
KubeCon
CloudNativeCon
North America 2023

KubeCon | CloudNativeCon
North America 2023

# Don't Lose Your Sleep Over It: Verifying Your Kubernetes Clusters with Kivi


Bingzhe Liu and Gangmuk Lim

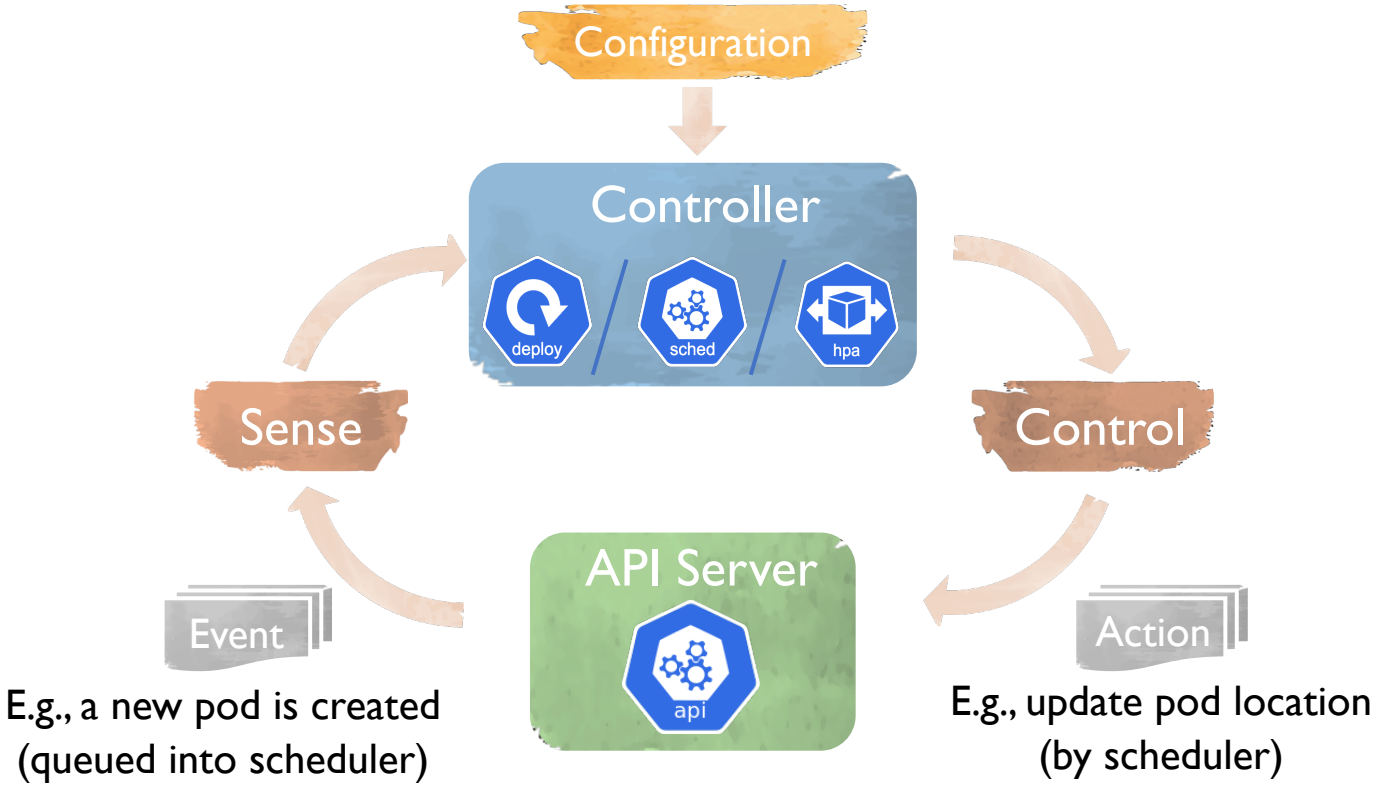
UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Collaboration with Ryan Beckett and Brighten Godfrey

# Kubernetes Consists of Dynamic Controllers



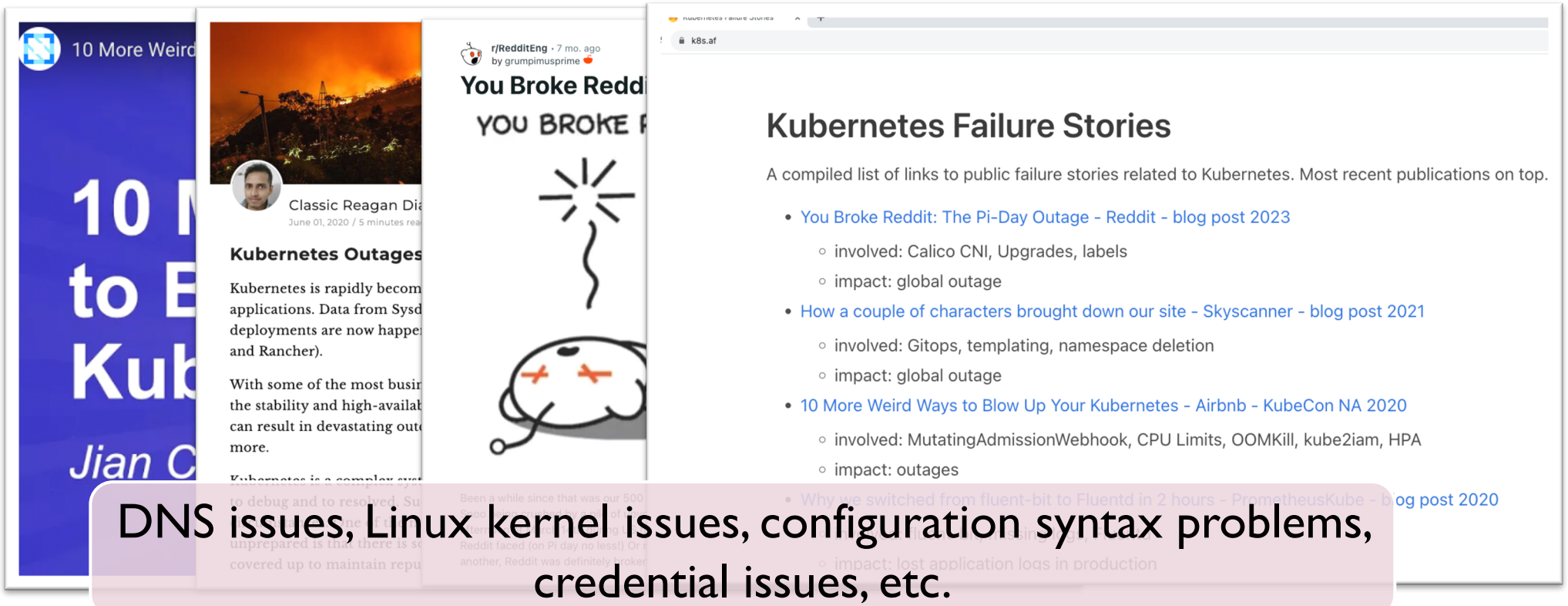
- Scheduler, HPA, deployment controller, etc.
- Making close-loop, dynamic decisions
- Each moves the system towards its desired state



# How Could Things Go Wrong?



We studied hundreds of failure reports collected by the community* + Github issues




DNS issues, Linux kernel issues, configuration syntax problems, credential issues, etc.



*Kubernetes Failure Stories: https://k8s.af/, maintained by Henning Jacobs

# How Could Things Go Wrong?



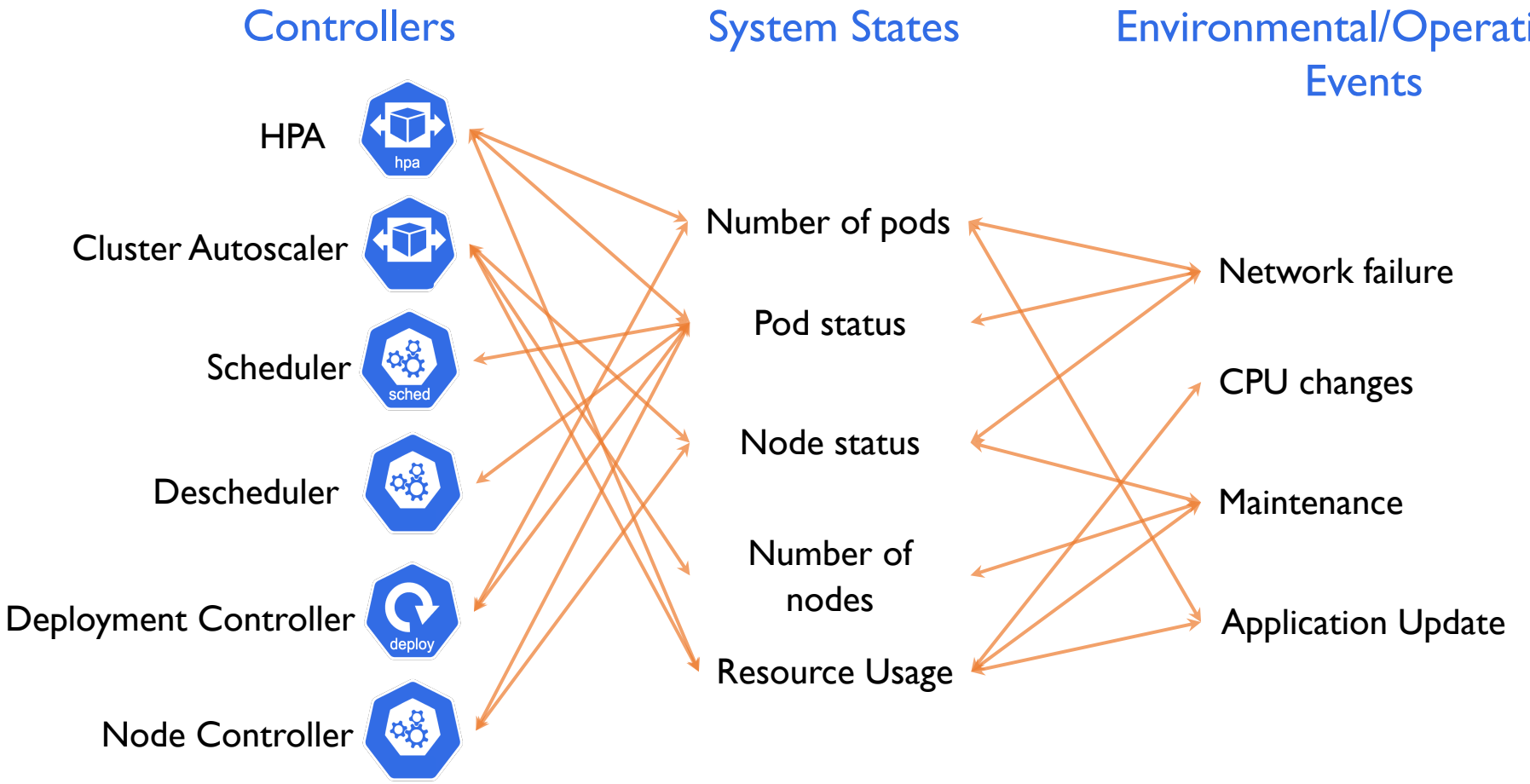
We identified a failure pattern that hasn't been discussed sufficiently

Non-trivial interactions between controller configurations or between controllers and events (e.g., node failure, maintenance)

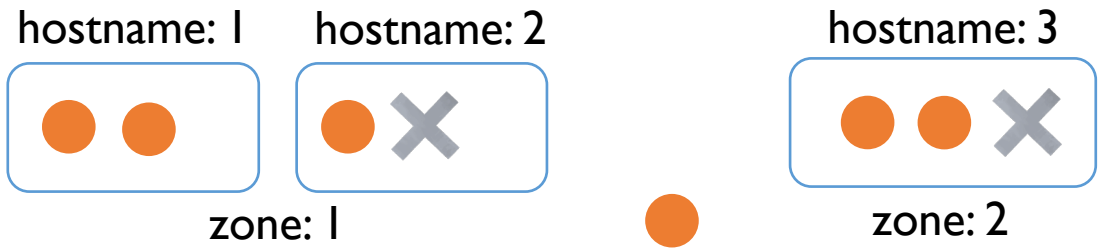
No global coordination, each control component has its own goals yet have shared dependency
Possibly configured by different teams that have different objectives

# Non-trivial Interactions Emerged from Controllers and Events





# Example 1:
## Conflicting Configurations for a Single Controller




Intent: pods stably scheduled

Need to modify the conflict constraints or change the node topology

Deployment Controller — The deployment should have 5 pods

Scheduler — `PodsTopologySpread`: Pods equally balanced across zones, with skew ≤ 1

Scheduler — `PodsTopologySpread`: Pods equally balanced across hostnames, with skew ≤ 1

Conflict, hard constraints

HPA — Metric: average CPU usage, MaxReplicas = 6

CPU increase — Average CPU usage increases (i.e., due to more traffic)



*Example modified from official documentation: https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/#example-conflicting-topologyspreadconstraints

# Example 2:
## Conflict Between Scheduler and Descheduler


Intent: pods stably scheduled ✗

The pod is in oscillation – unending cycle of scheduling and eviction

A descheduler bug we found + descheduler should not evict pods according to soft constraints

Deployment Controller: The deployment should have 6 pods.

Scheduler:
`PodsTopologySpread`: Pods equally balanced across zones, with skew ≤ 1
`PodsTopologySpread`: Pods equally balanced across hostnames, with skew ≤ 1
`NodeAffinity`: `weight(zone:1) = 4, weight(zone:2) = 5`

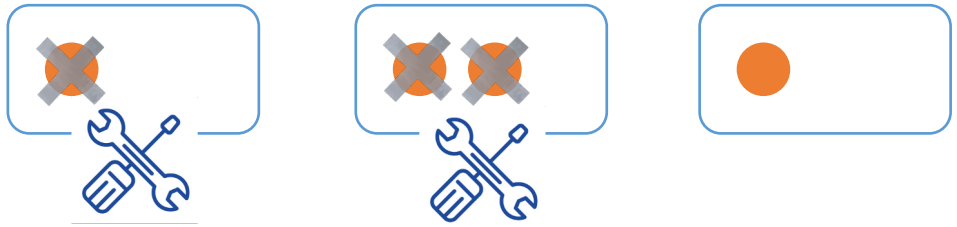
↺ Conflict, yet they are soft constraints

Descheduler: `RemovePodsViolatingTopologySpreadConstraint`



*Example simplified from a Github issue #921 in descheduler repo: https://github.com/kubernetes-sigs/descheduler/issues/921

# Example 3:
## Topology Unbalanced after Maintenance



Intent: #pods ≥ 2 ✗

A descheduler is needed to rebalance the pods

Deployment Controller — The deployment should have 3 pods.

Scheduler — `PodsTopologySpread`: Pods equally balanced across nodes, with skew ≤ 1

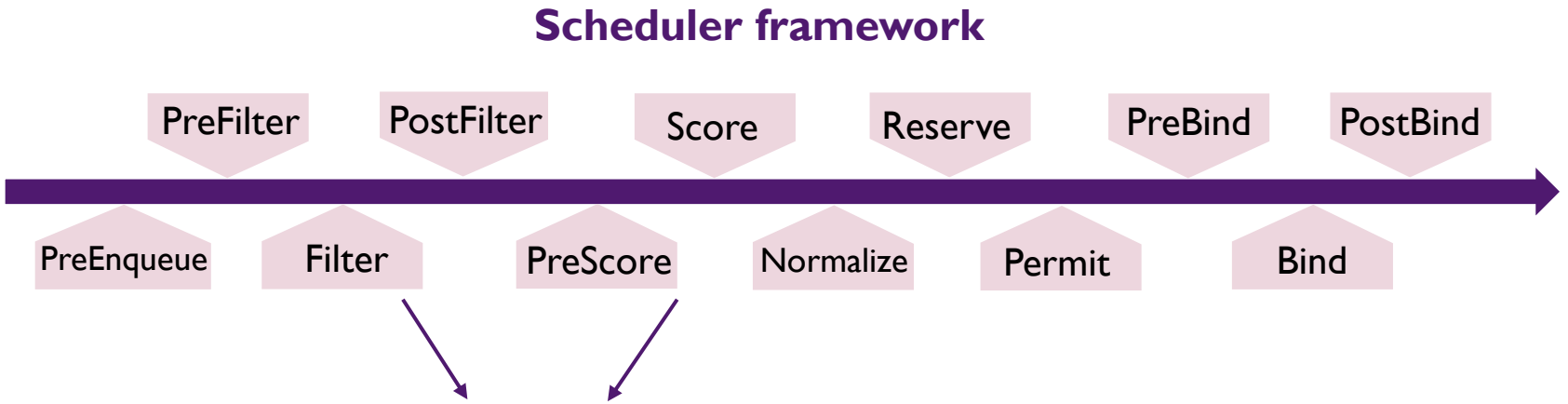
Maintenance — Take down one node at a time, update and put it back.



*Example from a talk "Moving the entire stack to K8s within a year – lessons learned". https://www.youtube.com/watch?v=tA8Sr3Nsx1I&t=1575s&ab_channel=CodeSpaceITeducation, 2018.

# Even Correctly Using a Single Controller is Hard






- 12 pipeline stages
- 21 default plugins

`PodTopologySpread` Plugins:

- Each constraint contains 8 parameters
- Many details, not well documented
  - If multiple constraints are defined, the node's labels need to match with all constraints to be considered in skewness calculation
- Interact with `NodeAffinity` and `NodeTaints` plugins
  - each constraint can choose to honor or not honor these plugins when calculating skewness



*graph reference https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

# More Issues

KubeCon CloudNativeCon North America 2023

More details in the paper

| Case ID | |
|---|---|
| C1 [21] | Pods leading |
| C2 [4, 6] | Not enoug without |
| C3 [26] | Configur |
| C4 [15] | Scheduler same node |
| C5 [24] | Conflict c |
| C6 [5] | Pod distri failures |
| C7 [16] | Conflict no |
| C8 [24] | Conflictin caused |

## Kivi: Verification for Cluster Management

Bingzhe Liu, UIUC
Gangmuk Lim, UIUC
Ryan Beckett, Microsoft
P. Brighten Godfrey, UIUC and VMware

**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.

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

| Case ID | Description | Properties |
|---|---|---|
| C9 [9] | Pods are being scheduled into the same node due to the image locality plugins outweighs all other scoring plugins | Unexpected Topology |
| | thy) and their average CPU er reduce the capacity. | Unexpected Object Numbers |
| | right taint, causing new g to be scheduled. | Unexpected Object Lifecycles |
| | nd the deployment of a new all the production pods. | Unexpected Object Lifecycles |
| | deletion by the deployment | Unexpected Object Lifecycles |
| | ile the ingress controller ance across nodes. | Unexpected Topology |
| | autoscaler to scale up nodes cale down the newly created s to deleted nodes. | Unexpected Object Lifecycles |
| | onents all woke up and nhealthy, at which time the thy nodes became healthy as nhealthy, and the cycle | Oscillation / Unexpected Object Lifecycle [13] |

# Kivi: Verification for Kubernetes Clusters

KubeCon CloudNativeCon North America 2023

- The first system for verifying correctness of controllers and their configurations in Kubernetes
  - Formal verification (model checking)

- Open-sourced software, research prototype
  - Github: https://github.com/bingzheliu/Kivi
  - Paper: https://arxiv.org/abs/2311.02800

Survey

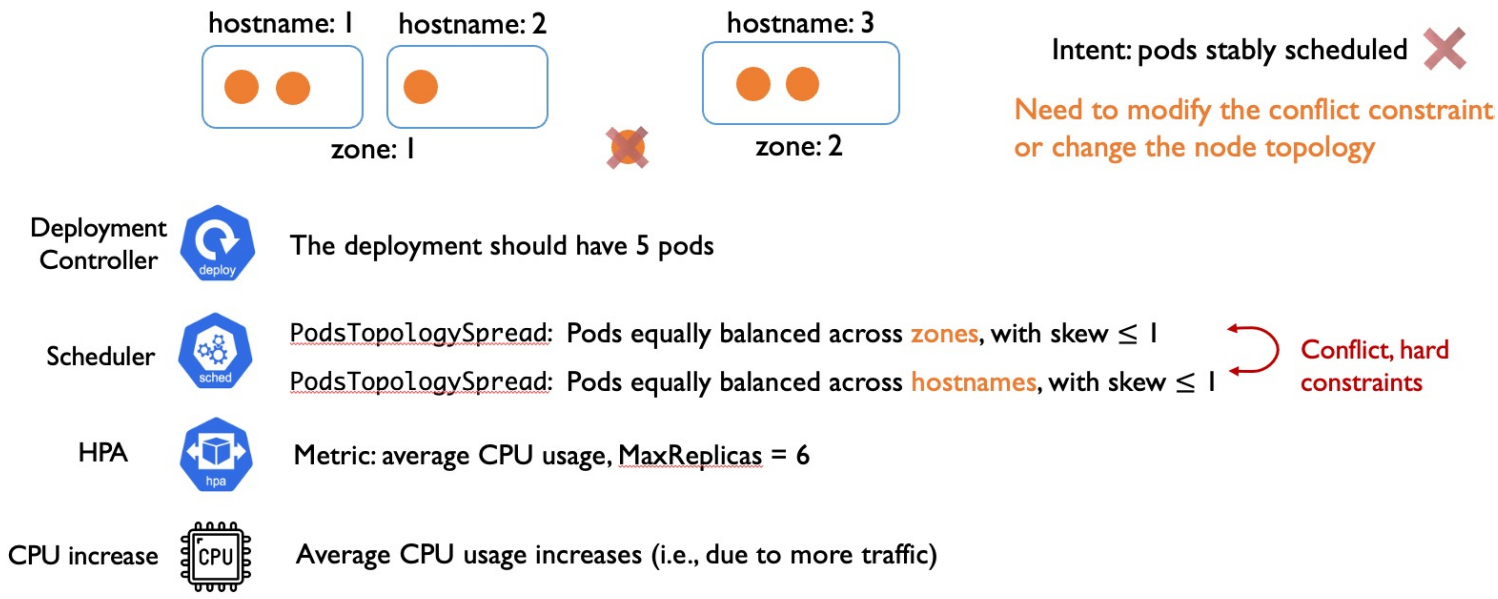
- Any early feedback is welcome! Please take our survey.
  - Survey link: https://forms.gle/AN2FKU5QbCdR3Jdv9
  - Contact: Bingzhe Liu bzheliu@gmail.com

# Demo

1. Simple Installation
2. Example 1

## Example 1:
### Conflict Configurations for a Single Controller

KubeCon CloudNativeCon North America 2023




15

*Example modified from official documentation: https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/#example-conflicting-topologyspreadconstraints

# What is Verification?

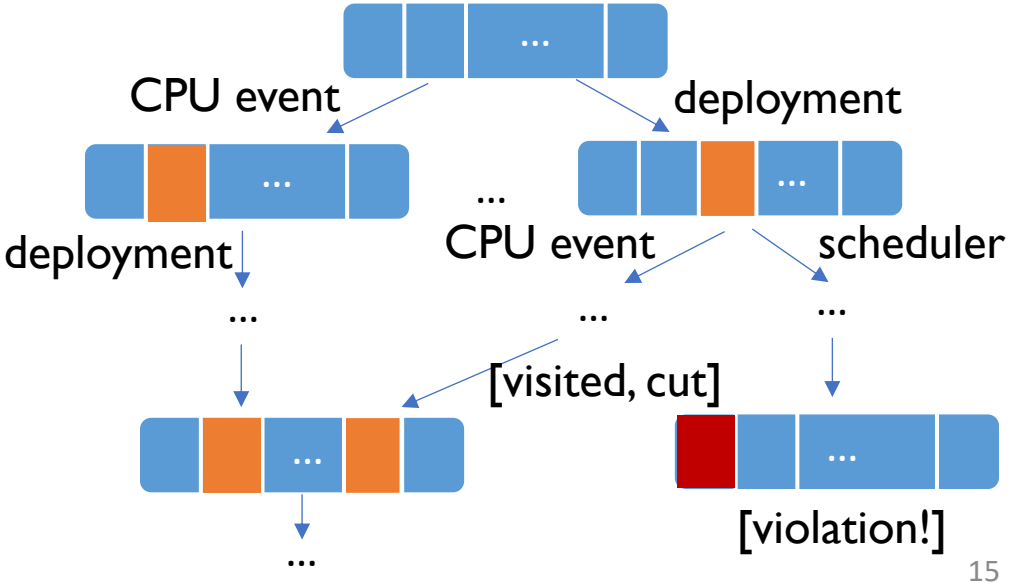


- Verification has been widely used in industry
  - cloud storage system (AWS S3), chip design, cluster networks, etc.

# What is Verification?



- Verification has been widely used in industry
- Formal verification can provide high coverage and a formal guarantee of correctness, via systematic, exhaustive exploration of a model.
- Model checking – we leverage SPIN
  - We translate system logic into a model
  - SPIN implements a depth-first search over all possible system states



# Why Verification?

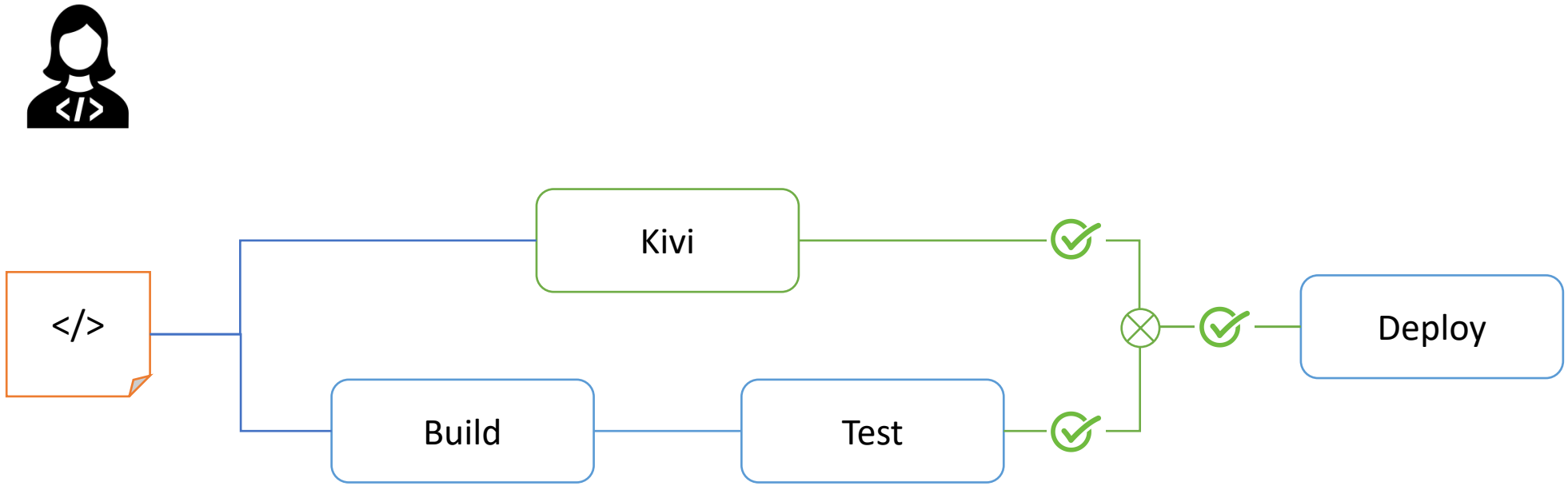


- Controllers are dynamic and interact in non-deterministic way
- Many problem only manifest
  - At a certain topology (e.g., example 1 and 2)
  - After non-deterministic events (e.g., CPU change)
- Comparing with testing and simulation/emulation
  - Testing/simulation are insufficient to handle such dynamic controller/event interactions in any non-deterministic order
  - Testing/simulation do not have full coverage of all possible scenarios, not exhaustive
  - Testing are mainly deployed for individual controller rather than the whole system

# Into CI/CD pipeline



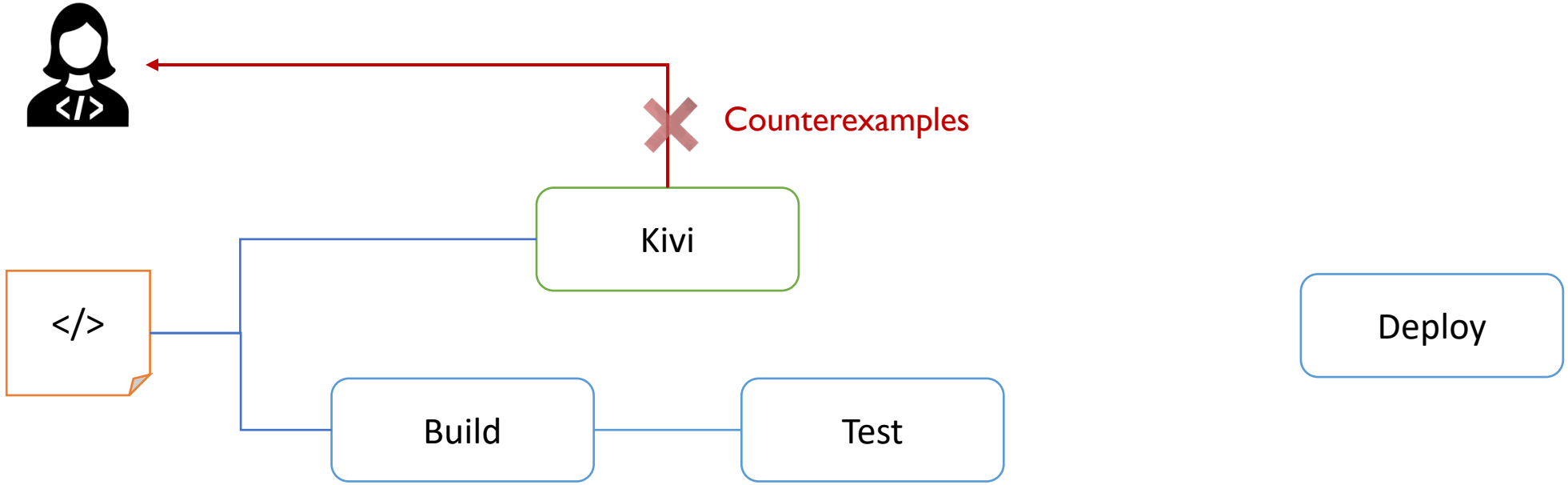
- Kivi can be integrated into part of the continuous integration workflow, sitting together with the testing tools.



# Into CI/CD pipeline



- Kivi can be integrated into part of the continuous integration workflow, sitting together with the testing tools.



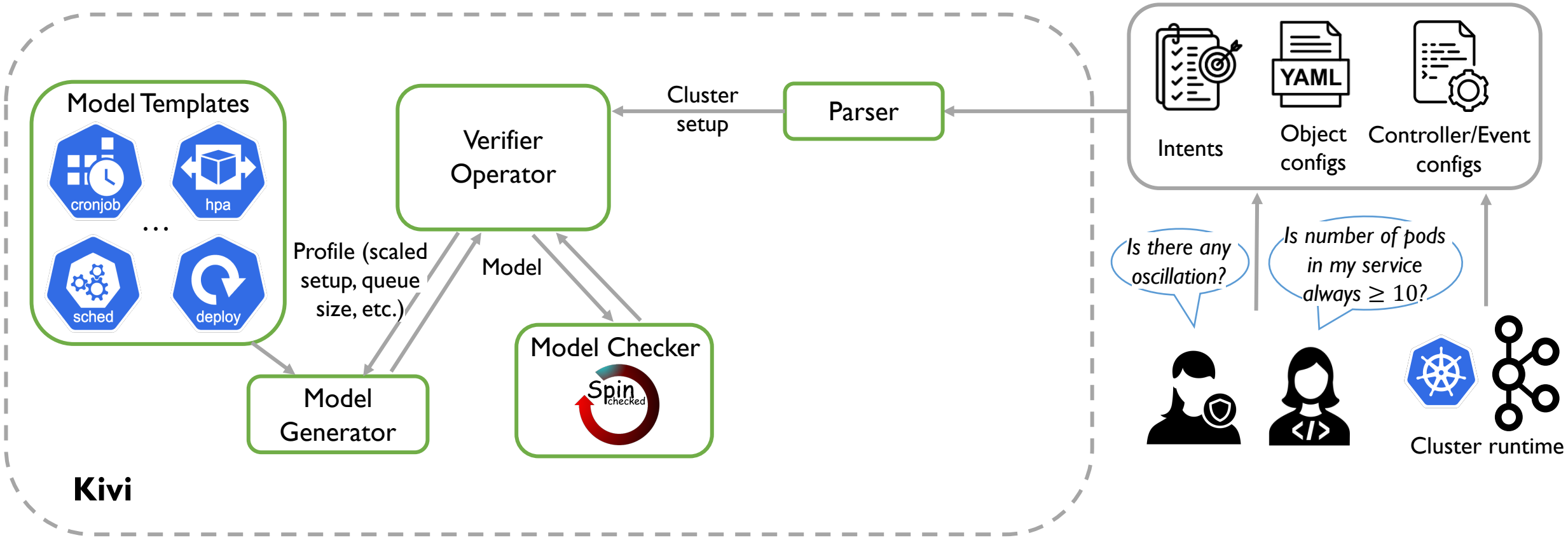
# How Could Users Use Kivi?



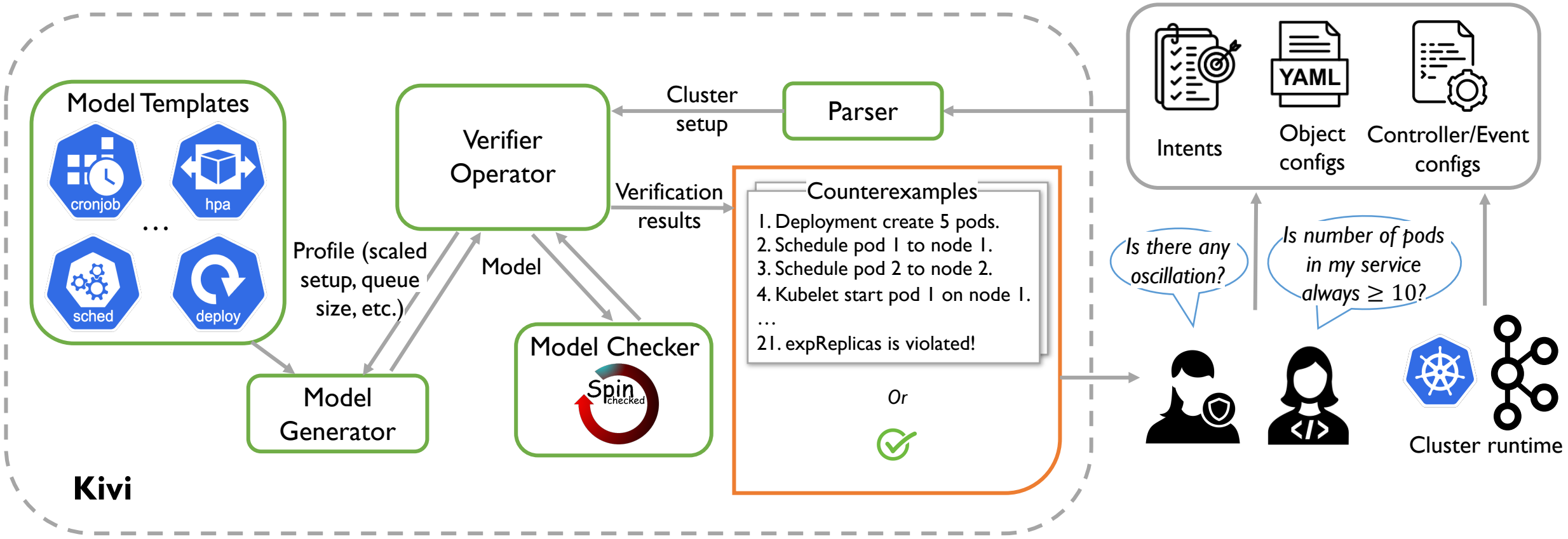
- Kubernetes End Users – Verify your cluster configurations
  - Deployment scheduling strategy, HPA configs, Descheduler configs, etc.

- Controller Developer – Understand how your controller interact with other parts of the system

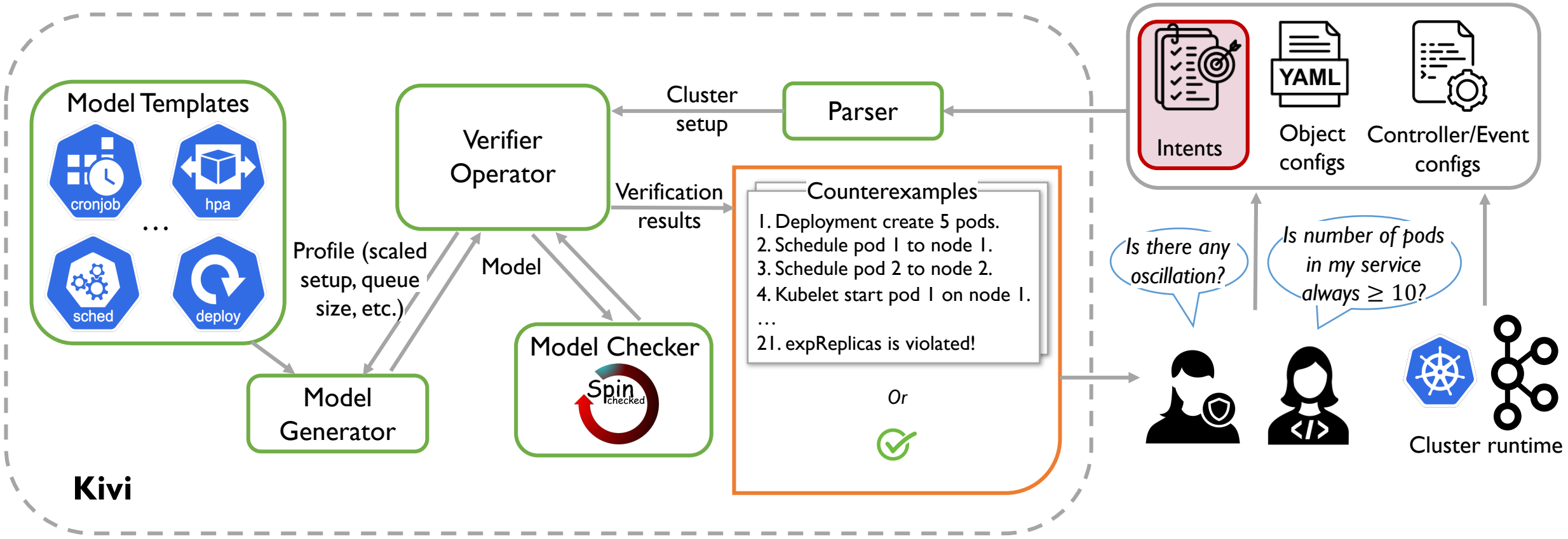
# Kivi System Workflow



# Kivi System Workflow



# Kivi System Workflow

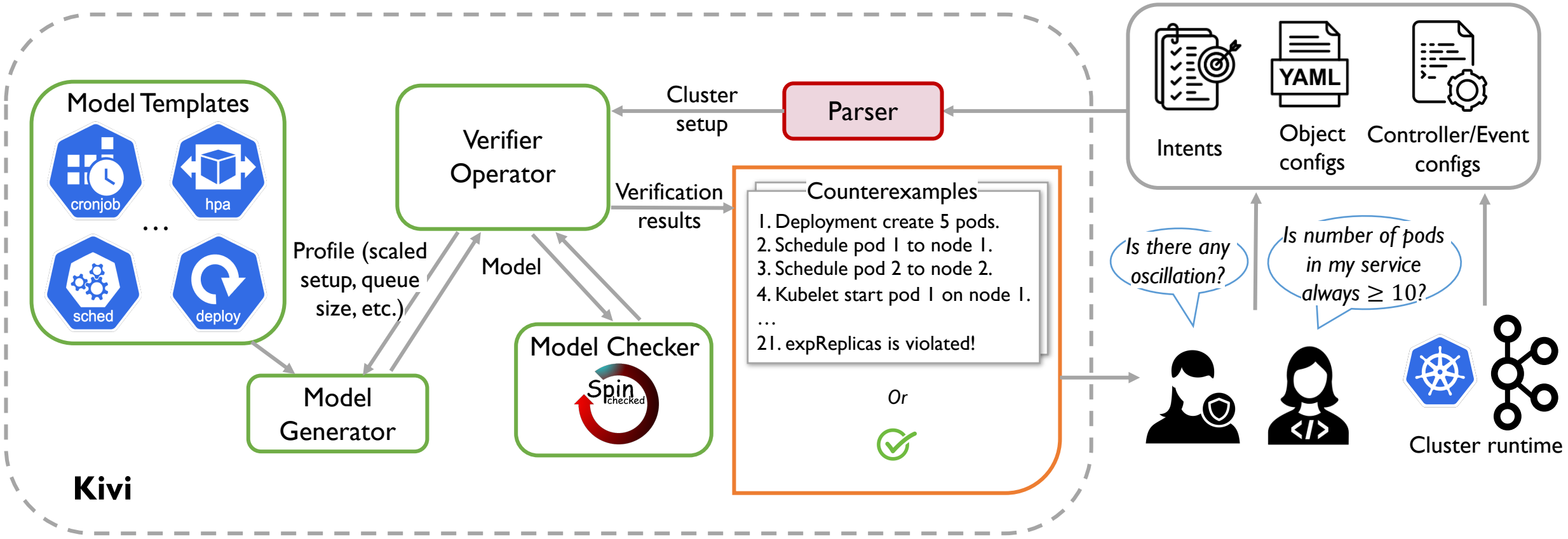


# Properties (Intents) to Verify for Kubernetes



- **Oscillation**
  Unstable cluster states. Change back and forth in an unending cycle. E.g., Example 2
- **Unexpected Topology**
  Objects should be placed in certain patterns. E.g., Example 3
- **Unexpected Object Numbers**
  Object numbers should fall into a certain range. E.g., Example 3
- **Unexpected Object Lifecycles**
  The creation, execution and termination of the objects should meet users' expectation. E.g., Example 1&2

# Kivi System Workflow



# The Parsed Uniform Format JSON

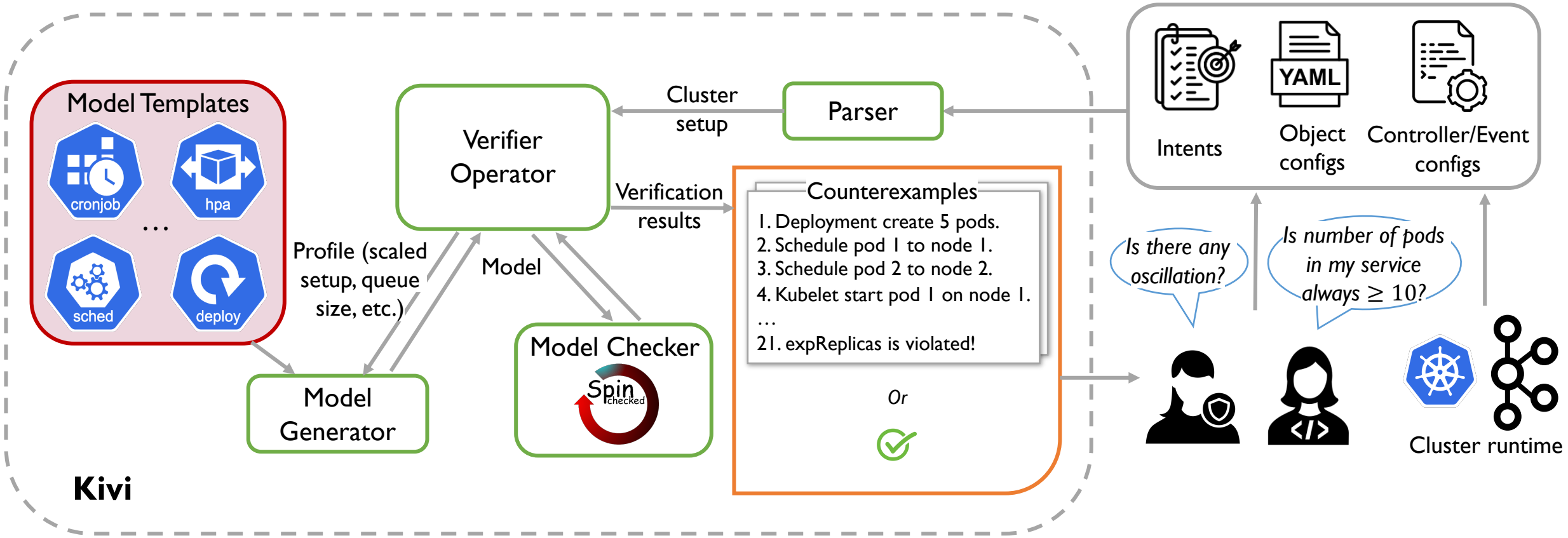


```
"setup": {
  "podTemplates": [
    {
      "labels": {
        "name": "app"
      },
      "cpuRequested": 8,
      "memRequested": 8,
      "topoSpreadConstraints": [
        {
          "maxSkew": 1,
          "topologyKey": "zone",
          "whenUnsatisfiable": 1,
          "numMatchedLabel": 1,
          "labels": {
            "name": "app"
          }
        }
      ]
    }
  ]
},
```

```
"userDefined": {
  "nodesTypes": [
    {
      "template": {
        "cpu": 64,
        "memory": 64,
        "cpuLeft": 64,
        "memLeft": 64,
        "status": 1,
        "labels": {
          "zone": 1
        },
        "numPod": 0,
        "name": 1
      },
      "lowerBound": 0,
      "upperBound": 10
    },
    {
      "template": {
        "cpu": 64,
        "memory": 64,
        "cpuLeft": 64,
        "memLeft": 64,
        "status": 1,
        "labels": {
          "zone": 2
        },
        "numPod": 0,
        "name": 2
      },
      "lowerBound": 0,
      "upperBound": 10
    }
  ],
```

```
"controllers": {
  "scheduler": {},
  "hpa": {},
  "deployment": {},
  "descheduler": {
    "profiles": [
      {
        "RemovePodsViolatingTopologySpreadConstraint": {}
      }
    ],
    "args": {
      "constraintsTopologySpread": 2
    }
  }
},
"events": [],
"intents": [
  {
    "name": "checkEvictionCycle",
    "para": {
      "did": 1
    }
  }
]
```

# Kivi System Workflow

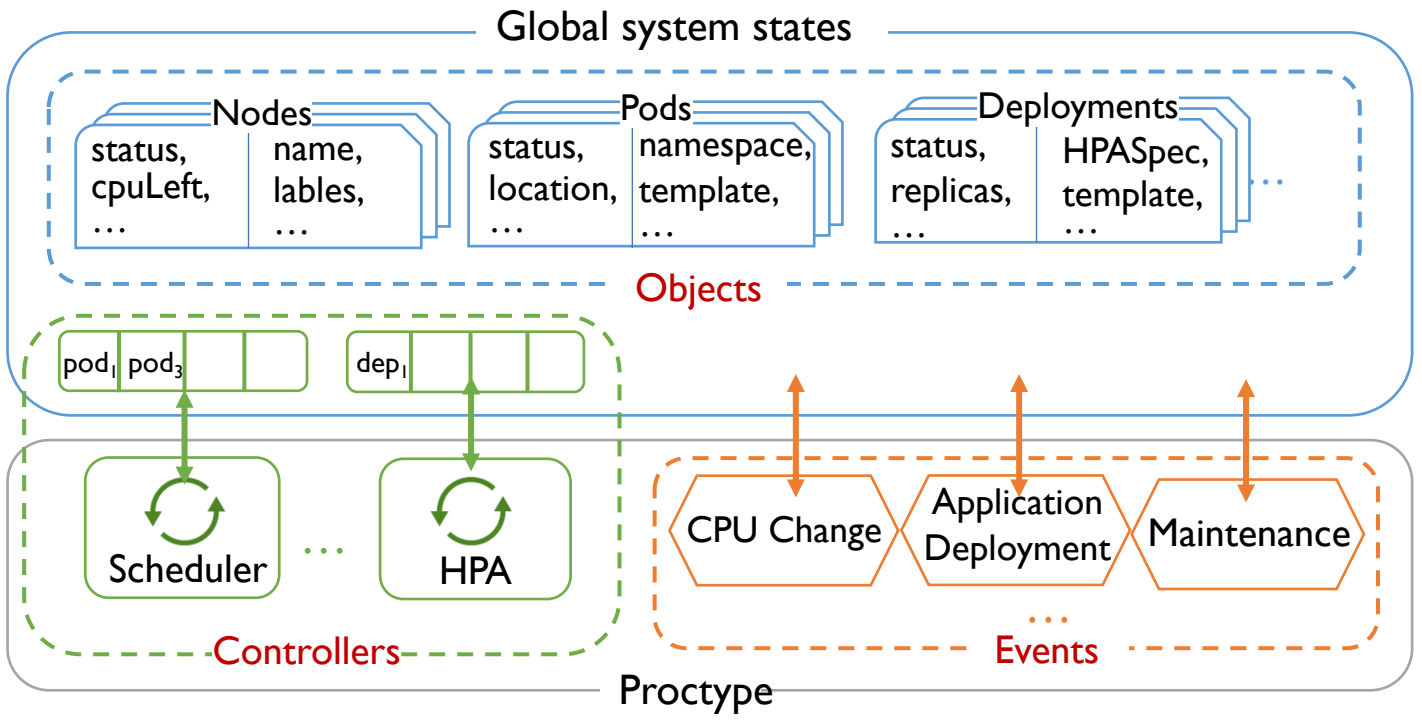


# Modeling



- Workload resources/Node
- Controllers
- Events

Using PROMELA language provided by SPIN:
A C-style language, easy to understand



# Modeling



- Workload resources/Node
- Controllers
- Events

Using PROMELA language provided by SPIN:
A C-style language, easy to understand

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

Defining node

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

Defining scheduler

# Implementation



- 6 most commonly used controllers and their features
- Manually examining the Kubernetes source code, and capture the most essential details
- Omitting implementation details like error handling, retries, API calls.

| Controllers | LoC | Events | LoC |
|---|---|---|---|
| Deployment | 199 | CPU Change | 86 |
| Scheduler | 783 | Kernel Panic | 25 |
| Descheduler | 471 | Node Failure | 6 |
| HPA | 222 | Apply/Create Deployment | 126 |
| Kubelet | 90 | Scale Deployment | 11 |
| Node Controller | 86 | Maintenance | 37 |

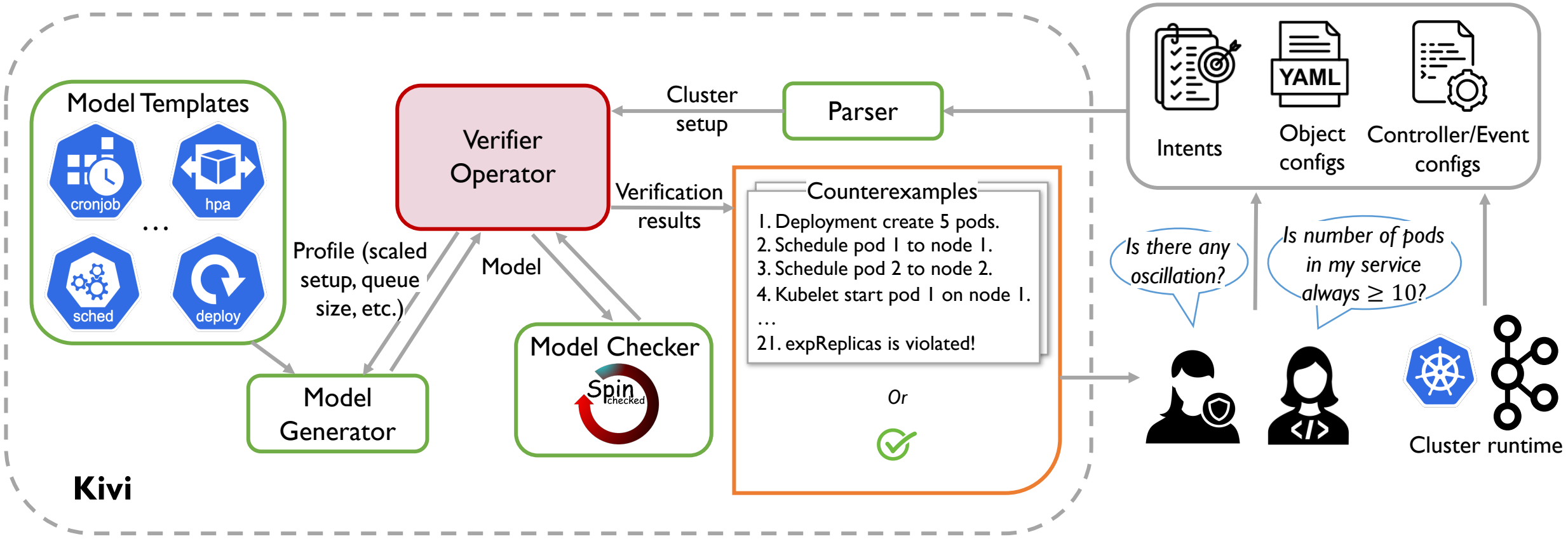
Kivi model can be used as a reference model for Kubernetes!

# Want to Support More Controllers?

KubeCon CloudNativeCon North America 2023

- The modeling language PROMELA is easy to use (C-style)
- Implement your own controller and put it into the pool of Model Template – the major logic is one-time effort
- Pull requests are welcome!

# Kivi System Workflow



# Scalability Challenges for Kubernetes



- A cluster can reach hundreds of nodes and many thousands of pods
- This can bring the known state explosion problem to verification

# Scalability Challenges for Kubernetes



- A cluster can reach hundreds of nodes and many thousands of pods

  This can bring the known state explosion problem to verification

- Autoscaling and elasticity is one of the key characteristic

  Users are interested in not only one but a wide range of cluster topologies

# Our Hypothesis:



**If a cluster can violate an intent, then it can do so at a relatively small scale.**

- The complexity mainly lies in the cluster controller logic and configuration that **does not grow with the scale**.
- All topologies are generated from one cluster configuration.
  A subset of the topologies are often representative for all the topologies.
- Similar observation in other types of distributed systems [Yuan et al.]*.

*Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. OSDI '14

# Our Hypothesis:



If a cluster can violate an intent, then it can do so at a relatively small scale.

# Our Approach: Incremental Scaling Algorithm

Starts to verify a cluster at the smallest possible scale, and increases the scale until finding violations or reaching a ‘confident size’

# Empirical Study – Small Scale is Enough



Examine a collection of real-world failure cases

| Case ID | Description | Properties |
|---|---|---|
| C1 | Pods consumed high CPU during bootstrapping leading HPA to scale up rapidly to max replicas. | Unexpected object numbers |
| C2 | Not enough replicas because users apply an updated YAML file without defining number of replicas (1 by default). | Unexpected object numbers |
| C3 (Example 1) | Configurations of two `PodTopologySpread` constraints caused the 6th pod to fail to be scheduled. | Unexpected object lifecycles |
| C4 | Scheduler kept assigning pods with high CPU usage to the same node causing a kernel panic and pod failure loop. | Oscillation / Unexpected object lifecycles |
| C5 | Conflict configurations of scheduler and `RemoveDuplicate` policy in descheduler. | Oscillation |
| C6 (Example 3) | Pod distribution unbalanced after maintenance. Node failures then caused the pod count to drop too low. | Unexpected object placement / numbers |
| C7 | Conflicting configurations of node taint and pod `NodeName` caused scheduling and eviction loop | Oscillation / Unexpected object lifecycles |
| C8 (Example 2) | Conflicting descheduler and scheduler configurations caused scheduling and eviction loop. | Oscillation / Unexpected object lifecycles |

# Empirical Study – Small Scale is Enough



| Case ID | Min Violation Scale <#nodes, #pods> |
|---|---|
| C1 | <1, 3> |
| C2 | <1, 2> |
| C3 | <3, 6> |
| C4 | <2, 5> |
| C5 | <2, 4> |
| C6 | <2, 2> |
| C7 | <1, 1> |
| C8 | <3, 6> |

The maximum min violation scale (across all cases) is **3 nodes** and **6 pods**.

# Empirical Study – Small Scale is Enough



| Case ID | Min Violation Scale <#nodes, #pods> | Reported Scale <#nodes, #pods> |
|---|---|---|
| C1 | <1, 3> | <Unknown, 150+> |
| C2 | <1, 2> | <Unknown, 3> |
| C3 | <3, 6> | <3, 6> |
| C4 | <2, 5> | Unknown |
| C5 | <2, 4> | <5, 10> |
| C6 | <2, 2> | <3, 3> |
| C7 | <1, 1> | <Unknown, 5> |
| C8 | <3, 6> | <5, 10+> |

The maximum min violation scale (across all cases) is **3 nodes** and **6 pods**.

# Empirical Study – Small Scale is Enough



| Case ID | Min Violation Scale <#nodes, #pods> | Reported Scale <#nodes, #pods> |
|---|---|---|
| C1 | <1, 3> | <Unknown, 150+> |
| C2 | <1, 2> | <Unknown, 3> |
| C3 | <3, 6> | <3, 6> |
| C4 | <2, 5> | Unknown |
| C5 | <2, 4> | <5, 10> |
| C6 | <2, 2> | <3, 3> |
| C7 | <1, 1> | <Unknown, 5> |
| C8 | <3, 6> | <5, 10+> |

The maximum min violation scale (across all cases) is **3 nodes** and **6 pods**.
In **6 of 7** cases, min violation scale is smaller than reported scale.

# Empirical Study – Small Scale is Enough



| Case ID | Min Violation Scale <#nodes, #pods> | Reported Scale <#nodes, #pods> | Reproduced? |
|---|---|---|---|
| C1 | <1, 3> | <Unknown, 150+> | ✓ |
| C2 | <1, 2> | <Unknown, 3> | ✓ |
| C3 | <3, 6> | <3, 6> | |
| C4 | <2, 5> | Unknown | |
| C5 | <2, 4> | <5, 10> | ✓ |
| C6 | <2, 2> | <3, 3> | ✓ |
| C7 | <1, 1> | <Unknown, 5> | ✓ |
| C8 | <3, 6> | <5, 10+> | ✓ |

The maximum min violation scale (across all cases) is **3 nodes** and **6 pods**.

In **6 of 7** cases, min violation scale is smaller than reported scale.

We reproduced the smaller scale in real Kubernetes clusters

# How to Determine the "Confident Size"?



| Case ID | Min Violation Scale <#nodes, #pods> | Reported Scale <#nodes, #pods> | Reproduced? |
|---|---|---|---|
| C1 | <1, 3> | <Unknown, 150+> | ✓ |
| C2 | <1, 2> | <Unknown, 3> | ✓ |
| C3 | <3, 6> | <3, 6> | |
| C4 | <2, 5> | Unknown | |
| C5 | <2, 4> | <5, 10> | ✓ |
| C6 | <2, 2> | <3, 3> | ✓ |
| C7 | <1, 1> | <Unknown, 5> | ✓ |
| C8 | <3, 6> | <5, 10+> | ✓ |

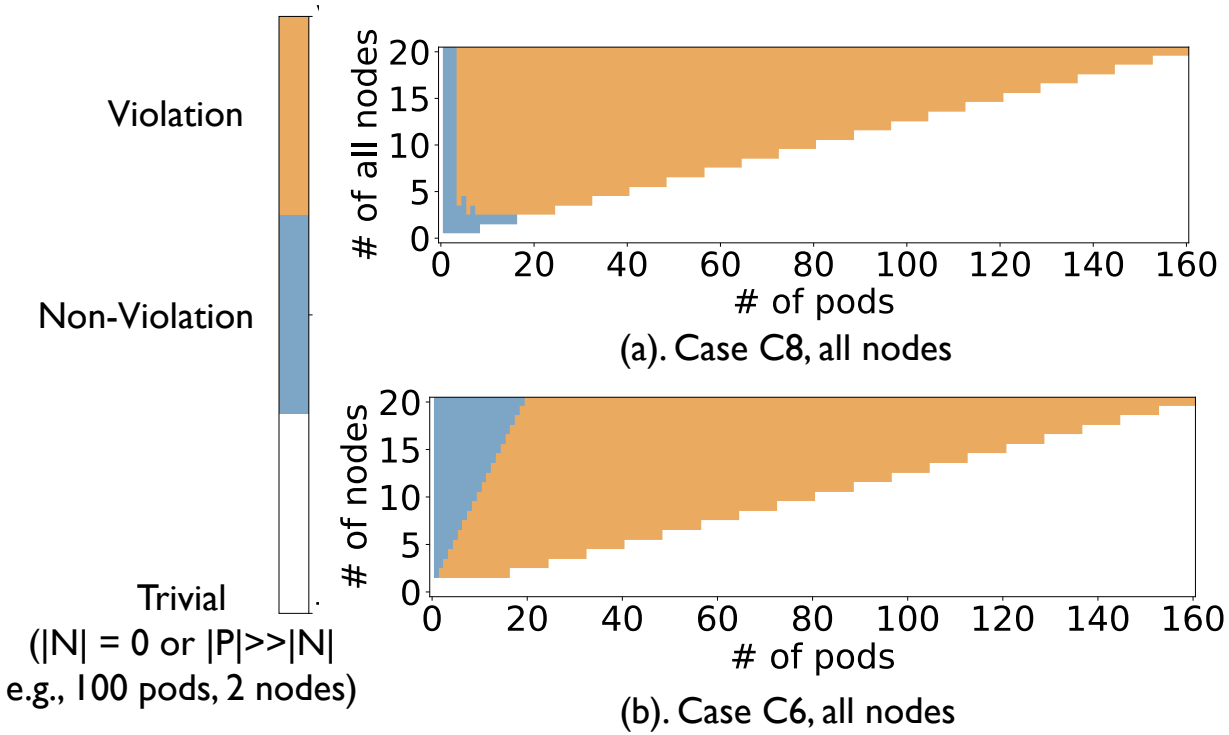
The maximum min violation scale (across all cases) is **3 nodes** and **6 pods**.

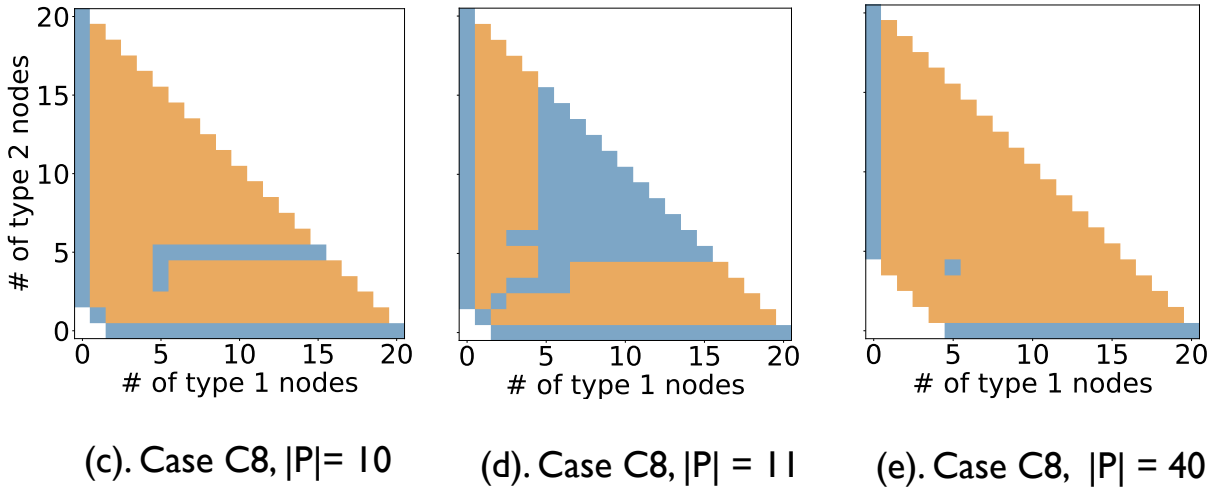
Chooses the confident size *empirically* by setting to 2x (double for more confidence) max min violation scale

In this case, confidence size is 6 nodes

# Need to Check All Combinations of Sizes Up to the Confident Size






Violations consistently appear for sufficiently large |N| and |P|.
The exact combinations of $|N_i|$ and |P| matters.

# Summary of the Incremental Scaling Algorithm



Chooses the confident size empirically from the past failures/failure database.

Starts from the smallest non-trivial scale (i.e., |N| = 0 or |P|>>|N|), check all combination of sizes for all types, until finding violations or reaching the confidence size.

If no violation found, Kivi concludes that there is no violation with high confidence

# Modeling Optimization for Scalability



- The size of the global states
  - Clearly defining small sets of mutating global variables: divide into two sections, a *stable* and a *mutating* section
- Concurrency
  - Merge multiple controller instances into one
  - Merge each control loop into an atomic block
- Search depth
  - Event-driven for all controllers to avoid unnecessary search
  - Merge back-to-back execution
  - Use partial key variables to track for loop properties

# Demo
# with Incremental Scaling Algorithm

# Example 2:
## Conflict Between Scheduler and Descheduler


Intent: pods stably scheduled

The pod is in oscillation – unending cycle of scheduling and eviction

A descheduler bug we found + descheduler should not evict pods according to soft constraints

Deployment Controller

The deployment should have 6 pods.

Scheduler

`PodsTopologySpread`: Pods equally balanced across zones, with skew ≤ 1
`PodsTopologySpread`: Pods equally balanced across hostnames, with skew ≤ 1
`NodeAffinity`: `weight(zone:1) = 4, weight(zone:2) = 5`

Conflict, yet they are soft constraints

Descheduler

`RemovePodsViolatingTopologySpreadConstraint`



*Example simplified from a Github issue #921 in descheduler repo: https://github.com/kubernetes-sigs/descheduler/issues/921

# Evaluation Result Summary



1. Can we use realistic failures to validate our hypothesis and scaling approach? - Yes!

2. What is the performance and scalability of Kivi? - Kivi performs well for large deployment
   - Finish most realistic cases within **100s**, and all cases within 25mins.

3. How accurate is Kivi? - Kivi can closely model the real system
   - Successfully found the correct violation for all violation cases; report no failure for all non-violation cases.
   - Compared with real K8s logs, most cases match **close to 100%**.

4. Can Kivi find new problems in Kubernetes? - Yes!
   - Found **two issues** for a Kubernetes controller.

# Two New Issues Found in Descheduler



- `RemoveTopologySpreadConstraint` does not consider all constraints and can mistakenly evict pods

- `RemoveDuplicates` does not respect node resources and can mistakenly evict pods.

# Acknowledgement

Thanks everyone for helping with our proposal and giving feedback for this project/presentation!

Venkat Arun, Madhu C.S, Yongli Chen, Timothy Goodwin, Jian Heung, Amine Hilaly, Rob Kooper, Yao Li, Timothy Andrew Manning, Haoran Qiu, Xudong Sun, Huang-Wei

Everyone in the Networking Group@UIUC

# Kivi: Verification for Kubernetes Clusters

KubeCon CloudNativeCon North America 2023

- The first system for verifying correctness of controllers and their configurations in Kubernetes

- Open-sourced software, research prototype
  - Github: https://github.com/bingzheliu/Kivi
  - Paper: https://arxiv.org/abs/2311.02800

- Any early feedback is welcome! Chat with us!

  We are seeking for early users!
  - Please take our survey: https://forms.gle/AN2FKU5QbCdR3Jdv9
  - Contact: Bingzhe Liu bzheliu@gmail.com

Survey


KubeCon
CloudNativeCon
North America 2023

# Scalability Challenges for Kubernetes

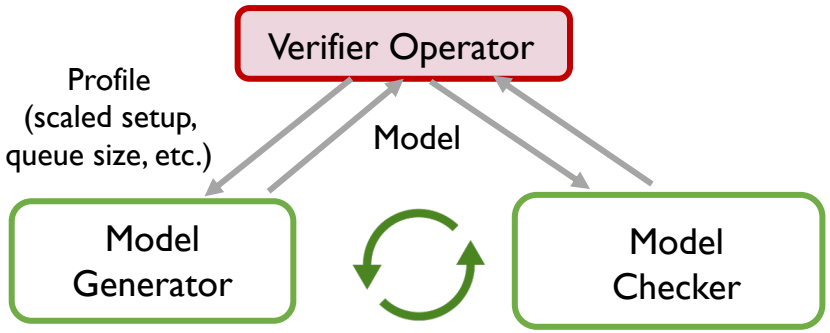


- A cluster can reach hundreds of nodes and many thousands of pods

  This can bring the known state explosion problem to verification

- Autoscaling and elasticity is one of the key characteristic

  Users are interested in not only one but a wide range of cluster topologies




Verifier operator operates verification in multiple cycles

Can run too many cycles if possible topology number is too big.

# How to Determine the "Confident Size"?



- Find it *empirically* by leveraging a library of known failure cases

1. Find the minimum scale that violations are found for each case

2. Define Confident Size as

$$Conf = < n_{conf}, \theta_{conf} >$$

$n_{conf}$ is the maximum nodes number among Min Violation Scale.

$\theta_{conf}$ is the maximum pods to nodes ratio.

3. Double $n_{conf}$ and $\theta_{conf}$ for more confidence.

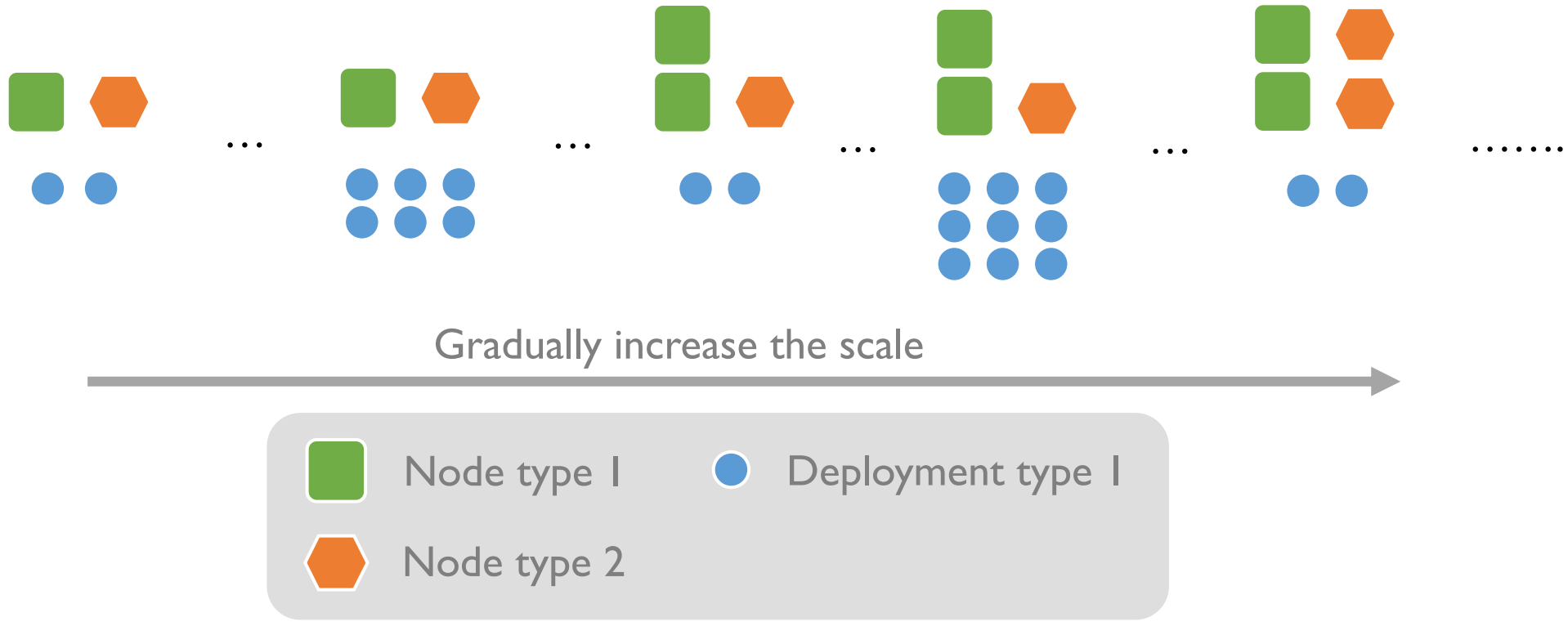
$n_{conf}$ = 6, $\theta_{conf}$ = 6

| Case ID | Min Violation Scale <#nodes, #pods> |
|---|---|
| C1 | <1, 3> |
| C2 | <1, 2> |
| C3 | <3, 6> |
| C4 | <2, 5> |
| C5 | <2, 4> |
| C6 | <2, 2> |
| C7 | <1, 1> |
| C8 | <3, 6> |

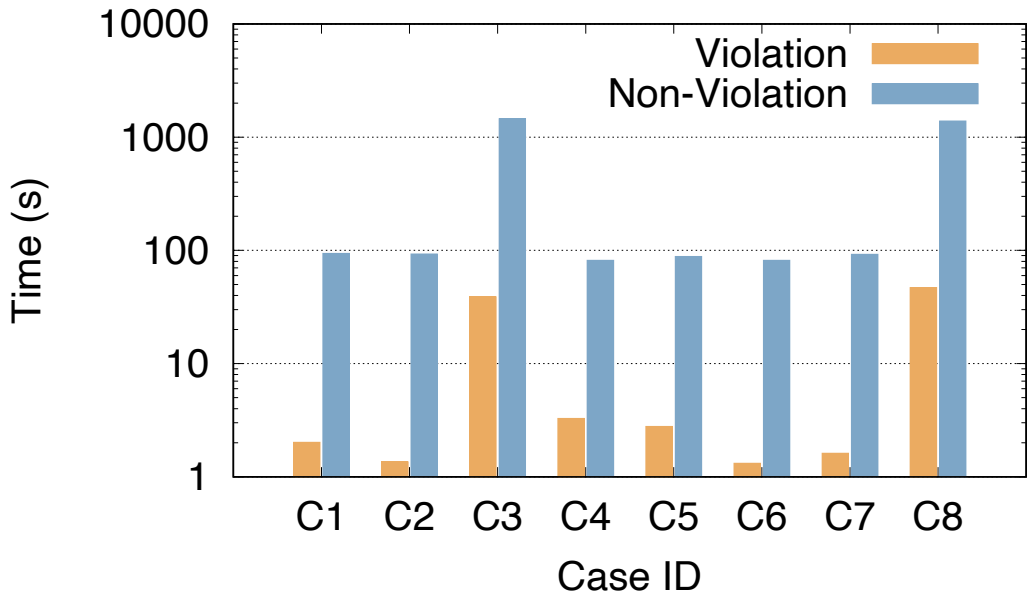
$^3/_1 = 3$

$\lceil ^5/_2 \rceil = 3$

# Incremental Scaling Algorithm

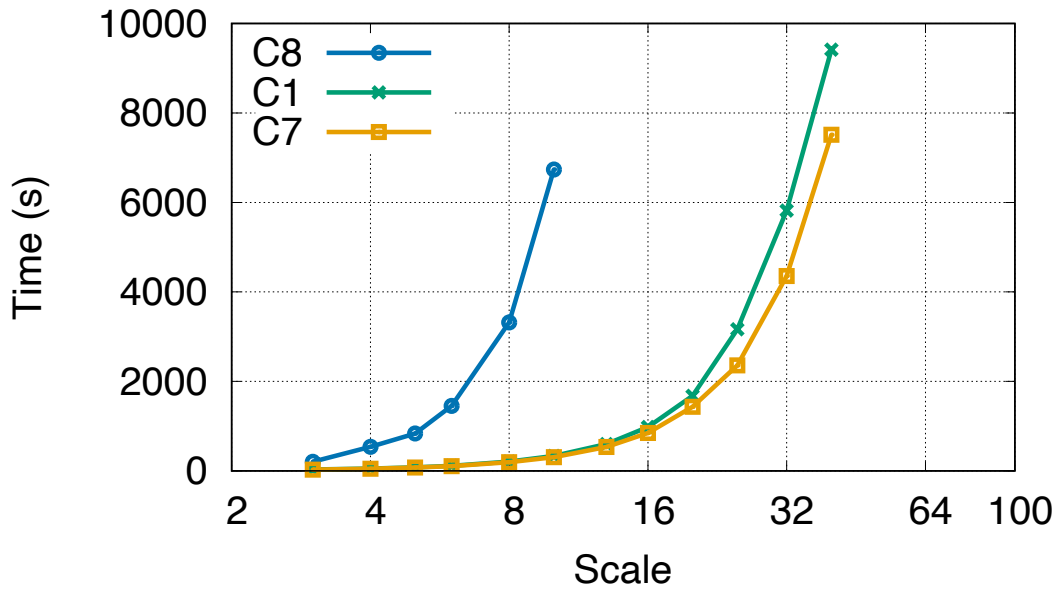


# What is the Performance and Scalability of Kivi?




Run time for Kivi, most cases finished within 100s

Run time is proportional to the number of setup that need to be tested, e.g., C3 and C8 have 1764 setup

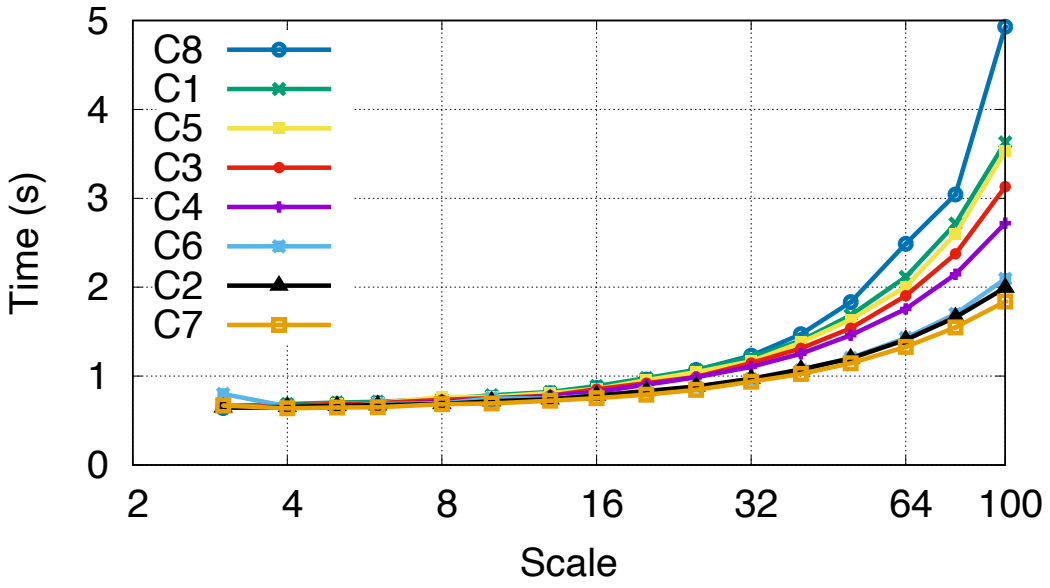



Run time without scaling algorithm

Times out ($> 10000\ sec$) even at moderate scale ($\leq 50$ nodes)

# Internal Test on Model Performance






Our model performs well even at large scale

Note: this shows performance for a single run of SPIN rather than checking the whole range of topologies

# How accurate is Kivi?



- Found the correct violation for all the violation cases
  Report no failures for all non-violation cases
- Compared with real Kubernetes logs

| | Cases w/ Non-deterministic Events | | | Cases w/o Non-deterministic Events |
|---|---|---|---|---|
| Case Id | C1 | C2 | C6 | C3, C5, C7, C8 |
| Matching rate | 81.6% | 97.8% | 100% | 100% |

$$matching\ rate = \frac{matched\ k8s\ actions + matched\ verifier\ actions}{all\ actions}$$

Kivi can closely models the real system

# Discussion and Future Work



## Limitation in scalability

May not scale to clusters with a large degree of node and deployment types

- Empirically, we need to verify only a small number of types
- Future work for optimization
  - Divide and conquer
  - Partial order reduction on symmetric objects
  - Multi-core
  - Faster ramp up in scaling algorithm

## Approximation

- More engineering efforts are needed to bring Kivi closer to real implementation
- Exists fundamental gap because of not verifying the code; testing are complementary.