each element in `faults` specifies the fault to inject (`faultType`) and the triggering conditions (`triggers`). Each element in `triggers` specifies a triggering condition, that causes the specified fault to be injected before or after a particular cluster state change if executed. A composite triggering condition can be specified in `compositeTrigger` by combining multiple conditions in `triggers` with boolean operators. For example, `t1 & t2` means the fault is only injected when both `t1` and `t2` are triggered. In Figure 7, `trigger1` is the only required condition to inject the fault. Similarly, composite faults can be constructed (e.g. crashing a controller after `t1` and restarting it after `t2`).

We now present the basic rules Sieve applies to compute test plans that exercise one of the three patterns in §3.1. We later describe how Sieve avoids ineffective test plans in §3.4. Optionally, one can customize patterns by implementing new rules or manually writing test plans.

Intermediate-state rule. For a controller, Sieve generates test plans that force all possible intermediate states and exposes them to the controller. To do so, Sieve analyzes the reference controller trace and marks the sequence of state updates made by the controller within each reconciliation loop. Concretely, for every reconciliation that issues multiple state updates, $U_1, U_2, ..., U_n$, Sieve generates one test plan per state update U_i, where Sieve crashes the controller after it issues U_i. When the controller restarts after the crash, it is presented with the target intermediate state.

Stale-state rule. For stale states, Sieve generates test plans that make the controller travel back in time and see stale states that it has already observed. Concretely, Sieve checks the controller trace for a notification-update pair (N, U), such that observing N results in an update U (see §3.4.1). It then

```
testWorkload: resizePVC
faults:
- faultType: crashController
  triggers:
  - triggerName: trigger1
    triggerAt: afterControllerIssues
    stateChange:
      beforeChange: 'VolCur:10GB'
      afterChange: 'VolCur:15GB'
  compositeTrigger: trigger1
```

Figure 7: **A test plan generated by Sieve.** This is a simplified view of the test-plan file that detected the bug in Figure 4. This test plan crashes the RabbitMQ controller right after the controller updates `VolCur` from 10GB to 15GB. Sieve learns every state change issued by the controller via the state-centric interfaces (e.g., `Update` in Table 1).

searches for a subsequent state-change notification N' which has a conflicting effect with U (e.g., U deletes an object and N' creates the same object). With time traveling, if the controller mistakenly issues U after seeing the stale state N, it could corrupt the newer cluster state as notified by N'.

Sieve generates test plans that 1) block a reserved API server to prevent it from advancing its own state after it sees N, 2) after the controller sees N', time-travel the controller to see N by reconnecting the controller to the reserved stale API server, and 3) unblock the stale API server; so, the introduced staleness is only transient—both the API server and the controller catch up eventually. We focus on deletions for U because they are destructive operations.

Unobserved-state rule. For unobserved states, Sieve generates plans that skip states that a controller might observe during normal executions, but could potentially miss in the presence of faults. Sieve checks the controller trace to find pairs of notifications (N, N') in which N' is the closest subsequent notification that cancels the effect of N. Sieve generates test plans that 1) block the controller to prevent it from seeing N, and 2) unblock the controller when N' arrives. Such a test plan causes the controller to miss cluster states from N (inclusive) up to N' (exclusive).

3.4 Avoiding Ineffective Test Plans

Sieve may potentially generate a large number of test plans using rules specified in §3.3. For example, in stale-state testing, Sieve might identify every notification the controller receives as a point to inject staleness, therefore generating test plans for every received notification. For example, the naïve rule above for stale states would generate 140,000+ test plans for the MongoDB controller in Figure 5. It is therefore key to prune ineffective test plans.

As a guiding principle, we prune a test plan if the test plan does not introduce an intermediate-, a stale- or an unobserved-state that can affect the controller's outputs, or the introduced state is identical to states introduced by other test plans. This naturally requires Sieve to have a clear notion of what input events affect the controller's outputs.

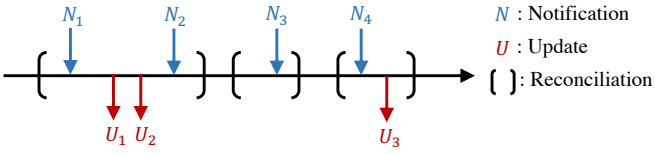

Figure 8: **Causality rules used by Sieve.** For simplicity, in this figure the object pertaining to each notification is immediately read by the controller. (N_1, U_1), (N_1, U_2), (N_3, U_3) and (N_4, U_3) are causally related according to the rules in §3.4.1. N_2 is *not* causally related to any update, given the earliest-reconciliation rule.

3.4.1 Pruning by Causality

If a controller makes an update U based on a notification N, we consider N and U to be causally related. We consider a pair (N, U) that is not causally-related to be irrelevant from a testing standpoint, because perturbing N will not affect U.

Inferring causality between events is generally a challenge in distributed systems. By focusing on the "narrow waist" of state-centric input and output events of the controller under test, we are able to design simple yet effective rules for Sieve to infer whether a pair (N, U) is causally related. These rules are lenient and only introduce false positives at best, but not false negatives. False positives increase testing times by generating redundant test plans, whereas false negatives risk reducing test coverage that could miss bugs. While causal tracing support [58] for Kubernetes is currently in its infancy [6], we might be able to leverage it in the future.

Sieve currently considers a pair (N, U) to be causally related if both the following conditions holds (Figure 8 exemplifies the two causality rules):

- *Read-before-update rule:* the object pertaining to N is read by the controller before it issues U;
- *Earliest-reconciliation rule:* N and U happen in the same or adjacent reconciliation cycles. The rationale is that controllers always issue updates relevant to N in the earliest possible reconciliation cycle after N is received.

For stale- and unobserved-state testing, Sieve only generates test plans involving a N if it has at least one causally-related U. We find pruning test plans by causality effective, especially when there are many notifications due to other activities irrelevant to the controllers under test.

3.4.2 Pruning Unsuccessful Updates

Sieve ignores any update U that does not change the current cluster state. Sieve checks whether an update U is successful based on whether U triggers a state change ΔS of the cluster state. This information is typically encoded in the return value of the U operation.

For stale-state testing, Sieve further ignores an update U that, if issued again, does not change any of the subsequent cluster states, (i.e., there does not exist an N' that is affected by U). Sieve checks whether the state objects updated by U would later be conflicted with N'.

The rationale for the above pruning is straightforward. If an update does not change the current cluster state, it is unlikely to cause new states in the execution. If an update U cannot affect any future cluster states, it would not perturb the controller's execution under the time-travel pattern either, i.e., if U is issued again, it would not corrupt any future state.

In practice, we found that many controllers issue unsuccessful updates that do not actually change the cluster state, including pathologically frequent ones caused by inefficient but benign behavior (see Section 5.2).

3.5 Test Plan Execution

Every test plan is executed by the Sieve test coordinator running in the testing cluster. The coordinator faithfully executes the test plan by running the test workload and injecting the faults specified in the test plan. Specifically, the test coordinator monitors state transitions of both the controller's view of the cluster state as well as the cluster state as seen by API servers. This is done based on the interposition described in §3.2; it allows the test coordinator to intercept and take actions (e.g., injecting faults) when state transitions happen. If the observed state transition matches the triggering condition ΔS specified in the test plan, the coordinator marks the condition as matched. The coordinator injects a fault (e.g., a controller crash) once all the corresponding conditions are matched. Most of the interposition and injection are done through the client APIs. But, for stale-state testing, the coordinator also needs to interpose at the API server (to make an API server stale).

As a concrete example, to execute the test plan in Figure 7, the test coordinator monitors every state transition issued by the RabbitMQ controller. The coordinator marks `trigger1` in the test plan as matched when it observes a state transition that updates `VolCur` from 10GB to 15GB. Since `trigger1` is the only required condition in the test plan, the coordinator injects a controller crash right after `trigger1` is matched. If the test plan specifies multiple faults, the coordinator injects them one by one according to the specified order.

The test coordinator also records the cluster states in a trace during testing, which will be compared with the reference cluster-state trace (§3.2) to detect buggy behavior.

3.6 Differential Test Oracles

Sieve has generic, effective oracles to automatically detect safety and liveness issues. The oracles detect buggy controller behavior based on the *cluster states* during and at the end of the test run. The goal is to validate that the testing traces are free of safety and liveness issues, in addition to monitoring anomalous controller behavior (e.g., crashes and hangs). Developers can also add domain-specific oracles.

In our experience, many buggy controller behaviors do not show immediate or obvious symptoms (e.g., crashes, hangs, and error messages). Instead, they lead to data loss, security issues, resource leaks, and unexpected application behavior which is hard to check with oracles typically used by prior art [55, 75, 84, 88, 89]; in our evaluation, only five (out of 46) bugs can be flagged by checking for exceptions or crashes.

We therefore develop differential test oracles that compare cluster states in a reference run versus those in test runs—with inconsistencies typically indicating buggy behavior. This methodology means we need to exclude nondeterministic states and state objects affected by the perturbation (§3.7).

We found that Sieve's differential oracles vastly outperform developer-written assertions in the test suites of the controllers we evaluated, because Sieve's oracles systematically examine all the state objects and their evolution during testing. It is challenging for developers to manually codify oracles that comprehensively consider the large number of relevant states.

Note that Sieve does implement regular error checks for obvious anomalies, including exceptions, error codes and timeouts. Sieve scans the controller's log and checks whether the controller encountered any unexpected exception (i.e., `panic` in Go). Sieve also checks whether the operations in the test workload return error codes or fail to complete on time.

3.6.1 Checking End States

Sieve systematically checks the end state after running a workload. Specifically, our oracles check the count of state objects by type and the field values of all the objects after accounting for nondeterminism (§3.7). It compares the end state of the test run versus the reference run. Sieve fails the test if it finds inconsistencies between the end states and prints human-readable messages to pinpoint inconsistencies.

Such checking is effective compared to the simpler assertions that we found in test suites for the controller projects we studied. For example, in an intermediate-state bug [46], the MongoDB controller fails to create an SSL certificate used for securing communications inside the MongoDB cluster. This causes the controller to fall back to insecure communications. Such security issues do not manifest in the form of crashes or error messages. Sieve however automatically catches the bug, because the certificate object in the faulty run does not exist in the cluster state, which is different from a normal run. The bug was detected by Sieve and confirmed by the developers.

We found that none of the 71 test cases shipped with the controller has an assertion that checks the certificate object, despite the fact that enabling TLS is recommended and is the default configuration [66]. We would not be able to repurpose the assertions in these test cases to catch this bug.

3.6.2 Checking State-Update Summaries

Besides the end state, Sieve also checks how the controller updates the cluster state over time. It does so by comparing summaries of constructive and destructive state updates for each object (e.g., `CREATE` and `DELETE` operations). Such checks are complementary to the end-state checks, because a correct end state does not imply that the controller behavior is always correct during the test. We find that buggy behavior can end in correct states (same as in the reference runs).

For example, in a stale-state bug [47], the XtraDB controller mistakenly deletes the front-end proxy (which routes user requests to the XtraDB cluster), causing service unavailability. After the staleness ends, the API server and the controller eventually catch up with updated states and recreate the proxy. In this case, the end state of the proxy in the test run is the same as in the reference, but the update that deleted the proxy in the test run is buggy. Sieve detects the bug by noting that the proxy pod receives 2 `CREATE` and 1 `DELETE` operations in the faulty run, but only 1 `CREATE` in the reference run.

In another intermediate-state bug [62], the NiFi controller fails to reload configuration files. The end state is the same as a normal run; however, in the faulty run, the controller did not restart the NiFi pod to reload the configuration. Sieve flags this by noting the NiFi pod receives a `CREATE` and a `DELETE` operation (to reload the configuration) in the normal run, but neither appears in the faulty run.

Note, comparing the *sequence* of state-update is unreliable and would lead to false alarms—the sequences are not strictly the same due to concurrent controller operations. The summaries instead are robust to different event orderings.

3.7 Dealing with Nondeterminism

The shape of a state object (the set of fields and their values) might be nondeterministic. This nondeterminism affects Sieve's test plan generation and the differential test oracles. We now describe how Sieve combats this problem.

All objects have identifying metadata (e.g., a type, namespace, and name). This is key for Sieve to identify two instances of the same object, both within a run (e.g., checking for conflicting operations in the stale-state rule) and across runs (e.g., comparing configurations of objects across runs).

Sieve identifies nondeterministic field values by running the test workloads without perturbation multiple times when generating reference runs, and then comparing the values of each field in each state object.

Objects whose identifying metadata is nondeterministic are excluded from test plan generation and subsequent steps, because Sieve cannot reliably match them across runs or setup triggering conditions for them. If other kinds of fields have nondeterministic values (typically IP addresses, timestamps, or even random port numbers), Sieve does not exclude the object but simply masks the field values. Note that these two rules still allow Sieve to spot unexpected changes to the set of fields on the object (e.g., missing deletion timestamp fields).

API	Component	Instrumentation
`reconcileHandler`	Client	Log entry and exit
`Get`, `List`	Client	Send objects to coordinator
`Create`, `Update`, `Patch`	Client	Send objects to coordinator
`Delete`, `DeleteAllOf`	Client	Send objects to coordinator
`HandleDeltas`	Client	Send objects to coordinator
`processEvent`	API server	Send objects to coordinator
`Get`, `List`	Client	Add delay
`processEvent`	API server	Add delay

Table 1: **Instrumentation performed by Sieve to monitor and perturb states.** The instrumentation is automated.

In addition, Sieve provides an API for Sieve users to exclude specific state objects or fields from test plans or oracles based on domain knowledge, if needed.

4 Implementation

We implement Sieve for Kubernetes controllers. Sieve uses `kind` [10] to run a Kubernetes cluster on a single machine, so every test plan can be run entirely on one machine. Sieve configures two API servers for stale-state testing. Sieve is implemented in 5,500 lines of Python code (for test plan generation and oracles) and 3,100 lines of Go code (for automated instrumentation and fault injection).

Sieve instruments 10 API methods, representing the state-centric interface, for monitoring and perturbing states (Table 1). Those methods are in Kubernetes client libraries [13, 14] and the API server. Sieve implements an automated procedure to instrument the 10 methods using `dst` [8] to work with different versions of Kubernetes client libraries and API servers. Sieve analyzes the syntax tree for each method to insert monitoring and fault-injection code. Sieve applies the instrumentation when building the controller image. Sieve does not need to analyze or instrument the controller code.

In Kubernetes, level-triggered controllers do not immediately read notifications when they arrive [21]. Instead, the controller first updates a locally-cached view of the state objects; the controller reads from this cache when it uses `Get` or `List` APIs to query the cluster state. In causality analysis (§3.4.1) Sieve needs to know whether a notification is read before an update. To do so, Sieve analyzes the state objects updated by each notification and those read by each `Get`/`List`.

Some controllers are multi-threaded, where each thread calls a different reconcile function. Sieve uses the instrumented client libraries to obtain stacktraces whenever the controller reads or updates the cluster state (e.g., `Get`, `Create`). These stacktraces are used to differentiate between controller threads when generating and executing test plans.

5 Evaluation

Sieve's premise is that automatic and effective reliability testing for unmodified controllers is viable, by a) exhaustively

Controller	Systems	Dev.	#Stars	#Commits	#WL
cass-operator	Cassandra	DataStax	287	477	2
cassandra-operator	Cassandra	Instaclustr	227	337	2
casskop	Cassandra	Orange	177	1643	3
elastic-operator	Elasticsearch	Official	1832	3375	2
mongodb-operator	MongoDB	Percona	142	1407	5
nifikop	NiFi	Orange	101	232	3
rabbitmq-operator	RabbitMQ	Official	343	1679	3
xtradb-operator	XtraDB	Percona	302	1693	5
yugabyte-operator	YugabyteDB	Official	41	36	4
zookeeper-operator	ZooKeeper	Pravega	242	220	2

Table 2: **Kubernetes controllers used in our evaluation.** "#WL" stands for the number of different test workloads.

perturbing a controller's view of the cluster states and b) using differential oracles to flag safety and liveness issues.

We validate this hypothesis with three evaluation questions: 1) Can Sieve find new bugs in real-world controllers? 2) Does Sieve do so efficiently? 3) Are Sieve's testing results trustworthy? We answer these questions in the affirmative:

- §5.1: Sieve finds *new* bugs in *all* ten evaluated controllers, resulting in a total of 46 *new* bugs, which represent a swathe of safety and liveness issues. So far, 35 of them have been confirmed and 22 have been fixed by the developers.
- §5.2: All controllers can be tested in seven hours on a cluster of 11 machines, representing a typical nightly test. This is attributed to the effective reduction techniques which reduce test plans by 46.7%–99.6% across the controllers.
- §5.3: Sieve poses a low false positive rate of 3.5%.

Tested controllers We evaluated Sieve on ten popular controllers from the Kubernetes ecosystem for managing widely-used cloud systems (Table 2). The controllers are either developed by the official development team of the corresponding system, or by companies that have production-grade offerings around said systems. The term *operator* [12] in the project names refers to the Kubernetes design pattern of using a custom controller to manage an application.

Sieve employs 2–5 basic test workloads for each controller (Table 2). Each workload exercises a feature of the controller. Every evaluated controller supports software deployment and autoscaling, and therefore has at least two workloads. Sieve also employs workloads for controllers that support more features, such as sharding, storage resizing, reconfiguration, and load balancing. A test workload is typically implemented in 6–12 lines of code and takes 4–12 minutes to run.

It took us on average three hours to apply Sieve to each controller, which was mostly spent on understanding how to build the controller. We expect controller developers to expend much less effort to integrate Sieve in their workflow.

5.1 Finding New Bugs

Sieve finds a total of 46 *new* bugs in the evaluated controllers (Table 3). Those bugs include 11 intermediate-state bugs, 19 stale-state bugs, 7 unobserved-state bugs, and 9 bugs indirectly detected by Sieve during testing. Sieve finds new bugs in *all* the evaluated controllers. We have reported all these bugs. So far, 35 of them have been confirmed and 22 have been fixed. No bug report was rejected.

Controller	Interm. State	Stale State	Unobser. State	Indirect	Total
cass-operator	2	1	0	0	3
cassandra-operator	0	2	1	2	5
casskop	1	2	1	0	4
elastic-operator	0	2	0	0	2
mongodb-operator	2	3	1	3	9
nifikop	2	0	0	1	3
rabbitmq-operator	1	2	1	0	4
xtradb-operator	3	3	1	0	7
yugabyte-operator	0	2	1	2	5
zookeeper-operator	0	2	1	1	4
Total	11	19	7	9	46

Table 3: **New bugs detected by Sieve in each controller.**

Sieve can consistently reproduce all the 37 intermediate-, stale-, and unobserved-state bugs—running the test plan always reproduces the buggy behavior. In our experience, Sieve's reproducibility is invaluable for debugging test failures. It helps developers localize bugs in the source code and continuously iterate on bug fixes (§6).

Table 4 shows the consequences of the 46 controller bugs and an exemplar bug for each kind of consequence. We see that many bugs have severe consequences, such as application outages, security issues, service failures, and data loss. Note that these controllers are all mature projects (Table 2 and §6), suggesting that controller reliability is challenging to achieve.

The bugs that Sieve finds are deep and highly unlikely to be detected by manual testing or imprecise techniques like chaos testing or randomized fault injection tools [1, 2, 4, 5]. For example, a bug [63] in nifikop is triggered only when the controller crashes between issuing two specific state updates within one reconciliation loop (the time window between the two updates is about 0.7 milliseconds). In contrast, the test workload used for detecting the bug takes about 440 seconds to finish, and causes 481 reconciliation loops and 1,687 state updates issued by the controller. Sieve is able to detect and consistently reproduce the bug because it relies on injecting a fault precisely when a specific cluster-state change happens.

5.1.1 New Bugs Detected by Sieve

Intermediate-state bugs. Sieve found 11 intermediate-state bugs. Sieve stresses a common pattern among controllers, where they issue multiple updates per reconciliation, after the controller checks for a certain condition to hold in the cluster state. However, Sieve finds bugs when these condition checks only detect states from running the reconciliation loop in its entirety; that is, when the checks do not account for intermediate states that may arise due to controller crashes. For

Consequence	Example	# Bugs
Application outage	rabbitmq-operator-648: The RabbitMQ cluster is mistakenly turned down [69].	12
Service failure	K8SPSMDB-433: Sharding service for the MongoDB cluster wrongly terminated [44].	5
Data loss	K8SSAND-559: Storage volumes of Cassandra replicas wrongly deleted [49].	8
Reduced reliability	zookeeper-operator-314: The ZooKeeper cluster scaled down unexpectedly [90].	7
Misconfiguration	nifikop-49: The NiFi pod is not updated with new configuration [62].	6
Security issue	K8SPXC-896: TLS is not enabled for the XtraDB cluster [48].	6
Resource leak	cassandra-operator-398: Volumes used by deleted replicas never recycled [29].	7
Controller malfunc.	casskop-370: The controller stops serving scaling requests [30].	8

Table 4: **Consequences of the bugs found by Sieve (Table 3).** One bug can lead to multiple consequences.

example, in the intermediate-state bug in Figure 4, rabbitmq-operator compares `VolCur` and `VolDesired` to check whether the volume has been expanded already. However, this check assumes that all subsequent steps in the reconciliation succeed whenever this condition is satisfied. In the bug in Figure 1, the condition check cannot differentiate an intermediate state versus an unexpected faulty state. Part of the challenge is that controllers lack mechanisms analogous to write-ahead logging or journaling to guarantee atomicity of each reconcile action to enforce crash consistency. Controllers typically run as Kubernetes pods themselves and the newly created controller pod instance lacks any memory of its past execution (as they *should* – controllers must only depend on the *current state*). Sieve exposes those bugs without the need to understand source code—it systematically tests a controller with all possible intermediate states and checks for correctness.

Stale-state bugs. Sieve found 19 stale-state bugs. In our experience, it is notoriously challenging to anticipate all possible stale states. That said, we found controllers were not adequately using Kubernetes' mechanisms to tolerate asynchrony and staleness: like object versioning and unique IDs (instead of referring to objects by names, that need not be unique), or using coordination mechanisms to enforce ordering between events. Controllers also have the option to avoid staleness by using quorum reads to API servers, but this creates a scalability bottleneck as it drives more load to etcd – developers therefore choose to synchronize selectively. In general, we do not believe there is a shortcut to reasoning about any given update under all possible staleness or time-travel scenarios. Sieve therefore aids developers by systematically testing controllers under all possible time-traveling scenarios.

Unobserved-state bugs. Sieve found 7 unobserved-state bugs. We find that all of them are rooted in latent edge-triggering behavior in the controllers, that go against the Kubernetes philosophy of designing controllers to be level-triggered (§3.1). That is, these bugs arise when the controller's correctness relies on observing a specific state transition (edges), as exemplified by Figure 6. By identifying states that would be later overwritten and preventing those states from being observed by the controller, Sieve is effective at exposing unobserved-state bugs in controllers.

Bugs indirectly detected by Sieve. Sieve also finds 9 bugs that were not directly triggered by input states Sieve generated but *were still correctly flagged* by its differential oracles. All these bugs could (and do) happen in reference runs as well; but because Sieve executes many test plans, some test traces inevitably *differ* from the reference traces due to these bugs, allowing Sieve to detect them. These bugs are caused by a range of issues, including 1) controllers making incorrect assumptions about the Kubernetes API (e.g., assuming a list of pods from a query have stable ordinals); 2) spurious, dangling object creations, masked by Kubernetes' garbage collection (e.g., accidentally creating ZooKeeper pods after deleting the high-level ZooKeeper cluster object); 3) the applications managed by the controller being buggy and failing. Sieve can be extended with new perturbation patterns to systematically force out some of those bugs. For example, after understanding the root causes, we were able to reproduce two of these bugs consistently with manually written test plans.

5.1.2 Oracle Effectiveness

Sieve's differential oracles are crucial to detect buggy executions. Of the 46 newly found bugs, 45 were flagged by the differential test oracles. Checking logs for errors only flagged 5 bugs of which 4 were also found by our differential oracles.

Our end-state checker (§3.6.1) finds 28 bugs by comparing the end states of a test workload with and without perturbation. The state-update summaries checker (§3.6.2) finds 17 more bugs by checking the number of object updates through an execution. These oracles allow Sieve to detect bugs such as security and reliability issues (see §3.6) that do not manifest as simple failure symptoms (e.g. exceptions or process crashes).

The only bug that the differential oracles fail to find but is found by a regular error check in log files, is a null-pointer dereference bug [45] that causes an unexpected controller crash. Since Kubernetes automatically restarts the controller, it does not affect the end states or the state updates.

5.2 Test Efficiency

Table 5 shows the total time Sieve takes to test each controller in terms of machine hours. All experiments were run on 11 Amazon EC2 virtual machines, each with 8-core Intel(R) Xeon(R) Platinum 8259CL CPU with 2.50GHz and 32 GB memory, running Ubuntu 20.04.2 LTS.

Sieve's total testing time varies from 11.07 to 67.24 machine hours across the controllers. Sieve runs tests in parallel because every test plan is independent. With eleven virtual

Controller	Testing Time (Machine Hours)			# Test Plans
	Generation	Execution	Total	
cass-operator	0.60	43.67	44.27	218
cassandra-operator	0.49	10.72	11.21	81
casskop	0.57	12.40	12.97	125
elastic-operator	0.43	30.10	30.53	245
mongodb-operator	1.00	66.24	67.24	584
nifikop	1.17	41.61	42.78	239
rabbitmq-operator	0.47	10.60	11.07	133
xtradb-operator	1.40	62.96	64.36	395
yugabyte-operator	0.67	17.38	18.05	196
zookeeper-operator	0.33	13.75	14.08	164

Table 5: **Sieve's total testing time for each controller.**

machines, the total testing time for each controller is no more than 7 hours. Therefore, it is practical to run Sieve as a regular nightly test.

Over 95% of the testing time is spent on executing test plans. With the perturbations introduced by Sieve, a workload takes 8.8% longer to run on average. The overhead mainly comes from delays injected by Sieve for stale- and unobserved-state testing. In a few cases, when Sieve triggers bugs that lead to liveness issues, the controller hangs and triggers a timeout (by default, 10 minutes).

Sieve also spends 0.33–1.40 hours to 1) collect the reference trace and 2) generate test plans for each controller. The collected trace for each workload contains 3,386 events of notifications, updates, or reads on average. Generating test plans takes only 20 seconds for each workload on average.

Test reduction. Sieve's techniques to avoid ineffective test plans are key for tractability. Figure 9 breaks down the cumulative contribution of each technique. The baseline represents the basic rules described in §3.1 without any of the pruning techniques in §3.4. Overall, Sieve prunes away 46.7%–99.6% possible test plans across the evaluated controllers.

Specifically, pruning by causality (§3.4.1) reduces test plans by up to 95.0% across the controllers. This reduction is particularly effective for controllers that receive many notifications that are not causally related to any update. For example, mongodb-operator receives 700+ notifications regarding 20+ state objects, which are not causally related to most of its updates. This allows Sieve to prune *136,000*+ causally *unrelated* pairs of notifications and updates.

Pruning unsuccessful updates (§3.4.2) further prunes up to 75.8% of test plans across the controllers. In casskop, 60.0+% of updates issued by the controller do not affect the cluster state because the controller redundantly recreates two `service` objects that already exist. As none of these updates are relevant, Sieve excludes them when generating test plans.

Sieve finally prunes up to 72.9% of test plans across all controllers by focusing on deterministic triggering conditions (§3.7). This makes Sieve robust to many peculiar behaviors. For example, zookeeper-operator has an inefficient but benign behavior – it regularly clears the `NodePort` field of a `service`

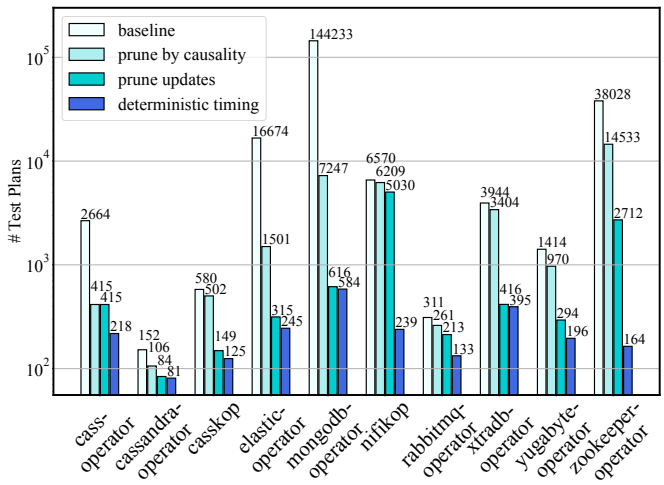

Figure 9: **Effectiveness of Sieve's test plan reduction techniques (§3.4).** The number of generated test plans is reduced by 46.7%–99.6% compared with the baseline.

object in every reconciliation, forcing Kubernetes to randomly allocate a port. This leads to thousands of state transitions with random port numbers. Sieve identifies these transitions to be nondeterministic and avoids related test plans.

5.3 False Positives

Sieve has a low false positive rate of 3.5%. It reports a total of 227 test failures for the ten evaluated controllers. 219 of them were true alarms—the test failures are caused by the 46 bugs described in §5.1 (one bug might fail multiple tests). The other eight test failures are false alarms.

The eight false alarms come from test results of three controllers (casskop, nifikop and xtradb-operator). All of them are caused by benign state transitions introduced in the faulty runs that did not happen in the reference runs.

The false alarms do not lead to opaque test failures—Sieve pinpoints the inconsistent fields. In all eight cases, the false alarms are easy to identify based on the identified fields and we could validate them by running the vanilla workload.

6 Discussion

In this section, we reflect on our experience building Sieve and studying the root causes of bugs it found (§5.1).

We find that all the evaluated controllers adopt mature software testing practices and have numerous unit, integration, and end-to-end test cases. Some even test scenarios involving faults. However, it is prohibitively difficult for developers to anticipate all possible cluster states that may occur, let alone codify them into test cases. Sieve fills this gap by exhaustively testing input states according to patterns of interest. For two bugs, Sieve detects that the initial bug fixes are deficient in covering all the conditions. *We run Sieve on the patched controllers and Sieve still detects the bugs!*

We also find that it is challenging for developers to comprehensively check test results, given the enormous state objects and their fields. Developers typically check a few fields of interest but such assertions can easily miss subtle, but serious issues (e.g., security vulnerabilities as discussed in §3.6).

We also observe that certain bugs are likely rooted in misunderstandings of Kubernetes' design and API semantics. For example, some unobserved-state bugs are caused by incorrectly assuming that every state change can be observed by the controller; some stale-state bugs can be prevented by using Kubernetes' mechanisms like resource versioning and precondition checking. We expect such problems to be more prevalent as engineers implement more and more custom controllers for their cluster management needs.

While cluster managers may avoid some classes of bugs, they come with hard tradeoffs. For example, not caching state objects at the controllers and API servers (Figure 2) could avoid stale-state bugs. However, it would introduce significant performance overheads to the controllers (memory accesses become network round trips) and make the data store a scalability bottleneck [26]. Also, transactions are not a solution for intermediate-state bugs – it would complicate the state-centric interface and prevent controllers from independently making progress regardless of failures, a key factor for resilience.

Since there is no silver bullet to implementing reliable controllers, we believe that automatic tools like Sieve are critical to cluster management reliability.

7 Limitations

Like other testing tools, Sieve is neither sound nor complete. Sieve uses specific perturbation patterns and exhaustively drives controllers to input states according to those patterns.

Sieve's differential oracles can yield both false negatives and positives. Sieve only applies its oracles on cluster states exposed by the state-centric interface. It is possible that certain application-specific states cannot be observed by the interface, which would lead to false negatives. In addition, Sieve reports false positives if the inconsistencies captured by the differential oracles are caused by benign state transitions that did not happen in the reference runs (§5.3). We found the false positive rate low (3.5%) in our evaluation.

The way Sieve deals with nondeterminism also leads to false negatives. Sieve excludes objects with nondeterministic metadata and masks nondeterministic field values in test plan generation and the differential test oracles (§3.7). This approach effectively avoids many irreproducible test plans and false positives, but also misses bugs that are triggered by states involving nondeterministic fields.

Lastly, Sieve depends on test workloads provided by the user for coverage. Implementing a test workload only takes 6-12 lines of code from our experience, but it requires domain knowledge about the controller and the system.

8 Related Work

Testing control-plane software. Modern SDNs have state-reconciliation based elements [15, 16, 19, 73] that could be tested using Sieve's methodology. A body of orthogonal work tests [28, 83] or verifies [37] how an SDN controller affects a network topology. For example, NICE [28] focuses on the boundary where controllers process packet-in/out events and uses that vantage point to automatically test for bugs. To the best of our knowledge, Sieve is the first work to focus on automatic reliability testing for cluster-manager controllers.

Testing distributed systems. Fault-injection tools [3, 25, 32, 34, 35, 55, 59, 79] have been developed for distributed systems, including chaos testing tools from the industry for cluster managers [1, 2, 4, 5]. Sieve's goals differ from those in the fault injection literature. Sieve seeks to expose controllers to as many input states as possible to test their reliability. For us, faults just happen to be a good mechanism to drive controllers to the required states. Compared to randomized chaos testing approaches that are unaware of cluster state transitions, Sieve can precisely force specific bug-triggering state transitions and consistently reproduce bugs. Furthermore, unlike prior art [32, 55, 87], Sieve is not based on an expert's hypotheses about vulnerable regions in the code under test.

A few prior tools can, in principle, expose some bugs found by Sieve. For example, concurrency-testing tools [61, 64, 65, 88] may expose bugs triggered by unobserved states (which in essence occur due to reordering of events). Similarly, tools that check for crash safety [22, 32, 55, 67] could expose bugs caused by the intermediate states. Finally, tools that inject network partitions [3], with expert guidance, could find some bugs caused by the stale-state pattern, i.e., a partition might force a controller to talk to a lagging API server (after talking to an up-to-date one). In contrast to these tools, Sieve does not target one class of bugs. Through exhaustive state perturbations, Sieve finds many kinds of bugs, essentially combining the power of prior targeted tools. Further, the chances that prior tools will find the bugs Sieve does are small, as they lack the context required to efficiently drive controllers to their buggy corners (e.g., a network-partition injector is unlikely to reliably orchestrate time-travel bugs).

Model checking. Sieve bears similarities to implementation-level model checking [40, 50, 54, 56, 61, 85, 86], in that we drive an unmodified implementation to a range of states to find bugs. Unlike model checking, Sieve does not seek to exhaustively cover the controller's state space. It instead executes developer-supplied test cases and exhaustively perturbs these test cases according to some fault patterns. Additionally, model checkers typically rely on a specification for correct behavior. While Sieve intentionally does not require hand-crafted specifications, it leans on reference traces as a partial specification of expected correct behavior.

Automation and ease-of-use. Sieve treats automation and ease-of-use as first-class design goals. Our automation is pri-

marily enabled by state-centric interfaces, which Sieve uses to produce perturbed states. Prior work has leveraged similar interface boundaries (e.g., the system-call interface [36,67]), enabling a general scheme to test multiple applications. For ease of use, Sieve does not require formal specifications [28,33,39], special test input [3, 24], code modifications [34, 61, 65], or whitebox analysis [54]. It also uses differential oracles to avoid the additional effort to supply domain-specific oracles [3, 22, 39]. However, Sieve's efficacy can be further improved with such expert guidance.

9 Conclusion

We present Sieve, the first automatic reliability testing technique for cluster management controllers. We find that Sieve is effective and practical. Sieve's usability and reproducibility play a critical role in understanding, debugging, and fixing reliability bugs. Sieve's testing technique is general and easy to extend – it separates the policy (how to perturb a controller's view of state) from mechanisms (how to realize perturbations). Hence, we are able to use the technique to detect a wide range of bugs without brittle heuristics, specifications or hypotheses. Our goal is to make Sieve a part-and-parcel of every controller developers' toolkit, and to harden the growing number of controllers that power today's data centers. We have made Sieve publicly available at https://github.com/sieve-project/sieve.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Ranjita Bhagwan, for their insightful comments. We thank Marcos Aguilera, Sujata Banerjee, Mihai Budiu, Jon Howell, Rob Johnson, Matthew Lentz, Darko Marinov, Davanum Srinivas, Chaitanya Bhandari, Yinfang Chen, Lilia Tang, and Shuai Wang for valuable feedback and discussions that helped shape this work. We thank all the controller developers who engaged with us and reviewed our reports and patches. This work was funded in part by NSF SHF-1816615, CNS-2130560, CNS-2145295, and a VMware Research Gift.

References

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

[2] Chaoskube: chaoskube periodically kills random pods in your kubernetes cluster. https://github.com/linki/chaoskube, 2020.

[3] Jepsen. https://jepsen.io/, 2020.

[4] Kubemonkey: An implementation of netflix's chaos monkey for kubernetes clusters. https://github.com/asobti/kube-monkey, 2020.

[5] Pumba: Chaos testing, network emulation, and stress testing tool for containers. https://github.com/alexei-led/pumba, 2020.

[6] API Server Tracing. https://github.com/kubernetes/kubernetes/pull/94942, 2021.

[7] Controllers and Reconciliation. https://cluster-api.sigs.k8s.io/developer/providers/implementers-guide/controllers_and_reconciliation.html, 2021.

[8] Decorated Syntax Tree. https://github.com/dave/dst, 2021.

[9] How finalizers work. https://kubernetes.io/docs/concepts/overview/working-with-objects/finalizers/#how-finalizers-work, 2021.

[10] kind: Kubernetes IN Docker - local clusters for testing Kubernetes. https://kind.sigs.k8s.io/, 2021.

[11] Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/, 2021.

[12] Kubernetes Operators. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/, 2021.

[13] kubernetes-sigs/controller-runtime. https://github.com/kubernetes-sigs/controller-runtime, 2021.

[14] kubernetes/client-go. https://github.com/kubernetes/client-go, 2021.

[15] Open Virtual Networking. https://github.com/ovn-org/ovn, 2021.

[16] Open vSwitch: Production Quality, Multilayer Open Virtual Switch. https://www.openvswitch.org/, 2021.

[17] OpenShift: The developer and operations friendly Kubernetes distro. https://github.com/openshift, 2021.

[18] Resource versions. https://kubernetes.io/docs/reference/using-api/api-concepts/#resource-versions, 2021.

[19] VMware NSX-T. https://docs.vmware.com/en/VMware-NSX-T-Data-Center/index.html, 2021.

[20] vSphere: Unified Management for Containers and VMs. https://www.vmware.com/products/vsphere.html, 2021.

[21] What is a Level Based API. https://book-v1.book.kubebuilder.io/basics/what_is_a_controller.html, 2021.

[22] Alagappan, R., Ganesan, A., Patel, Y., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).

[23] Alpar, A. Using Finalizers to Control Deletion. https://kubernetes.io/blog/2021/05/14/using-finalizers-to-control-deletion/, May 2021.

[24] Alquraan, A., Takruri, H., Alfatafta, M., and Al-Kiswany, S. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)* (Oct. 2018).

[25] BASIRI, A., BEHNAM, N., DE ROOIJ, R., HOCHSTEIN, L., KOSEWSKI, L., REYNOLDS, J., AND ROSENTHAL, C. Chaos Engineering. *IEEE Software 33*, 3 (Mar. 2016), 35–41.

[26] BROOKER, M. The Fundamental Mechanism of Scaling. http://brooker.co.za/blog/2021/01/22/cloud-scale.html, 2020.

[27] 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.

[28] 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).

[29] CASSANDRA-OPERATOR-398. [BUG] Reconcilation fails to delete PVCs if missing a deletion timestamp of the Cassandra pod. https://github.com/instaclustr/cassandra-operator/issues/398, 2021.

[30] CASSKOP-370. [BUG] Casskop fails to clean up PVCs and refuses to handle user requests after crash and restart. https://github.com/Orange-OpenSource/casskop/issues/370, 2021.

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

[32] CHEN, H., DOU, W., WANG, D., AND QIN, F. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering (ASE'20)* (Sept. 2020).

[33] DELIGIANNIS, P., MCCUTCHEN, M., THOMSON, P., CHEN, S., DONALDSON, A. F., ERICKSON, J., HUANG, C., LAL, A., MUDDULURU, R., QADEER, S., AND SCHULTE, W. Uncovering Bugs in Distributed Storage Systems during Testing (not in Production!). In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)* (Feb. 2016).

[34] DRĂGOI, C., ENEA, C., OZKAN, B. K., MAJUMDAR, R., AND NIKSIC, F. Testing Consensus Implementations Using Communication Closure. In *Proceedings of 2020 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'20)* (Nov. 2020).

[35] GANESAN, A., ALAGAPPAN, R., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)* (Feb. 2017).

[36] GONG, S., ALTINBÜKEN, D., FONSECA, P., AND MANIATIS, P. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-Thread Communication Analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)* (Oct. 2021).

[37] GUHA, A., REITBLATT, M., AND FOSTER, N. Machine-Verified Network Controllers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)* (June 2013).

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

[39] GUNAWI, H. S., DO, T., JOSHI, P., ALVARO, P., HELLERSTEIN, J. M., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., SEN, K., AND BORTHAKUR, D. Fate and Destini: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)* (Mar. 2011).

[40] GUO, H., WU, M., ZHOU, L., HU, G., YANG, J., AND ZHANG, L. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (Oct. 2011).

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

[42] 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.

[43] K8SPSMDB-430. [BUG] Stale deletion timestamps lead to undesired statefulset and PVC deletion. https://jira.percona.com/browse/K8SPSMDB-430, 2021.

[44] K8SPSMDB-433. [BUG] Sharding stateful set gets mistakenly deleted when reading stale field values. https://jira.percona.com/browse/K8SPSMDB-433, 2021.

[45] K8SPSMDB-434. [BUG] Nil pointer dereference when reconfiguring spec.sharding.enabled. https://jira.percona.com/browse/K8SPSMDB-434, 2021.

[46] K8SPSMDB-578. [BUG] Failure of creating SSL-internal certificates when the controller crashes and restarts at some particular point. https://jira.percona.com/browse/K8SPSMDB-578, 2021.

[47] K8SPXC-725. [BUG] HAproxy stateful set and services get mistakenly deleted when reading stale spec.haproxy.enabled. https://jira.percona.com/browse/K8SPXC-725, 2021.

[48] K8SPXC-896. [BUG] The controller fails to set up SSL-internal certificates if a crash happens at some particular point. https://jira.percona.com/browse/K8SPXC-896, 2021.

[49] K8SSAND-559. [BUG] PVCs can be deleted mistakenly when reading stale deletion timestamp information. https://k8ssandra.atlassian.net/browse/K8SSAND-559, 2021.

[50] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI'07)* (Apr. 2007).

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

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

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

[54] LEESATAPORNWONGSA, T., HAO, M., JOSHI, P., LUKMAN, J. F., AND GUNAWI, H. S. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[55] LU, J., LIU, C., LI, L., FENG, X., TAN, F., YANG, J., AND YOU, L. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *Proceedings of the 26th ACM Symposium on Operating System Principles (SOSP'19)* (Oct. 2019).

[56] LUKMAN, J. F., KE, H., STUARDO, C. A., SUMINTO, R. O., KURNIAWAN, D. H., SIMON, D., PRIAMBADA, S., TIAN, C., YE, F., LEESATAPORNWONGSA, T., GUPTA, A., LU, S., AND GUNAWI, H. S. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the 14th EuroSys Conference 2019 (EuroSys'19)* (Mar. 2019).

[57] MACCÁRTHAIGH, C. PID loops and the art of keeping systems stable. https://www.infoq.com/presentations/pid-loops/, June 2019.

[58] MACE, J., ROELKE, R., AND FONSECA, R. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[59] MAJUMDAR, R., AND NIKSIC, F. Why is Random Testing Effective for Partition Tolerance Bugs? In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)* (Jan. 2018).

[60] MUSAJI, M. Why Operators are essential for Kubernetes. https://www.redhat.com/en/blog/why-operators-are-essential-kubernetes, Apr. 2021.

[61] MUSUVATHI, M., QADEER, S., AND BALL, T. CHESS: A Systematic Testing Tool for Concurrent Software. Tech. Rep. MSR-TR-2007-149, November 2007.

[62] NIFIKOP-49. [BUG] NiFi configuration cannot be reloaded if the controller crashes and restarts in the middle of a reconciliation. https://github.com/konpyutaika/nifikop/issues/49, 2021.

[63] NIFIKOP-79. [BUG] Nifikop fails to scale down NiFi cluster due to a crash in the middle of reconcileNifiPod. https://github.com/konpyutaika/nifikop/issues/79, 2022.

[64] OZKAN, B. K., MAJUMDAR, R., NIKSIC, F., BEFROUEI, M. T., AND WEISSENBACHER, G. Randomized Testing of Distributed Systems with Probabilistic Guarantees. In *Proceedings of 2018 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'18)* (Nov. 2018).

[65] OZKAN, B. K., MAJUMDAR, R., AND ORAEE, S. Trace Aware Random Testing for Distributed Systems. In *Proceedings of 2019 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'19)* (Oct. 2019).

[66] PERCONA DISTRIBUTION FOR MONGODB OPERATOR DOCUMENTATION. Transport Layer Security (TLS). https://www.percona.com/doc/kubernetes-operator-for-psmongodb/TLS.html, 2021.

[67] PILLAI, T. S., CHIDAMBARAM, V., ALAGAPPAN, R., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[68] PIPES, J., HAUSENBLAS, M., AND TABER, N. Introducing the AWS Controllers for Kubernetes (ACK). https://aws.amazon.com/cn/blogs/containers/aws-controllers-for-kubernetes-ack/, Aug. 2020.

[69] RABBITMQ-OPERATOR-648. [BUG] Reading stale RabbitMQ cluster information leads to unexpected StatefulSet deletion. https://github.com/rabbitmq/cluster-operator/issues/648, 2021.

[70] RABBITMQ-OPERATOR-782. [BUG] PVC expansion fails if the controller crashes in the middle of a reconciliation. https://github.com/rabbitmq/cluster-operator/issues/782, 2021.

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

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

[73] SCOTT, C., WUNDSAM, A., RAGHAVAN, B., PANDA, A., OR, A., LAI, J., HUANG, E., LIU, Z., EL-HASSANY, A., WHITLOCK, S., ACHARYA, H., ZARIFIS, K., AND SHENKER, S. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM'14)* (Aug. 2014).

[74] 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.

[75] 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).

[76] SUN, X., SURESH, L., GANESAN, A., ALAGAPPAN, R., GASCH, M., TANG, L., AND XU, T. Reasoning about Modern Datacenter Infrastructures Using Partial Histories. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'21)* (June 2021).

[77] 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).

[78] TANG, Z., LI, X., AND GUO, F. Demystifying Kubernetes as a service – How Alibaba cloud manages 10,000s of Kubernetes clusters. https://www.cncf.io/blog/2019/12/12/demystifying-kubernetes-as-a-service-how-does-alibaba-cloud-manage-10000s-of-kubernetes-clusters/, Dec. 2019.

[79] TSEITLIN, A. The Antifragile Organization. *Communications of the ACM 56*, 8 (Aug. 2013), 40–44.

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

[81] VMWARE. Introducing vSphere Lifecycle Management (vLCM). https://core.vmware.com/resource/introducing-vsphere-lifecycle-management-vlcm#section1.

[82] VMWARE. What is intent-based networking (IBN)? https://www.vmware.com/topics/glossary/content/intent-based-networking.html.

[83] XU, L., HUANG, J., HONG, S., ZHANG, J., AND GU, G. Attacking the Brain: Races in the SDN Control Plane. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)* (Aug. 2017).

[84] 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).

[85] YABANDEH, M., KNEZEVIC, N., KOSTIC, D., AND KUNCAK, V. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)* (Apr. 2009).

[86] YANG, J., CHEN, T., WU, M., XU, Z., LIU, X., LIN, H., YANG, M., LONG, F., ZHANG, L., AND ZHOU, L. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)* (Apr. 2009).

[87] 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).

[88] YUAN, X., AND YANG, J. Effective Concurrency Testing for Distributed Systems. In *Proceedings of the 25th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'20)* (Mar. 2020).

[89] 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 ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)* (Oct. 2021).

[90] ZOOKEEPER-OPERATOR-314. [BUG] Reading stale ZooKeeper cluster status can lead to undesired pod and PVC deletion. https://github.com/pravega/zookeeper-operator/issues/314, 2021.