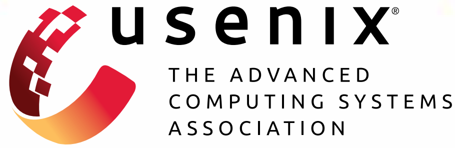


Kivi: Verification for Cluster Management

Bingzhe Liu and Gangmuk Lim, *UIUC;* Ryan Beckett, *Microsoft;*
P. Brighten Godfrey, *UIUC and Broadcom*

https://www.usenix.org/conference/atc24/presentation/liu-bingzhe

This paper is included in the Proceedings of the
2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the
2024 USENIX Annual Technical Conference
is sponsored by

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

Kivi: Verification for Cluster Management

Bingzhe Liu
UIUC

Gangmuk Lim
UIUC

Ryan Beckett
Microsoft

P. Brighten Godfrey
UIUC and Broadcom

Abstract

Modern cloud infrastructure is powered by cluster management systems such as Kubernetes and Docker Swarm. While these systems seek to minimize users' operational burden, the complex, dynamic, and non-deterministic nature of these systems makes them hard to reason about, potentially leading to failures ranging from performance degradation to outages.

We present Kivi, the first system for verifying controllers and their configurations in cluster management systems. Kivi focuses on the popular system Kubernetes, and models its controllers and events into processes whereby their interleavings are exhaustively checked via model checking. Central to handling autoscaling and large-scale deployments are our modeling optimizations and our design which seeks to find violations in a smaller and reduced topology. We show that Kivi is effective and accurate in finding issues in realistic and complex scenarios and showcase two new issues in Kubernetes controller source code.

1 Introduction

Modern cloud infrastructure relies on technologies like containers, microservices, and serverless computing to develop applications that are resilient and manageable. To orchestrate these components, practitioners are widely adopting cluster management systems like Kubernetes (K8s) and Docker Swarm. These systems consist of a diverse collection of controllers (e.g., scheduler, autoscaler). Aiming to minimize the users' operational burden, these controllers run in a continuous closed loop and automatically drive the cluster towards desired goals through manipulating shared cluster objects.

However, it is challenging for users to safely and correctly use these cluster management systems, and many outages have occurred as a result [10, 15, 20, 22, 35]. Even a single controller can consist of sophisticated workflow logic that is influenced by diverse configuration parameters, introducing opportunities for human mistakes. For example, the scheduler in K8s has 12 different pipeline stages and 21 complex scheduling strategy plugins (each of which contains a few or even a dozen of parameters to tune) to choose from for these stages [25]. Moreover, even if users correctly configure a single controller, multiple controllers can have non-trivial interactions because they operate on overlapping sets of cluster components [56]. Furthermore, frequent operational changes, multiple teams of engineers making changes with different goals, and unpredictable environmental events (e.g., workload changes and failures) all amplify the chance of a mistake.

Through case studies of community-collected failure cases [2], we have identified a collection of unintended or pathological behaviors (§3.1) that can happen due to the aforementioned challenges. Unintended behaviors include the number of pods (a pod is the smallest deployable unit in K8s) dropping below a necessary minimum, or the placement of pods becoming unexpectedly unbalanced. Pathological behaviors include a pod oscillating in an unending cycle of scheduling and eviction, or failing to be scheduled despite the existence of plentiful resources. These behaviors can result in performance degradation and even serious service outages, yet are hard for users to reason about manually.

With the increasing complexity of cluster management systems, we believe users need automated ways to check the correctness of their clusters. Testing and emulation [44, 47, 57–59, 63, 74, 75, 78] are common ways to find issues, but they are insufficient for cluster management systems where controllers run asynchronously and events may take place in any non-deterministic order. On the other hand, verification techniques are known to provide high coverage of distributed systems by considering all interleavings of components. There have been many successful stories using verification for distributed systems and networking (e.g., [40–43, 49, 50, 53, 61, 62, 67, 72, 73, 76, 77, 79]), yet none have been tailored to cluster management systems. They either focus on low-level implementation details or specific protocols (e.g., Paxos, BGP), instead of the properties that are related to the entanglement of several or even tens of control components. For instance, in K8s, the number of replicas is affected by the deployment controller, horizontal pod autoscaling (HPA), scheduler (and its various plugins), descheduler, events like node failures, and more.

In this paper, we present Kivi, the first system for verifying cluster management system controllers and their configurations. Kivi focuses on Kubernetes, the most popular open-source cluster management platform. Kivi takes the users' intent[1], the state and configuration of the cluster, and

[1] We use "intent" to mean a set of properties that the cluster administrator wants to ensure.

the event assumptions as inputs, and verifies if the cluster can violate the intent. If a violation is possible, it generates a minimal counterexample. Kivi checks on four categories of properties (§3.2) that we have identified in the domain of cluster management systems: unexpected topology, object numbers, object lifecycles, and oscillation.

It is challenging to effectively verify K8s, as it has a very complex and large implementation. To make verification tractable, we need to move from source code to a model; but this in turn involves work to build a faithful model. Our main goal in Kivi is to verify the *interactions* between controllers or between controllers and events. Therefore, to make modeling feasible and sufficiently faithful, we found it is enough to model the high-level logic the controllers use to interact with the cluster (i.e., via manipulating object states like the status of pods). Kivi leverages the explicit model checker SPIN [51]. We model K8s objects into shared global states and model controllers and events into processes whereby their non-deterministic interleaving can be searched exhaustively by SPIN.

Scalability is another daunting challenge in cluster management systems. A cluster can reach hundreds of nodes and many thousands of pods leading to the state explosion problem [45]. Furthermore, autoscalers are common in cluster management systems, so users would be interested in not only one but a wide range of cluster topologies. There may be multiple dimensions for autoscaling (i.e., various types of nodes and pods), and the possible topologies that need to be verified can grow exponentially against the cluster size.

To tackle the scalability challenges, we posit a hypothesis: *if a cluster setup can violate an intent, then it can do so at relatively small scale*. Prior work [60, 74] has made a similar observation in other types of systems. According to this intuition, we designed an incremental scaling algorithm that starts to verify a cluster at the smallest non-trivial scale, and then intelligently increases the scale across multiple scaling dimensions until either finding a violation or reaching an *empirically sufficient scale (ESC)* and concluding that – if the ESC is sufficiently large to make the hypothesis true – then no violation is possible at any scale. This dramatically improves performance compared to verifying all possible scales, and produces violations that are more minimal and easier for users to understand. However, it does not guarantee zero false negatives; confidence in the no-violation response is empirical.

In addition, to improve the performance of individual runs of the model, we have implemented a few optimization mechanisms to reduce the verification search space, which enables our model to scale to sufficiently large problem sizes.

Our implementation of Kivi includes six commonly-used and representative controllers with logic derived from the K8s source code. We evaluate Kivi on a test suite of eight representative cases derived from realistic failures. Our key findings are:

- *Validating the small-scale hypothesis.* Although the hypothesis is not radical, it is still necessary to (a) validate it in the novel domain of cluster management systems, and (b) quantify what scale is enough. Using our collected test cases, we show experimentally that intent violations do consistently appear at small scale: the maximum minimum size needed to produce a violation is only 3 nodes and 6 pods. In 6 of 7 cases, Kivi found violations at even smaller scale than the original problem report.
- *Performance.* Our evaluation using realistic failure cases shows that Kivi verifies most cases within 100 seconds and all cases within 25 minutes. Without our incremental scaling algorithm, verification times out ($>$ 10000 sec) even at moderate scale ($\leq$ 50 nodes).
- *Accuracy.* Kivi has successfully found the correct violations for all configuration violation cases and reported no failures for non-violation cases. We have also performed a comparison with real K8s cluster runs and found that Kivi closely models the real system.
- *New issues found.* Though we mostly focus on misconfiguration issues, Kivi manages to find two new issues in the implementation of a K8s controller.

This work does not raise any ethical issues. We summarize our contributions as follows:

1. Though K8s is widely used, there is limited work on controller interaction problems. We shine a light by presenting categories of failures that are caused by the non-trivial interactions in K8s, and identifying new properties for verification in this domain.
2. We present Kivi, the first system for verifying controllers and their configurations in cluster management systems. Kivi mostly focuses on finding misconfiguration issues, but can also find controller logic issues. We implement an accurate model of selected K8s objects and controllers. Our model is designed with a set of optimizations that can be adopted by future verification systems in the domain. Our evaluation shows that Kivi is scalable and can verify realistic failure cases.
3. We posit the hypothesis that for properties of interest in K8s it is sufficient to verify at small scale. Although this hypothesis is known to be true in some other domains, we empirically validate and quantify it in the novel domain of cluster management systems.

2 Modern Cluster Management Systems

Cluster management systems [1, 3, 30, 66] orchestrate the lifecycles of compute resources by allocating and scaling resources efficiently to improve application performance and reliability. They consist of multiple controllers that operate asynchronously to drive the cluster towards desired goals through manipulating shared *objects*. While the concepts are more general, we describe here the basic objects in K8s. ***Pods***[2]

[2]Pods are also called replicas in the K8s deployment configuration. We use replicas and pods interchangeably in this paper.

are the smallest deployable unit that each consists of one or several containers with shared resources. ***Nodes*** are virtual or physical machines that run pods. ***Workloads*** define the lifecycle of a group of pods, including both user configuration on how they should be scaled, updated, and terminated, and also the live state such as replica numbers. ***Deployments*** is the most common workload that manages stateless applications. There are other types of workloads like StatefulSets [34], DaemonSet [26] and Job [28].

Each controller attempts to drive one or more objects to their goal states by periodically checking the states via the logically centralized communication channel *API server* and reconciling. Multiple controllers may operate on the same objects. Though controllers share states via the API server, there is no agreed-upon notion of a globally desired goal state. We briefly summarize the goal of the most popular Kubernetes controllers in Table1.

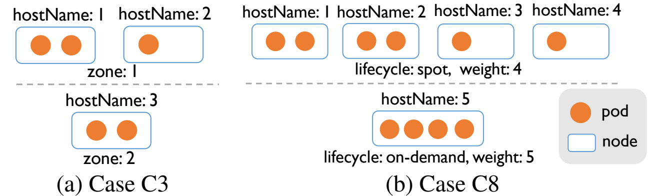
Controllers	Description	Target Objects
Scheduler	Places pods to nodes. It filters out infeasible nodes and selects the best node based on scheduling preference.	Pods
Descheduler	Actively evicts pods when placement could drift away from the desired state.	Pods
Horizontal Pod Autoscaler (HPA)	Autoscales pod number based on given target resource utilization (e.g., CPU usage) within a specified range.	Workloads
Kubelet	Responsible for managing the lifecycle of pods on each node.	Pods
Workload Controllers (WLC)	A set of controllers that each manages a specific type of workload, e.g., deployment controller.	Pods, Workloads
Node Controller	Manages the lifecycle of nodes.	Nodes
Cluster Autoscaler (CA)	Adjusts the number of nodes according to cluster resource usage.	Nodes
Ingress Controller	Load balances outside requests to pods.	Requests

Table 1: Most popular Kubernetes controllers.

3 Failure Case Study and Takeaways

We studied community-collected failure cases [2] supplemented with failures mentioned in talks at KubeCon (the main conference for the Kubernetes developers and users) between 2017-2023, and potential problems mentioned in the Kubernetes official documentation. While there are various reasons for failures, including DNS issues [11, 18], Linux kernel issues [17, 19, 23], configuration syntax problems [13, 22], and credential issues [8, 13], we are interested in the failures that are caused by **the non-trivial interactions between controllers, or between controllers and events**. We have collected 16 failure cases related to such non-trivial interactions and summarized them in Table 2 and an extended Table 6 in Appendix A.

In §3.1, we summarize the causes of failure into three categories and describe a few motivating examples. Based on these failures we identify several important properties to verify for cluster management systems in §3.2. We finally discuss the takeaways for verification tool design in §3.3.



(a) Case C3 (b) Case C8

Figure 1: Cluster configurations for two failure examples.

3.1 Why do problems occur?

Non-trivial interactions between components in a single controller. Even configuring a single controller correctly is challenging. For example, the scheduler contains 12 pipeline stages that can be extended with 21 default plugins [25] for a wide range of strategies. Each plugin contains various configs, some of which can interact with other plugins. For example, users can configure `PodTopologySpread(SPTS)` plugin (which aims to evenly spread pods) with an unlimited number of constraints where each constraint has 8 parameters, 2 of which are related to another two plugins[3]. Furthermore, many important details have not been documented well and users can easily make mistakes[4].

We describe a simple example that shows the non-trivial interaction between configurations in a single plugin. Before we dive into details, we define each `SPTS` constraint as `skew(`k`)` $\leq a$, which means that the difference in the number of pods in any two *groups* is $\leq a$, where each group is defined as the set of nodes that have the same value of a given label k. For example, `skew(zone)` $\leq$ `1` is satisfied by Fig. 1a: there are two groups, one defined by the label `zone:1` that has 3 pods, and the other defined by `zone:2` that has 2 pods, and hence the difference between the two groups is 1.

Case C3 (Conflict SPTS constraints) [27]. A 3-node cluster was labeled as shown in Fig. 1a. A 6-pod deployment was configured with two `SPTS` constraints: `skew(hostName)` $\leq$ `1` and `skew(zone)` $\leq$ `1`. The first 5 pods were scheduled successfully as shown in the figure. However, the 6th pod failed to be scheduled because it could not satisfy both constraints. This issue only manifests under specific topologies. If each zone is configured with a similar number of nodes, the user would not encounter this issue.

Non-trivial interactions between controllers. Kubernetes contains various controllers, each of which has its own goals while manipulating shared objects. Furthermore, these controllers may be configured by multiple teams with different goals. The interactions between controllers may affect each

[3] The configurations define if `NodeAffinity` and `NodeTaints` would be considered when calculating the skewness in topology.

[4] For example, if multiple `SPTS` constraints are defined for a pod, the node's labels need to match with all constraints to be considered in the skewness calculation and as a candidate for the pod.

Case ID	Description	Reasons	Min Violation Scale $\langle |N|, |P| \rangle$	Reported Scale $\langle |N|, |P| \rangle$
C1 [21]	Pods consumed high CPU during bootstrapping leading HPA to scale up rapidly to max replicas.	CE	$\langle 1, 3 \rangle$	$\langle \text{Unknown}, 150+ \rangle$
C2 [4,6]	Not enough replicas because users applied an updated YAML file without defining number of replicas (1 by default).	MC, CE	$\langle 1, 2 \rangle$	$\langle \text{Unknown}, 3 \rangle$
C3 [27]	Configurations of two `SPTS` constraints caused the 6th pod to fail to be scheduled.	SCC	$\langle 3, 6 \rangle$	$\langle 3, 6 \rangle$
C4 [15]	Scheduler kept assigning pods with high CPU usage to the same node causing a kernel panic and pod failure loop.	CE	$\langle 2, 5 \rangle$	Unknown
C5 [24]	Conflict configurations of scheduler and `RemoveDuplicate` policy in descheduler.	MC	$\langle 2, 4 \rangle$	$\langle 5, 10 \rangle$
C6 [5]	Pods unbalanced after maintenance. Node failures then caused the pod count to drop too low.	CE	$\langle 2, 2 \rangle$	$\langle 3, 3 \rangle$
C7 [16]	Conflicting configurations of node `taint` and pod `NodeName` caused scheduling and eviction loop	MC	$\langle 1, 1 \rangle$	$\langle \text{Unknown}, 5 \rangle$
C8 [24]	Conflicting descheduler and scheduler configurations caused scheduling and eviction loop.	MC	$\langle 3, 6 \rangle$	$\langle 5, 10+ \rangle$

Table 2: Failure cases that are caused by the non-trivial interactions between controllers and events. MC, SCC and CE stand for interactions between *multiple controllers*, *single controller components*, and *controllers and events*. The last two columns show the minimum scale (where $|N|$ and $|P|$ are the total number of nodes and pods accordingly) that we find violations, and whether it is smaller than the reported scale.

Property Categories	Description	Related Controller	Type	Failure Cases
Unexpected Topology	Objects should be placed in certain patterns according to users' optimization metrics. E.g., requests or pods may need to be evenly spread to improve failure tolerance, or certain nodes may contain special resources (e.g., GPUs) that should be saved for specific pods.	Scheduler, CA, Ingress Controller	Safety	C6 (C9, C15)
Unexpected Object Numbers	The number of objects should fall into a certain range. Too many pods or nodes can consume excessive resources and too few can affect the reliability of the applications.	Scheduler, Descheduler, HPA, WLC, CA	Safety	C1, C2, C6 (C10)
Unexpected Object Lifecycles	The lifecycle of an object includes its creation, execution, and termination after task completion or failure. An object may go through an unexpected lifecycle, e.g., due to failure during creation or unexpected eviction.	Scheduler, Descheduler, Kubelet, WLC, Node Controller	Safety	C3, C4, C7, C8 (C11, C12, C13, C15)
Oscillation	The state of the cluster becomes unstable and changes back and forth in an unending cycle.	Any	Liveness	C4, C5, C7, C8

Table 3: Taxonomy of properties. Failure cases in parentheses are in the Appendix A.

other's optimization goals or even become conflicts, causing performance degradation or even pathological behavior.

Case C8 (Conflict between scheduler and descheduler) [24]. A 5-node cluster was labeled with `hostName` and `lifecycle` as shown in Fig. 1b. A 10-pod deployment was configured to use two scheduling plugins: (1) `SPTS` with two soft constraints `skew(hostName)`≤1 and `skew(lifecycle)`≤1; (2) `NodeAffinity(SNA)` where the nodes with `on-demand` for the label `lifecycle` are preferred during scheduling than the nodes with the value `spot`, with weight of 5 and 4 respectively. The two `SPTS` constraints in fact conflict with each other (similar to Case C3) and with the `SNA`: `SPTS` is targeted to spread evenly while `SNA` prefers the `on-demand` nodes. Therefore, the `SPTS` constraints cannot be satisfied, yet scheduling was still successful because `SPTS` were soft constraints. However, a descheduler was configured to hold the `SPTS` constraints even though they were soft. The non-trivial interactions between three controllers led to an unending eviction and scheduling cycle: the descheduler evicted pods, the deployment controller added the pods back to maintain the desired replicas, and scheduler scheduled pods back.

Non-trivial interactions between controllers and events. Various events can occur that affect the status of the objects in a live cluster. For example, environmental events like pod CPU changes (e.g., due to increased user requests) and node failures, and operational events like maintenance and new application deployment. Users may not consider these events when they configure controllers, and moreover, these events can happen non-deterministically, resulting in failures that are hard to predict prior to deployment.

Case C6 (Pod unbalance after maintenance) [5]. Pods initially were evenly spread across the nodes. A maintenance event happened that took down a node, resulting in rescheduling of the affected pods. After the maintenance was completed and the node came back, however, the pods was not rescheduled back to the node, resulting in an unbalanced topology and leaving the cluster with potential vulnerability to failures. A descheduler should have been used to re-balance the pods.

Case C2 (Exceeded number of pods caused by CPU usage) [6]. The pods in a deployment consumed high CPU (100% of requested) at the bootstrapping phase and then dropped back to normal usage. However, the high CPU usage at the beginning led the HPA to rapidly scale up the pods until hitting the maximum number of replicas allowed. After pods gradually went through the bootstrapping phase, the HPA then slowly scaled down these extra pods. These extra pods wasted many resources without serving actual traffic and caused great confusion for the users.

3.2 Properties to Verify

We observe that each controller interacts with the cluster by manipulating *a small set of well-defined objects and their*

states. For example, the scheduler listens to the creation of pods and controls the cluster by either changing the *topology (i.e., the placement) of the pods*, or affecting the *number of pods* and *pod lifecycles* when it fails to find suitable nodes. To get good coverage of the properties for how controllers interact with the cluster, we enumerate the most popular controllers in K8s, and we identify three object states that cover what these controllers manipulate: **object topology**, **object numbers**, and **object lifecycles**. We cross-validate this finding with the failure cases. We find that 15 of 16 cases are caused by not meeting the expected goal of one or more of the three object states, except for one case caused by the unexpected lifecycle of the controller itself. We summarize four categories of properties[5] to check in Table 3 according to the three object states. **Oscillation** is a special type, as it checks if any object states can be changed back and forth in an unending cycle. Note that each property category can contain several properties to verify, as each category can be "parameterized" with object types (i.e., pods, nodes, requests, workloads). Among these properties, oscillation is a liveness property while the rest are safety properties. Oscillation checks for *pathological* behaviors, and others check on *unintended* behavior that does not match with users' intent.

3.3 Takeaways

The goal of our case study is to learn the characteristics of the failures that are caused by the non-trivial interactions between components in the cluster management systems. Although the study is not broad enough to serve as a complete quantitative study for the domain, we believe the cases we collected have good diversity and are representative in terms of the involved Kubernetes components, the root causes, and the underlying properties being violated. These findings can guide us on what to model in Kivi: we should include events, controllers and their features (e.g., we should model different plugins in the scheduler), objects and their attributes (e.g., we should model the CPU usage of pods), the properties mentioned in §3.2, and model quantities (both real and integer numbers).

We believe the lessons we learned can also help with other future verification work in this domain.

4 Kivi System Design

To use Kivi, users provide their cluster configurations and select the properties that they want to verify from our property library. Kivi takes users' input, and either assures that the desired properties will be preserved or generates minimal counterexamples. §4.1 describes the workflow of Kivi. A critical component of Kivi is the *model*. Kivi implements a carefully-designed model of Kubernetes controllers, objects, and events suitable for exhaustive analysis by a model checker. In order to give this component a deep enough discussion, we describe it separately in §5.

[5]These properties are not fully orthogonal. E.g., C8 counts for both oscillation and unexpected object lifecycles. Users can verify either property.

Verifying the cluster management systems raises significant performance challenges due to many execution paths, parameter configurations, and potentially large scale; we describe how Kivi tackles the problems of performance and scale with a scaling algorithm in §4.2 and model design and optimization in §5. We describe the workflow of the main component that operates the verification procedures in §4.3.

4.1 System Workflow

Figure 2 shows the workflow of Kivi, including the five main components of its design:

The *Parser* takes three main elements as inputs: (1) the object configurations that contain the configurations of nodes and Kubernetes workloads, e.g., deployment YAML file, (2) the controller and event configurations that configure the behavior of the controllers and event assumption, e.g., the HPA YAML file, and (3) intent that describe operators' expected behavior of the cluster. *Parser* takes these inputs and parses them into a uniform format, the *Cluster Setup*, and sends it to the *Verifier Operator*. A *Cluster Setup* (illustrated in Figure 2) consists of the *object setup* that defines the configuration for each object type (e.g., minimum number of nodes), and the *control setup* that includes the configurations for controllers, events and intent.

The *Verifier Operator* carries out the actual verification procedure given the inputs from the *Parser*. The *Verifier Operator* implements a scaling algorithm (introduced in §4.2) that conducts the verification in multiple cycles with incrementally larger and larger scale. For each cycle, the verifier operator first decides a *Profile*, which contains the setup of a particular scale, a subset of intent, and other verification configurations for that cycle. It then calls the *Model Generator* to generate a verification model, then receives the verification result from the *Model Checker* and finally decides if the results are ready to send to the users or if another cycle is needed. We describe this workflow in more detail in §4.3.

The *Model Generator* pre-processes the *Profile* to complete any missing elements with default values, discretizes all the values, and selects a subset of controller and event templates from the *Model Templates* that are related to the designated intent and configurations according to the *Profile*.

The *Model Templates* include the model logic with "holes" that will be filled in with configuration parameters from the *Profile* for the controllers and events Kivi supports. We have implemented these templates manually, where most controllers are modeled based on the Kubernetes source code. These independent templates enable the modularization of the verification process. We discuss more details in §5.

The *Model Checker* performs exhaustive verification for the generated model. It either assures desired properties will be preserved, or finds property violations and generates counterexamples demonstrating violations. We leverage the explicit-state model checker SPIN [51] in Kivi.

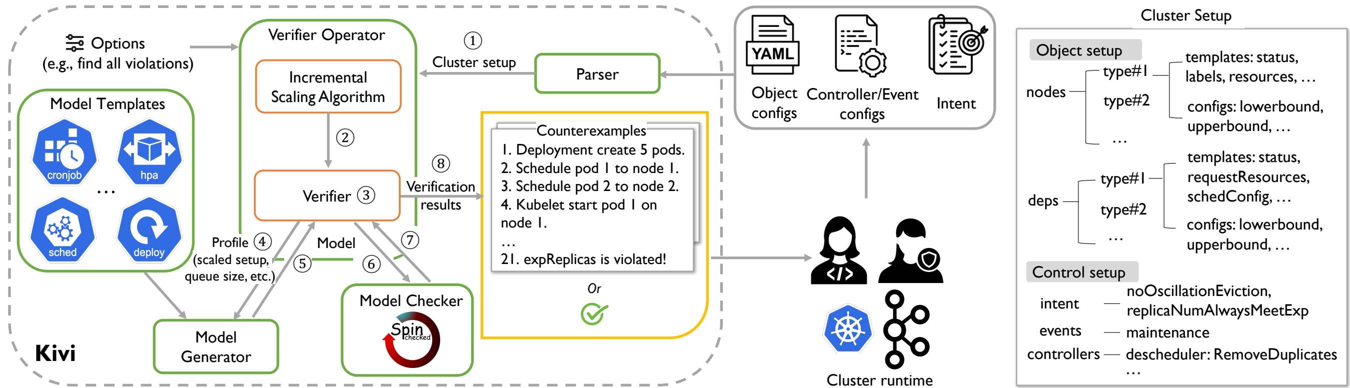


Figure 2: Kivi workflow. It consists of five main components: *Parser*, *Verifier Operator*, *Model Generator*, *Model Templates* and *Model Checker*. It receives the configurations and intent from the users and outputs verification results. A cluster setup consists of the *object setup* that defines the configurations for object types and the *control setup* that includes the configurations for controllers, events and intent.

4.2 Verifying Clusters at Small Scale

4.2.1 Incremental Scaling Algorithm Intuition

On the one hand, verifying cluster management systems has a daunting scale challenge. Clusters can reach hundreds or thousands of nodes, and many thousands of pods, bringing well-known state explosion problems [45]. Moreover, clusters typically run autoscalers and vary in size, so that users would be interested in not only one but a wide range of the cluster topologies that are generated from a cluster setup, and there may be multiple dimensions of scale (i.e., pod size and node size), so there is a very large number of possible cluster topologies even for a medium-sized cluster (tens of nodes).

On the other hand, the problem is actually simpler in the domain of cluster management systems, because *much of the complexity mainly lies in the cluster setup which does not grow with the size of the cluster*. All the cluster topologies are generated from the cluster setup, where each object is generated from a template in the setup rather than being crafted by hand. Because of this, the patterns and properties demonstrated in a subset of the topologies are often representative for all the topologies.

This leads to a key hypothesis, similar to observations in other domains [60, 74]: we posit that *if a cluster setup can violate an intent, then it can do so at relatively small scale.* This means that it will be sufficient to verify the cluster at small scale. If we cannot find a failure at small scale, we can conclude that, with high confidence, there are no violations at any scale.

Of course, it is possible to invent cluster setups that do not satisfy the hypothesis. The question is whether it is valid empirically – and if so, what scale is typically enough to ensure finding a violation? To answer that question, we evaluated the failure cases shown in Table 2. We use our verification workflow to explore all the cluster topologies to find the minimum scale in which the problems occur for each case, in the "Min Violation Scale" column. All cases show a violation at relatively small scale, with the maximum minimum violation scale being 3 nodes and 6 pods. (We discuss the study in detail in §6.1.)

4.2.2 Incremental Scaling Algorithm

The above intuition leads to our scaling algorithm: start at small scale, and gradually increase the size until we find violations or reach a *empirically sufficient scale (ESC)*, i.e., a scale at which we have high confidence that the violation should have been found if there is one. In addition to performance, smaller counterexamples will generally be easier for users to understand.

We describe the algorithm first assuming we are given the ESC, in the form $ESC = \langle n_{esc}, \theta_{esc} \rangle$, where n_{esc} is an integer number of nodes and θ_{esc} is a ratio of number pods to number of nodes, representing the scale that is considered empirically sufficient.

One complication is that there may be multiple node or pod types that are treated differently by controller configurations (e.g., one node type could be a set of nodes with the same available resources). Given that, do we need to explore all the combinations of the scale dimensions? We observe in practice that violations can appear or disappear depending on the exact scale values in each dimension (§6.1). We therefore check all combinations of node and pod scale dimensions, up to the ESC limits, to avoid compromising confidence. As we will see, Kivi is still sufficiently fast.

Our scaling algorithm is shown in Algorithm 1. For each node type i, it explores its scale from n_i^{min} to n_i^{max}, where by default $n_i^{min} = 0$ and $n_i^{max} = n_{esc}$ for all i. For each pod type i, it explores its scale from p_i^{min} to p_i^{max}, where by default $p_i^{min} = 0$ and $p_i^{max} = \theta_{esc} \cdot n$ where n is the number of nodes in the topology being tested.

This description omits a few details. We allow the user to configure minimum and maximum node and pod scales if desired. We skip any scale that results in trivial failure cases, meaning if the total node or pod size is 0 or generates non-

Algorithm 1 incremental scaling algorithm

```
1: procedure SCALING
2:     α = number of node types
3:     ϕ = number of pod types
4:     for (n_1,...,n_α) ∈ sort(Π_{i=1}^{α} {n_i^min,...,n_i^max}) do
5:         n = Σ_1^α n_i
6:         for (p_1,...,p_ϕ) ∈ sort(Π_{j=1}^{ϕ} {p_j^min,...,θ_esc · n}) do
7:             t = topology of n_1,...,n_α nodes and p_1,...,p_ϕ pods
8:             if not trivalCase(t) then
9:                 verifier(t)                      ▷ verify the topology
```

interesting failures (i.e., an excessive number of pods[6] cannot be scheduled onto the nodes). A complete description is in Appendix B.

To illustrate the scaling algorithm with a tiny example, suppose there are two types of nodes both with $n^{min} = 0$ and $n^{max} = 1$ and one type of pod with $\theta_{esc} = 2$. The incremental scaling algorithm then explores topologies of the form $\langle |N_1|, |N_2|, |P_1| \rangle$ in the following order, where $|N_1|$ and $|N_2|$ are the number of nodes of type 1 and type 2 respectively, and P_1 is the number of pods of type 1: $\langle 1,0,1 \rangle$, $\langle 0,1,1 \rangle$, $\langle 1,0,2 \rangle$, $\langle 0,1,2 \rangle$, $\langle 1,1,1 \rangle$, $\langle 1,1,2 \rangle$, $\langle 1,1,3 \rangle$, $\langle 1,1,4 \rangle$. Each scale will be verified in one cycle by the verifier operator (discussed in §4.3).

4.2.3 Determining ESC

To determine $ESC = \langle n_{esc}, \theta_{esc} \rangle$, Kivi leverages a library of known failure cases (in our experiments, those in Table 2) to empirically find at which minimum scale the failures have been found for all the cases.

We run Algorithm 1 with small changes to find the *minimum violation scale* for each failure case: 1) we set n_i^{max} and p_i^{max} to infinity to allow the algorithm to explore all possible scale; 2) we stop the algorithm after finding the first violation, which determines the minimum violation scale.

We then find the maximum, across all failure cases, of the total number of nodes (summed across all node types) among all minimum violation scales and define it as the *ESC* for nodes (n_{esc}). Because the number of pods that a cluster can host generally scales with the number of nodes (due to resource limits), we define the *ESC* for the pods (θ_{esc}) as the ratio of the total number of pods (summed across all pod types) to the number of nodes. We calculate this ratio as the maximum value of the ratios among all the minimum violation scale mentioned above. (Note that although these values are sums across types, when we use the *ESC* limits in Algorithm 1 we will test each individual type ranging up to the maximum, to be conservative.) We double both n_{esc} and θ_{esc} to provide more confidence.

Taking the failure cases in Table 2 as an example, we can find $n_{esc} = 3 \cdot 2 = 6$ and $\theta_{esc} = 3 \cdot 2 = 6$ (after doubling), where n_{esc} is from C8 and C3, and θ_{esc} is from C1 and C4. Users can also leverage their cluster failure history and execute Kivi's verification workflow without setting the bounds of the scaling algorithm to find the minimum violation scale for their failure cases and empirically customize the *ESC*.

While the intuition that failures appear at small sizes (if they appear at all) works for our collected failure cases, it may not be applicable to some other properties, like the failures related to proportions or absolute values[7]. If desired, rather than using *ESC* as a limit, users can use Kivi to check on a specific scale in a live cluster (by providing the cluster logs), a range of scale of interests, or even the entire scale-spaces. However, checking the entire scale-space may require significant time.

4.3 Workflow of Verifier Operator

Consider again Figure 2. The incremental scaling algorithm (②) takes in the cluster setup (①) and generates an array of scaled setups each of which is a setup for a particular scale, sorted from smallest to largest. The verifier (③) takes each scaled setup in order and generates a profile with the setup and other verification parameters. The parameters include options set by the users, e.g., stop at the first violation or find all, whether to enable randomness in search (optimization options in §5.2), or other internal parameters like the queue size for the controllers (§5.1) or verifying for a subset of intent at a time if applicable. The profile is sent to the model generator (④) to generate the SPIN model (⑤, see §5.1). Then the model checker is triggered to compile and verify the model (⑥) and sends the result back to the operator (⑦).

If a violation is found, the verifier can generate the counterexample by analyzing the error trace produced by SPIN, and present the result to the user (⑧). Or it can continue to find more violations if the user has chosen the "find all" option. If no violation is found, the operator picks the next profile (e.g., larger queue size or another subset of intent if applicable, or the next scaled setup) and repeats ③ - ⑦ again.

5 Model

In this section, we first discuss the overview of our model in §5.1. We then discuss a few optimization mechanisms to help improve run time performance in §5.2. We summarize the implementation details in §5.3.

5.1 Modeling Overview

It is challenging to effectively verify Kubernetes, as it has a complex and large implementation. Our main goal is to verify the *interactions* between control components as well as with events, rather than implementation details (i.e., error handling, data structures, APIs). Thus, instead of verifying source code, our model focuses on the high-level logic of

[6]The user can provide us an upper bound of the ratio of pods to nodes numbers in their cluster, or we can approximate it from the resource requests of pods and available resources of nodes.

[7]For example, a slowly progressed problem where if one node goes down in a 3-node cluster, it is $\frac{1}{3}$ of the nodes and is noticed immediately, while for 50 nodes, one node failure becomes $\frac{1}{50}$; or absolute value problems like an application can only go down if there are too many requests at the same time.

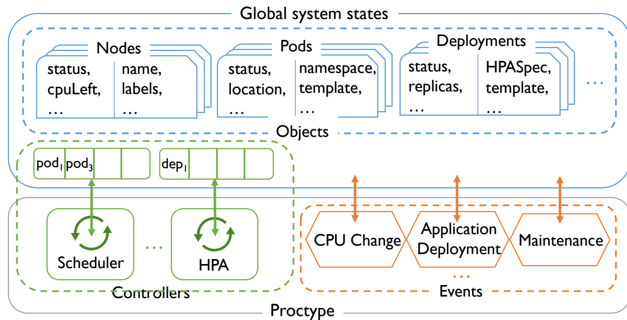


Figure 3: Kivi model structure.

the control components and captures the essential logic that affects the shared global states (i.e., the status of pods, nodes).

Kivi leverages the explicit state model checker SPIN [51]. We choose SPIN because it can help to effectively verify the interactions mentioned above, as SPIN targets the efficient verification of concurrent and asynchronous distributed software. SPIN provides a high-level language PROMELA to specify system behavior as non-deterministic automata. Each asynchronous process is modeled as a `proctype` process in PROMELA, and SPIN will exhaustively explore all possible interleavings between these processes through a depth-first search (DFS) of a state graph that is constructed from the supplied PROMELA model.

There are three main parts to model for Kubernetes: (1) the *controllers*, (2) the *objects*, including pods, nodes and workload, and (3) the *events*, including environmental events like CPU change and operational events like maintenance.

As introduced in §2, the API server provides a central place for controllers to query and manipulate the state of the objects. Our model is structured in a similar way as shown in Figure 3: we model *controllers* and *events* using the `proctype` in PROMELA, and model *objects* using global system state that is shared across all the `proctypes`. The design differs from Kubernetes's API-style interaction slightly – instead of listening to the updates from the API server, the controllers trigger each other. This avoids having a centralized procedure that needs to update its state after each control loop, which can unnecessarily add search depth and affect verification optimization like partial order reduction.

Modeling the objects. We model each kind of object using an array of customized `typedef` in the global states. Each object contains a set of attributes that are related to the system behaviors of interest. To detect issues that occur while state transitions are in progress, we model the intermediate states of the objects (e.g., pending and terminating states for pods). For example, as bringing up pods takes time, the scheduler can only change the state of the pod to pending, and other controllers may take action in the pod pending phase. Listing 1 shows a code snippet of a node definition.

```
typedef nodeType{
    short status;
    short cpuLeft;
    short numPod;
    ...
}
nodeType nodes[SIZE];
...
```

Listing 1: Defining nodes.

```
byte sQueue[MAX_SCHED_QUEUE];
short sTail, sIndex;
proctype scheduler() {
  atomic{
    do
        :: (sTail != sIndex) ->
        // control loop logic
    od;
  }
}
```

Listing 2: Defining scheduler.

Modeling the controllers and events. We model each controller in an event-driven loop. The queue is in the global system states that can be enqueued by other controllers and events when a control loop should be triggered. For example, when the deployment controller adds a new pod, it enqueues the pod into the scheduler's event queue, and a scheduling loop will be triggered. Listing 2 shows a code snippet for the scheduler. We model events in `proctype`, the same as controllers, so that SPIN can exhaustively explore the interleavings of the events and controllers non-deterministically.

Modeling the properties. We implement the properties introduced in §3.2. For the safety properties, Some (e.g., unexpected object lifecycles) are implemented as `assert` inserted into controllers, e.g., when there is no feasible node for scheduling. Some (e.g., unexpected object numbers and topology) involve temporal logic, e.g., after deployment is stable the pod number should always be more than expected. These properties are implemented as `proctypes`. Each `proctype` is implemented as a "monitor" process that runs independently and keeps its own states that track changes in the global states and use assertions to catch violations. These `proctypes` can be explored by SPIN before and after each step.

For the oscillation (liveness) property, we find that the default way in SPIN is often slow, because it only considers a loop as when all the global variables appear the same repeatedly, which is unnecessarily strict. We instead only check if the loop has appeared in key relevant variables. However, as we are not checking full system state, it is possible that this subset of state could recur without causing permanent oscillation since other parts of the system state may evolve. Thus, to weed out most such false positives, we look for multiple occurrences. For example, when checking on the eviction and scheduling cycle, we check if an eviction flag and a deploy flag for a deployment appear in turns 3 times in a row. One can increase this number to gain confidence. Here we believe it is worth alerting users if events repeat 3 times, even if the loop is not infinite.

Modeling time. Time is an important variable in Kubernetes. For example, HPA runs periodically (every 15 seconds by default), and CPU usage of a pod changes after the initialization phase of a certain time. Modeling time can help avoid generating non-interesting counterexamples (i.e., failures caused by HPA reacting too fast), and further help to avoid unnecessary interleavings between `proctype` to improve runtime performance. SPIN does not provide a built-in notion of time. We model time as a variable $\mathcal{T}$ in the global system state. Each `proctype` i has its own local time variable τ_i representing the last time of execution, and it can only execute its next control loop or event logic when $\tau_i + \delta_i \leq \mathcal{T}$, where δ_i denotes the time interval between control loops or events. $\mathcal{T}$ is updated accordingly after each execution of an event or a control loop.

Modeling quantitative values. Many controllers use integers or real numbers, e.g., when calculating the average CPU usage in HPA and calculating scores in the scheduler. There are also strings like labels on nodes. SPIN can model integers yet not real numbers or strings. We pre-process and discretize any string into integers. For real numbers, we observe that they mostly appear either in configurations with two decimals (e.g., average CPU utilization threshold in HPA) or in division. We hence convert each real number r into $\lfloor r \cdot 100 \rfloor$. For division, luckily, the final results of most calculations are integers (e.g., replica numbers proposed by HPA).

5.2 Optimization

SPIN implements DFS to search for violations. It stores the visited state spaces to implement strategies (e.g., partial order reduction) to reduce its search space. We summarize three major aspects that can affect the run time: (1) *the size of the global states*, where large global states can cause huge memory usage during search and lead to potential out-of-memory and memory swapping; (2) *concurrency*, where if there are too many processes that can be chosen from for each search step, the search spaces can be huge; (3) *search depth*, where large search depth can lead to a lot more search spaces.

We discuss a few heuristic mechanisms in our design to improve run time performance according to the three aspects.

Clearly defining small sets of mutating global variables. To reduce the state size, we carefully pick the attributes that are related to the properties of interest. We divide these global states into two sections for performance[8]: *a stable section* that will not be changed at runtime and is not tracked by SPIN, like pod templates, node labels and names, HPA specification; and *a mutating section* that keeps changes and is tracked by SPIN, including status, resource usage, pod locations, etc.

Reducing concurrency. Some controllers are deployed as one instance per object, e.g., each node has a Kubelet controller, and each deployment has its own HPA instance if enabled. The size of the concurrent controller instances can increase relative to the number of objects and the search space can explode exponentially. Instead, we model these multiple instances of a controller as a single process. In most cases, it is safe to model in this way, as each instance operates on its own object (e.g., configurations are defined per deployment). In addition, we model each control loop into an atomic block to avoid interleaving between other controllers in the middle of its execution with the observation that the execution time of one control loop is negligible. We still can check on the interleaving between multiple control loops.

Reducing search steps. While some controllers (e.g., scheduler) are implemented with queuing designs and some are not (e.g., HPA runs periodically), we modeled all of them using an event-driven schema to avoid unnecessary search when controllers are not involved, e.g., HPA would not be triggered until there are resource changes. We also reduce the execution of back-to-back control loops or events into a single event, e.g., if multiple pods change their CPU back-to-back, we change all of them at the same time. Another example is how we check oscillation property as discussed in §5.1.

More options. SPIN provides additional options to improve its runtime, e.g., partial order reduction enabled by default, searching up to a bounded depth and time, state compression (e.g. leveraging bloom filter), and searching with random seeds. Users can turn on these options through Kivi.

5.3 Implementation

Though our modeling mechanisms are rather general, due to limited time, we focus on implementing a selected set of controllers (and their features) and objects (and their attributes) that are most common and representative in terms of the failure types they are involved in (e.g., need to be able to represent how we verify for each property category) and the required verification techniques (e.g., choosing HPA that require modeling time and real values). We carefully pick five built-in controllers and one add-on controller (the descheduler)[9]. For each controller, we understand all the details of the source code and only model the most essential details needed to accurately capture how the controller manipulates the shared objects and interacts with the cluster. We now do not model when controllers themselves encounter issues. For example, we omitted error handling, retries, and handler registration. Some controllers (e.g., Kubelet) have too many low-level details (e.g., manage the pod image) that are unrelated to the properties of interest while their higher-level functionalities are simple, so we model them according to their documentation [32, 33]. We also implement a few common events that may result in interesting failure cases. Table 4 shows the controllers and events that we have modeled. We currently implement 7 properties spanning all four categories in §3.2 that focus on the objects of pods and deployments, such as `checkBalanceNode(k)` which ensures the pods in a

[8]SPIN stores tracked global states in its search stack. Unnecessarily storing and mutating these states can result in increased memory and time.

[9]We focus on Kubernetes v1.26.0 and Descheduler v0.27.1. Our code is publicly available on Github [38].

Controllers	**Features/Plugins Beyond Basic Framework**	**Source code?**	**Source code LoC**	**Model LoC**
Deployment/Replicaset Controller	ReCreate, RollingUpdates	●	781	199
Scheduler	NodeName, NodeAffinity, TaintToleration, NodeResourcesFit PodTopologySpread	●	3826	783
Descheduler	RemovePodsViolatingTopologySpread, RemoveDuplicates	●	1306	471
HPA	Metric type Utilization and Values	●	1157	222
Kubelet	N/A	○	2108	90
Node Controller	Taint Manager	◐	435	86
Events	**Description**			**LoC**
CPU Change/CPU Pattern Change	The CPU usage of the pods can change randomly or in a pre-defined pattern.			86
Kernel panic	High resource usage can cause kernel panic and node become unhealthy.			25
Node Failure	Node can fail non-deterministically.			6
Apply/Create Deployment	Users deploy their deployment configured in YAML files.			126
Scale Deployment	Users scale up their deployment on the fly.			11
Maintenance	Users put down the nodes for maintenance and put them back when updates are done.			37

Table 4: Kivi implementation of controllers and events. We have implemented a subset of features or plugins beyond the basic framework for each controller. We label whether we derived the model from the source code (or from documentation otherwise). We calculate the lines of code (LoC) for both the controller source code and our model, which includes log printing and excludes blank lines and comments. For our model, all the code shares a utility library of 311 LoC which is not included in the table. For the controller source code, we approximately include the main framework and the features that we modeled. In particular, for Kubelet, we only count the Kubelet main file; the actual Kubelet implementation is much more complicated as it handles lower-level Linux details like network, volumes, journal, and more – which is one reason that our model is simplified.

deployment are balanced across nodes with skew of no more than k, `checkExpReplicas(k)` which checks if the number of pods is $\geq k$. Modeling other objects (i.e., nodes and traffic) could leverage the same framework in future work.

6 Evaluation

We seek to answer four main questions in our evaluation: (1) Can we use realistic failures to validate our intuition and scaling approach in §4.2? (2) What is the performance and scalability of Kivi? (3) How accurate is Kivi? (4) Can Kivi find new problems in Kubernetes controllers?

To answer these questions, we evaluate Kivi on a test suite with the 8 realistic failure cases as shown in Table 2. These cases are representative in terms of the involved controllers (and their features) and events, the properties (covering all four summarized in §3.2), and the type of failure reasons (covering all three as summarized in §3.1).

To be able to evaluate the performance of our system, we need to generate test suites of various sizes. All these 8 failure cases are either discussed using a fixed size of cluster or are vague on details of their configurations. We fill in reasonable details to make each example complete and parameterized to scale up while preserving its failure pattern. To affirm our understanding of each case, we have reproduced all these cases in a Kubernetes cluster.

To be able to evaluate Kivi under situations both with and without property violations, we further extend our test suite by generating non-violation configurations for each failure case. We make small changes to the configurations to keep the main skeleton the same while avoiding failures.

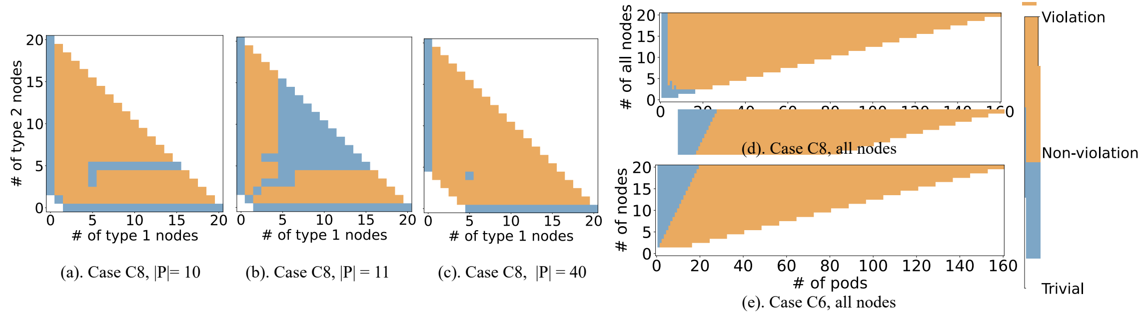
We perform all experiments on a 2021 Macbook Pro with an M1 Max processor and 64GB RAM and reproduction on VM having 8 CPU cores of Intel(R) Xeon(R) CPU E5-2630L 0 @ 2.00GHz and 8GB RAM.

Failure Case Reproduction. We leverage Kind [29] to build a cluster with adjustable size (i.e., 1 master and 1-3 workers) on a local machine. We successfully reproduced 7 of the 8 cases, but skipped C4 as it involves kernel panic that is hard to emulate. We used reproduction to further affirm our scaling observation (§6.1) and evaluate Kivi's accuracy (§6.3). The reproduction code is available on Github [37].

6.1 Empirical Study

In §4.2.2, we introduced the methodology to find the empirically sufficient scale (ESC) for the incremental scaling algorithm. Here, we empirically find the ESC for the collected failure cases in Table 2, and discuss how our study affirms our intuition of §4.2.1.

We run the verification workflow with the scaling algorithm without bounding it by the ESC: we explore any combination of sizes that do not have trivial failures (explained in §4.2.2). The last two columns in Table 2 show the minimum violation scale and whether such scale is smaller than the scale in the original failure reports. Our results show that intent violations consistently appear at relatively small scale: the maximum (across our test cases) of the minimum scale needed to demonstrate a violation is only 3 nodes and 6 pods. Indeed, in 6 of 7 test cases, Kivi found violations at even smaller scale than the scale in the original report, and we confirmed these by running the configurations in a real Kubernetes cluster. This



(a). Case C8, |P|= 10 (b). Case C8, |P| = 11 (c). Case C8, |P| = 40 (d). Case C8, all nodes (e). Case C6, all nodes

Figure 4: Heatmap of the verification results at various scale. (e) shows the result for Case C6 that scales against nodes and pods. (a)-(d) shows the result for Case C8. C8 contains two types of nodes and we test on all the combinations of $|N_1|$, $|N_2|$, and $|P|$ until reaching the scale of trivial failures. (a)-(c) each shows a heatmap against $|N_1|$ and $|N_2|$ with varying $|P|$. (d) shows the result against $|N|$ and $|P|$, where if we find a violation in any ($|N_1|$, $|N_2|$), we count $|N_1|+|N_2|$ as a violation point.

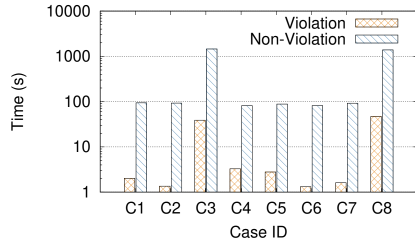


Figure 5: Kivi performance. Most cases can finish within 100s and all cases finish within 25mins. The run time is proportional to the number of scaled setups.

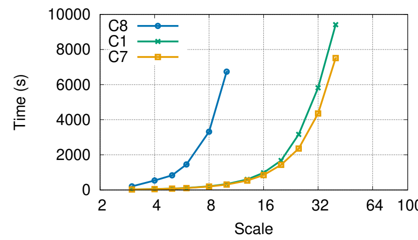


Figure 6: Performance without scaling algorithm. Times out at medium sizes. (Note: this is not the actual performance for Kivi. Lines stop at time-out scale.)

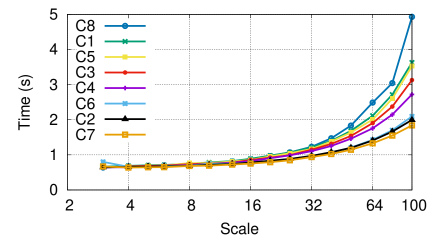


Figure 7: Internal test on the model performance: non-violation cases range from $|N|=3$ to $|N|=100$. Each data point is a single run of SPIN at a specific size.

also demonstrates Kivi's ability to minimize counterexample sizes to simplify debugging. Based on this result, doubling to provide additional confidence, we set $n_{esc}=3\cdot 2=6$ (extracted from C3 and C8) and $\theta_{esc}=3\cdot 2=6$ (from C1 and C4) for our evaluation.

To further understand how failures appear, we sweep a range of combinations of node and pod sizes for two representative cases with diverse patterns in Figure 4. Each cell in the heatmaps is colored based on the result at that scale: *violation*, *non-violation*, or *trivial failure*. $|N_i|$ denotes the number of nodes of type i and $|P_i|$ denotes the number of pods of type i. $|N|$ and $|P|$ denote the total number of nodes and pods, respectively. From these graphs, we make a few observations: (1) Violations consistently appear for sufficiently large $|N|$ and $|P|$, again affirming that we can use small scale to find violations that would appear large scale. (2) The exact combinations of the $|N_i|$ and $|P|$ dimensions matter. There are many different patterns – generally more violations at larger scale, some clear relationships between $|N|$ and $|P|$, but also complex patterns can emerge. This justifies the choice in our incremental scaling algorithm to explore all combinations up to the ESC.

6.2 What is the Performance of Kivi?

Figure 5 shows the performance for Kivi under both violations and non-violations for the test suite. The result shows that Kivi verifies most cases within 100 seconds and all cases within 25 minutes. Some non-violation cases take longer, as the run time is proportional to the number of scaled setups that need to be tested: if a case contains various sizes of objects (i.e. C8 and C3 contain 1764 scaled setups) or is being tested for non-violation where the verification needs to explore all scale, it can take longer. Figure 6 shows the performance without the incremental scaling algorithm for three representative cases: the verification times out (>10000 sec) even at moderate scale (≤ 50 nodes).

To provide a sense of the performance of the underlying model itself, we do an internal measurement on the non-violation version of the test suite at a few specific scale ranging from $|N|=3$ to $|N|=100$, where 100 is a large enough scale for most clusters. Figure 7 shows our model performs well even at large scale. Note, however, this shows performance for a single run of SPIN (a single topology) at each scale, rather than checking a whole range of combinations of scale parameters which we require for high coverage.

6.3 Is Kivi accurate?

We evaluate Kivi on both violation or non-violation versions of the test suite. Kivi successfully found the correct violation for all the violation cases, while reporting no failures for all the non-violation cases.

To further evaluate the accuracy of our model in terms of controller interactions, we compare the counterexamples Kivi generated against the real Kubernetes event logs and see if the order of actions (i.e., a control action, an event) matches with each other. We convert each action in the related reproduc-

tion logs and verification counterexamples into a canonical representation. We skip the actions that we don't model, like pulling an image, HPA failure due to unavailable metric server at bootstrapping phase. We calculate the matching rate as the number of matched reproduction actions plus the number of matched verifier actions divided by the total number of both actions. Table 5 shows the results.

	Non-deterministic cases			Deterministic cases
Case ID	C1	C2	C6	C3, C5, C7, C8
Matching rate	81.6%	97.8%	100%	100%

Table 5: Actions matching rates.

Among all the cases, the cases with only deterministic events (C3, C5, C7 and C8) match 100%. The other cases contain non-deterministic events: C1 involves CPU changes, C2 and C6 involve operational events. Among them, matching rates of C2 and C6 are near or at 100%[10] as the operational events happen at slower speeds and don't interact much with the controllers. For C1, 43.8% of the mismatches are due to extra CPU change events in the reproduction logs. The rest are due to an inaccuracy in HPA modeling, where we do not model stabilization windows when the HPA controller is scaling down, causing the verification logs to scale down faster than the reproduction, though the final stable number is the same. Note that for non-deterministic cases in general, the accuracy as evaluated here is pessimistic because Kivi's reported event sequence could be entirely valid but not the one that happened to occur in our runs of Kubernetes.

These results show good accuracy of Kivi in verifying realistic issues and modeling the interactions of controllers.

6.4 New Issues Found

Though we mostly focus on misconfiguration issues, we used Kivi to find two new issues in the implementation of the Kubernetes descheduler. We identified these failures in the process of running Kivi on failure scenarios C5 and C8. Although those failure scenarios are known, the discussions [24] had focused on configuration work-arounds and had not uncovered the root causes in the controllers.

RemoveTopologySpreadConstraint does not consider all constraints together and can mistakenly evict pods. When there are multiple `PodTopologySpread` constraints, the `RemoveTopologySpreadConstraint` plugin decides which pods to evict per constraint instead of solving all the constraints together. This results in a sub-optimal decision to evict more pods. C8 is a failure caused by this issue. If implemented correctly, `RemoveTopologySpreadConstraint` should have known that the two constraints in C8 cannot both be satisfied and hence decided not to evict any pods.

[10] C2 has one event mismatched because the verifier deleted a pod in a different node than reproduction logs, though the two nodes are symmetric.

RemoveDuplicates does not respect node resources and can mistakenly evict pods. When the `RemoveDuplicates` plugin decides which pods to evict, it first collects the available nodes that can serve the duplicated pods. If there are fewer available nodes, fewer pods are evicted. However, this plugin fails to consider the available resources and mistakenly counts the occupied nodes as available, resulting in more pods being evicted than desired. C5 is a failure caused by this issue. If the `RemoveDuplicates` had considered node resources, no pods would be evicted.

We have submitted both issues to the Descheduler project on Github [36, 39].

7 Discussion and Future work

Limitation in accuracy. We have made several approximations that can potentially affect the accuracy of Kivi.

From Kubernetes code to model. As shown in Table 4, our model uses fewer LoC than the original Kubernetes code. While this makes verification tractable, it comes at the cost of potential deviations from the code (§5.2 and §5.3 discuss some, and we enumerate more in Appendix C). With more engineering effort, our model can be brought closer to real implementations. To implement the model of a new controller, depending on programmer familiarity of the controller, one would need to read the documentation or the controller source code, decide which parts are essential to model and simplify the workflow if possible.

However, there are always gaps that are a fundamental result of verifying a model rather than code. Ultimately, verification of a model (as in Kivi) is complementary to testing the real implementation: while testing avoids the need to model and its inevitable imperfections, verification can provide much higher coverage of all possible scenarios.

Incremental scaling algorithm. Kivi may miss some failures that only manifest at large scale as discussed in §4.2.2. One can run Kivi at a specific large scale to find these issues, though there may be a performance penalty.

Limitation in the failure dataset. Our failure cases may not be large enough to serve as a complete quantitative study and to demonstrate that Kivi can empirically extract the ESC for arbitrary clusters. Although there is plentiful open source *code* available online, we need real K8s *configurations* which are much less frequently released. Besides searching online, we have talked with dozens of operators, and they confirmed the types of failures we discussed are big concerns, yet due to security concerns or limited operational practices (i.e., no postmortem reports), they cannot share more details. Hopefully people can start to see the needs of open-sourcing their configs, and future work can leverage a larger dataset.

Limitation in scalability and future work. While Kivi's scaling algorithm performs well on our test setups, Kivi may not scale to clusters with a large degree of node and deployment type diversity, where the number of scaled setups to

verify can grow exponentially with the types. However, we have found that empirically we need to verify only a small number of types. For the nodes, it is recommended that a cluster should minimize the number of node groups (i.e., set of nodes that share the same properties) to ensure the cluster autoscaler can perform well on large clusters [31]. For the deployment, we find that most failures happened for a single deployment (i.e., most configurations are defined per deployment) and hence verifying one deployment at a time while removing the resources taken by others is often enough.

If users are still interested in operating more types of nodes or interested in the interactions between deployments, we also propose a few optimizations to explore as future work:
(1) Divide and conquer. For multiple deployments, we can still verify one deployment at a time, and abstract the impact of other deployments together into a small set of arbitrary "external" events (e.g., CPU changes on shared nodes).
(2) Partial order reduction on symmetric objects. Additional partial order reduction mechanisms can be explored to reduce the search branches that explore the symmetric objects.
(3) Multi-core. One can leverage multi-core computation to parallelize the verification for various scale.
(4) Faster ramp up in scaling algorithm. Instead of increasing the size by 1 at a time, we could add it by 2 or even multiply it by 2. As shown in the heatmap in Fig. 4, increasing the speed can still potentially catch the failure with high confidence.

8 Related Work

Verification for cluster management systems. Sun et al. [65] present Anvil, a framework for developing controller implementations and verifying if the controller correctly implements the property of eventual stable reconciliation (ESR). Our work complements their work in several ways: 1) while Anvil verifies single controller implementation logic at the executable level, Kivi addresses issues arising from the complex interactions among multiple controllers at the model level, and it can detect misconfiguration issues across various cluster topologies; 2) while Anvil focuses on the ESR property, Kivi checks on sets of commonly overlooked properties related to unintended or pathological behaviors.

A couple of other works have applied formal methods to the domain of cluster management systems. Turin et al. [68] demonstrate a Kubernetes formal model yet it is based on much simplified assumptions of controllers and does not test it for verification. Liu et al. [56] presents a proof-of-concept verification approach yet it models a couple of selected failure scenarios rather than a comprehensive implementation of controllers. Flux [46] applies verification to serverless yet focuses on idempotence properties from the application perspective. Compared to these works, Kivi is a more comprehensive verification system based on a novel set of properties derived from real Kubernetes issues.

Verification for systems. Verification has been successfully applied to many distributed systems (e.g., [42, 43, 49, 50, 61, 62, 67, 69, 72, 77, 79]) as well as in networking (e.g., [40, 41, 53, 73, 76]). While cluster management systems are also a distributed system, existing work is insufficient to verify them. First, many works focus on verifying specific protocols (e.g., Paxos, BGP) instead of dynamic closed-loop controllers. Second, many works rely on theorem proving, which involves a great deal of human effort and can be hard to apply to this domain with ever-evolving large-scale implementations. Third, the properties for this domain (§3.2) are quite different from what these works focus on (e.g., crash-safety, integer overflow, network reachability). However, Kivi shares some of the underlying technologies with distributed verification work such as the use of model checking techniques [48, 52, 54, 71].

Cluster management reliability. Several works seek to improve the reliability of cluster management systems [47, 55, 64, 70]. Sieve [64] presents testing tools for state reconciliation issues in customized Kubernetes controllers. Häyhä [55] presents a tool to detect intra-update sniping vulnerabilities using dataflow graph analysis. Kivi targets different issues and is complementary to these works in improving various perspectives of the cluster management systems.

9 Conclusion

We present Kivi, the first system for verifying cluster management system controllers and configurations. Kivi empirically demonstrates the insight that failures that happen at large scale can manifest at small scale and leverages it to tackle the scalability challenge. Kivi has shown good performance and accuracy in verifying realistic failure and showcases two new issues in Kubernetes controller.

References

[1] Classic Swarm: a Docker-native clustering system. https://docs.docker.com/engine/swarm/.

[2] Kubernetes Failure Stories. https://codeberg.org/hjacobs/kubernetes-failure-stories.

[3] VM vSphere. https://www.vmware.com/products/vsphere.html.

[4] Expected behavior for Deployment replicas with HPA during update #25238. https://github.com/kubernetes/kubernetes/issues/25238, 2016.

[5] MOVING THE ENTIRE STACK TO K8S WITHIN A YEAR – LESSONS LEARNED. https://www.youtube.com/watch?v=tA8Sr3Nsx1I&t=1575s&ab_channel=CodeSpaceITeducation, 2018.

[6] Removing spec.replicas of the Deployment resets replicas count to single replica #67135. https://github.com/kubernetes/kubernetes/issues/67135, 2018.

[7] 10 Ways to Shoot Yourself in the Foot with Kubernetes, #9 Will Surprise You. https://youtu.be/QKI-JRs2RIE?t=1183, 2019. Incident #4 and #7.

[8] Build Errors of Continuous Delivery Platform . https://github.com/zalando-incubator/kubernetes-on-aws/blob/dev/docs/postmortems/jun-2019-kubelet-qps.md, 2019.

[9] Did Kubernetes Make My p95s Worse? https://youtu.be/QXApVwRBeys?t=727, 2019. At 727s, 987s and 1245s.

[10] How a Production Outage Was Caused Using Kubernetes Pod Priorities. https://grafana.com/blog/2019/07/24/how-a-production-outage-was-caused-using-kubernetes-pod-priorities/, 2019.

[11] Intermittent delays in Kubernetes. https://medium.com/techmindtickle/intermittent-delays-in-kubernetes-e9de8239e2fa, 2019.

[12] Kubernetes' dirty endpoint secret and Ingress. https://philpearl.github.io/post/k8s_ingress/, 2019.

[13] Kubernetes Failure Stories, or: How to Crash Your Cluster. https://www.youtube.com/watch?v=LpFApeaGv7A&ab_channel=ContainerDays, 2019. Incident #0 and #2.

[14] On Infrastructure at Scale: A Cascading Failure of Distributed Systems. https://danveloper.medium.com/on-infrastructure-at-scale-a-cascading-failure-of-distributed-systems-7cff2a3cd2df, 2019.

[15] Outage post-mortem. https://updates.moonlightwork.com/outage-post-mortem-87370, 2019.

[16] Replicaset controller bug: continuously creating pod to tainted nodes #75913. https://github.com/kubernetes/kubernetes/issues/75913, 2019.

[17] CPU limits and aggressive throttling in Kubernetes. https://medium.com/omio-engineering/cpu-limits-and-aggressive-throttling-in-kubernetes-c5b20bd8a718, 2020.

[18] DNS issues in Kubernetes. Public postmortem 1. https://medium.com/preply-engineering/dns-postmortem-e169efd45afd, 2020.

[19] Kubernetes: Make your services faster by removing CPU limits. https://erickhun.com/posts/kubernetes-faster-services-no-cpu-limits/, 2020.

[20] Kubernetes Outages with real-world case studies. https://kubevious.io/blog/post/kubernetes-outages-with-real-world-case-studies, 2020.

[21] 10 Ways to Blow Up Your Kubernetes. https://www.youtube.com/watch?v=1vfLuDBhABA&t=1516s&ab_channel=ShareLearn, 2021.

[22] How a couple of characters brought down our site. https://medium.com/@SkyscannerEng/how-a-couple-of-characters-brought-down-our-site-356ccaf1fbc3, 2021.

[23] Resource Requests and Limits Under the Hood: The Journey of a Pod Spec. https://www.youtube.com/watch?v=WB3_sV_EQrQ&ab_channel=CNCF%5BCloudNativeComputingFoundation%5D, 2021.

[24] Incorrect work with HPA (when replicas more than number of nodes) #921. https://github.com/kubernetes-sigs/descheduler/issues/921, 2022.

[25] Scheduler Configuration. https://kubernetes.io/docs/reference/scheduling/config/, 2022.

[26] DaemonSet. https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/, 2023.

[27] Example: conflicting topology spread constraints. https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/#example-conflicting-topologyspreadconstraints, 2023.

[28] Job. https://kubernetes.io/docs/concepts/workloads/controllers/job/, 2023.

[29] Kind . https://kind.sigs.k8s.io/, 2023.

[30] Kubernetes. https://kubernetes.io/docs/concepts/overview/, 2023.

[31] Kubernetes Cluster Autoscaler. https://aws.github.io/aws-eks-best-practices/cluster-autoscaling/#reducing-the-number-of-node-groups, 2023.

[32] Kubernetes kubelet . https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/, 2023.

[33] Node Controller . https://kubernetes.io/docs/concepts/architecture/nodes/#node-controller, 2023.

[34] StatefulSets. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/, 2023.

[35] You Broke Reddit: The Pi-Day Outage. https://www.reddit.com/r/RedditEng/comments/11xx5o0/you_broke_reddit_the_piday_outage/, 2023.

[36] The gettargetnodes in removeduplicates does not respect node resources and can mistakenly evict pods. https://github.com/kubernetes-sigs/descheduler/issues/1237, 2024.

[37] k8s-failure-reproduction. https://github.com/gangmuk/k8s-failure-reproduction, 2024.

[38] Kivi: verifying your Kubernetes clusters. https://github.com/bingzheliu/Kivi, 2024.

[39] Topologyspreadconstraint calculation has issues that can mistakenly evict pods. https://github.com/kubernetes-sigs/descheduler/issues/1219, 2024.

[40] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 1–16, 2021.

[41] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 155–168. ACM, 2017.

[42] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in amazon S3. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 836–850. ACM, 2021.

[43] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with perennial. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 243–258. ACM, 2019.

[44] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

[45] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011.

[46] Haoran Ding, Zhaoguo Wang, Zhuohao Shen, Rong Chen, and Haibo Chen. Automated verification of idempotence for stateful serverless applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 887–910, Boston, MA, July 2023. USENIX Association.

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

[48] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 265–278, New York, NY, USA, 2011. Association for Computing Machinery.

[49] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 99–115. USENIX Association, 2020.

[50] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17. ACM, 2015.

[51] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[52] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, Cambridge, MA, April 2007. USENIX Association.

[53] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russell J. Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX*

Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015, pages 59–72. USENIX Association, 2015.

[54] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, Broomfield, CO, October 2014. USENIX Association.

[55] Julien Lepiller, Ruzica Piskac, Martin Schäf, and Mark Santolucito. Analyzing infrastructure as code to prevent intra-update sniping vulnerabilities. In *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II 27*, pages 105–123. Springer, 2021.

[56] Bingzhe Liu, Ali Kheradmand, Matthew Caesar, and P Brighten Godfrey. Towards verified self-driving infrastructure. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 96–102, 2020.

[57] Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Jiaxin Cao, Sri Tallapragada, Nuno P. Lopes, Andrey Rybalchenko, Guohan Lu, and Lihua Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 599–613, New York, NY, USA, 2017. Association for Computing Machinery.

[58] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[59] Sixiang Ma, Fang Zhou, Michael D. Bond, and Yang Wang. Finding heterogeneous-unsafe configuration parameters in cloud systems. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 410–425, New York, NY, USA, 2021. Association for Computing Machinery.

[60] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding {Crash-Consistency} bugs with bounded {Black-Box} crash testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 33–50, 2018.

[61] Rian Shambaugh, Aaron Weiss, and Arjun Guha. Rehearsal: a configuration verification tool for puppet. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 416–430. ACM, 2016.

[62] Helgi Sigurbjarnarson, Luke Nelson, Bruno Castro-Karney, James Bornholt, Emina Torlak, and Xi Wang. Nickel: A framework for design and verification of information flow control systems. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 287–305. USENIX Association, 2018.

[63] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, November 2020.

[64] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 143–159, Carlsbad, CA, July 2022. USENIX Association.

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

[66] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 787–803. USENIX Association, November 2020.

[67] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In Robbert van Renesse and Nickolai Zeldovich,

editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 866–881. ACM, 2021.

[68] Gianluca Turin, Andrea Borgarelli, Simone Donetti, Einar Broch Johnsen, Silvia Lizeth Tapia Tarifa, and Ferruccio Damiani. A formal model of the kubernetes container framework. In Tiziana Margaria and Bernhard Steffen, editors, *Proc. 9th International Symposium on Leveraging Applications of Formal Methods (ISoLA 2020)*, volume 12476 of *Lecture Notes in Computer Science*, pages 558–577. Springer, 2020.

[69] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. PLDI '15, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[70] Xiaohan Yan, Ken Hsieh, Yasitha Liyanage, Minghua Ma, Murali Chintalapati, Qingwei Lin, Yingnong Dang, and Dongmei Zhang. Aegis: Attribution of control plane change impact across layers and components for cloud systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 222–233. IEEE, 2023.

[71] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*, Boston, MA, April 2009. USENIX Association.

[72] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 701–718. USENIX Association, 2020.

[73] Farnaz Yousefi, Anubhavnidhi Abhashkumar, Kausik Subramanian, Kartik Hans, Soudeh Ghorbani, and Aditya Akella. Liveness verification of stateful network functions. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 257–272. USENIX Association, 2020.

[74] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 249–265. USENIX Association, 2014.

[75] Xinhao Yuan and Junfeng Yang. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1141–1156, New York, NY, USA, 2020. Association for Computing Machinery.

[76] Yifei Yuan, Soo-Jin Moon, Sahil Uppal, Limin Jia, and Vyas Sekar. Netsmc: A custom symbolic model checker for stateful network verification. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 181–200. USENIX Association, 2020.

[77] Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 32:1–32:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[78] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. Understanding and detecting software upgrade failures in distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 116–131, New York, NY, USA, 2021. Association for Computing Machinery.

[79] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helpers for verifying the atomfs file system. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 259–274. ACM, 2019.

A More failure cases

We list more failure cases in the extended Table 6.

Case ID	Description	Properties	Reasons
C9 [9]	Pods are being scheduled into the same node due to the image locality plugins outweighs all other scoring plugins	Unexpected Topology	Non-trivial interactions between components in a single controller
C10 [9]	Network issues caused some pods to become unreachable (yet still healthy) and in turn their average CPU usage had dropped, causing the HPA to scale down pods and further reduce the capacity.	Unexpected Object Numbers	Non-trivial interactions between controllers and events
C11 [7]	Nodes that were supposed to only host the App pods failed to set the right taint, causing new Daemonset pods to run on every node while App pods failed to be scheduled.	Unexpected Object Lifecycles	Non-trivial interactions between components in a single controller
C12 [10]	The priorities of the pods in the production cluster were not set correctly and the deployment of a new cluster with higher priority triggered a cascading failure to preempt all the production pods.	Unexpected Object Lifecycles	Non-trivial interactions between controllers and events
C13 [12]	The ingress controller kept sending traffic to the pods that are pending deletion by the deployment controller.	Unexpected Object Lifecycles	Non-trivial interactions between controllers
C14 [9]	The scheduler scheduled more pods on one node over the other, while the ingress controller scheduled the traffic randomly. This caused more traffic imbalance across nodes.	Unexpected Topology	Non-trivial interactions between controllers
C15 [7]	One availability zone (AZ) had poor node availability, causing the node autoscaler to scale up nodes in another AZ. Later the nodes came back, causing the autoscaler to scale down the newly created nodes and pods failed to be scheduled due to bounded volumes to deleted nodes.	Unexpected Object Lifecycles	Non-trivial interactions between controllers and events
C16 [14]	When a system component became unavailable, the logging components all woke up and cumulatively consumed high resources, causing the nodes to report unhealthy, at which time the scheduler moved the workload to the healthy node. The previous unhealthy nodes became healthy as the workload was moved while the previous healthy node became unhealthy, and the cycle perpetuated.	Oscillation / Unexpected Object Lifecycle (object is the controller)	Non-trivial interactions between controllers and events

Table 6: Extended Failure Cases.

B Incremental Scaling algorithm

B.1 Terminology

We define a *cluster setup* $C = \langle CS, OT \rangle$ as the configuration for the cluster. The *control setup* CS includes the configurations for controllers, event assumptions, and users' intent. The *object setup* OT defines the configuration for types of objects, where $OT = \langle NT, DT \rangle$,[11] with NT denotes the set of node types and DT denotes the set of deployment types. Each node type $NT_i = \langle template, config \rangle$ consists of a template of object attributes (e.g., status, labels) and configs that define the lower bound o_i^l and upper bound o_i^u for replica ranges of object type o_i. DT is similar to NT yet their templates contain different sets of attributes (e.g., resource requests, scheduling configs). Figure 2 illustrates a graphic example of *cluster setup*. Kivi derives the *cluster setup* from users' inputs.

At run time, each *cluster setup* C can generate a set of *cluster topologies* T^C with different numbers of replicas for each type. We define a *cluster topology* $t \in T^C$ as $t = \langle N, P \rangle$. N is the set of nodes $N = N_1 \cup N_2 \cup \cdots$, where N_i denotes the set of nodes of type i. P is the set of pods $P = P_1 \cup P_2 \cup \cdots$, where P_i denotes the set of pods generated from the deployment of type i. The scale of a topology t can be defined as a vector of the sizes for all types of objects $s_t = \langle |N_1|, |N_2|, \cdots, |N_\alpha|, |P_1|, |P_2|, \cdots, |P_\phi| \rangle$, where α and ϕ are the number of node types and deployment types respectively.

We define a *scaled setup* $C_t = \langle t, C \rangle$ as an instance of the cluster setup C running with topology t.

B.2 Incremental Scaling Algorithm

More formally, we define the ESC as:

$$
\begin{aligned}
&ESC = \langle n_{esc}, \theta_{esc} \rangle && n_{esc} = \mathcal{N}(\max_{\forall C}(\min_{t \in T_V^C}(s_t))) \cdot 2 \\
&\mathcal{N}(s_t) = \textstyle\sum_{\forall |N_i| \in s_t} |N_i| && \theta_{esc} = \max_{\forall C}(\Theta(\min_{t \in T_V^C}(s_t))) \cdot 2 \\
&\mathcal{P}(s_t) = \textstyle\sum_{\forall |P_i| \in s_t} |P_i| && \Theta(s_t) = \left\lceil \frac{\mathcal{P}(s_t)}{\mathcal{N}(s_t)} \right\rceil
\end{aligned}
$$

T_V^C is the set of topologies that have violations for setup C. When comparing the two scale s_r and s_q where $r, q \in T^C$, $s_r \geq s_q$ if $\mathcal{N}(s_r) > \mathcal{N}(s_q)$ or $\mathcal{N}(s_r) = \mathcal{N}(s_q) \wedge \mathcal{P}(s_r) \geq \mathcal{P}(s_q)$. The min and max functions are defined on this comparison. We double both n_{esc} and θ_{esc} to provide more confidence.

With the determined ESC, we design our scaling algorithm as shown in Algorithm 2. It starts from the smallest scale, and gradually increases the scale. In particular, for each node type i, it explores its scale from n_i^{min} to n_i^{max} where

$$n_i^{min} = \max(0, n_i^l) \quad n_i^{max} = \min(n_i^u, n_{esc})$$

[11] It can also include other workloads like StatefulSet, Job. We only discuss the Deployment for simplicity.

n_i^l and n_i^u are the lower and upper bound of the number of node type i defined by the users. After determining the number of nodes to explore for each scale, we determine the number of pods according to the total number of nodes $|N|$. In particular, for each pod type i, we explore the pod size from p_i^{min} to $FP_i(|N|)$ (a function of total number of nodes) where

$$p_i^{min} = \max(0, p_i^l) \quad FP_i(|N|) = \min(p_i^u, \theta_{esc} \cdot |N|)$$

where p_i^l and p_i^u are the lower and upper bound of the number of pod type i defined by the users . If there are multiple types of nodes and pods, it explores all combinations of sizes $\Pi_{i=1}^{\alpha}\{n_i^{min}, \cdots, n_i^{max}\} \times \Pi_{j=1}^{\phi}\{p_j^{min}, \cdots, FP_j(|N|)\}$. We skip any scale that result in trivial failure cases, meaning if that scale is not meaningful (i.e., $|N| = 0$ or $|P| = 0$) or generates non-interesting failures (i.e., when $|P| \gg |N|$, the excessive number of pods cannot be scheduled onto the nodes).

Algorithm 2 incremental scaling algorithm

```
1: procedure SCALING
2:     for s_t in sort(Π_{i=1}^α {n_i^min, ··· , n_i^max} × Π_{j=1}^φ {p_j^min, ··· , FP_j(|N|)}) do
3:         if not trivalCase((s_t, C)) then
4:             verifer((s_t, C))                ▷ verify for the scaled setup ⟨t, C⟩
5:         end if
6:     end for
7: end procedure=0
```

C Approximation in Kivi

We summarize a list of major approximations when implementing the model.

- We do not model the requests and we abstract the impact of these requests into resource changes (i.e., CPU resource usage change on pods). If users are interested in the properties of the requests (e.g., if traffic distribution is unbalanced), we currently cannot support these intent.
- We only model subsets of the features in Kubernetes as listed in Table 4. If a cluster includes unmodeled features, our model may not provide accurate results. Examples include pod graceful termination period, pod priority, pod disruption budgets, HPA stabilization windows, and StatefulSets.
- We do not model anything related to the container image or smaller elements than a pod, e.g., at the container level.
- We put one control loop into an atomic step and merge back-to-back control loops and events. Normally, one control loop or event can finish in a very short time. However, this can miss failures caused by such small transiting time windows.
- We do not model when controllers themselves encounter issues. Hence we omit error handling. If there are failures caused by the behavior of error handling, we cannot catch them.
- We approximate real numbers to two decimal places, which may lead to a loss of precision, though we have not yet seen any cases that need such precision.
- We implement the oscillation properties by examining if the loop has appeared on a small set of key relevant variables. As we do not check full system states, it is possible the loop we found does not cause permanent oscillation.