KubeCon CloudNativeCon
North America 2026

Saturated by Design

Dakota Sullivan and Isaac Wilson





Dakota Sullivan
The Trade Desk
dqsully



Isaac Wilson
Elastic
jukie

Why go on-prem?

Cons

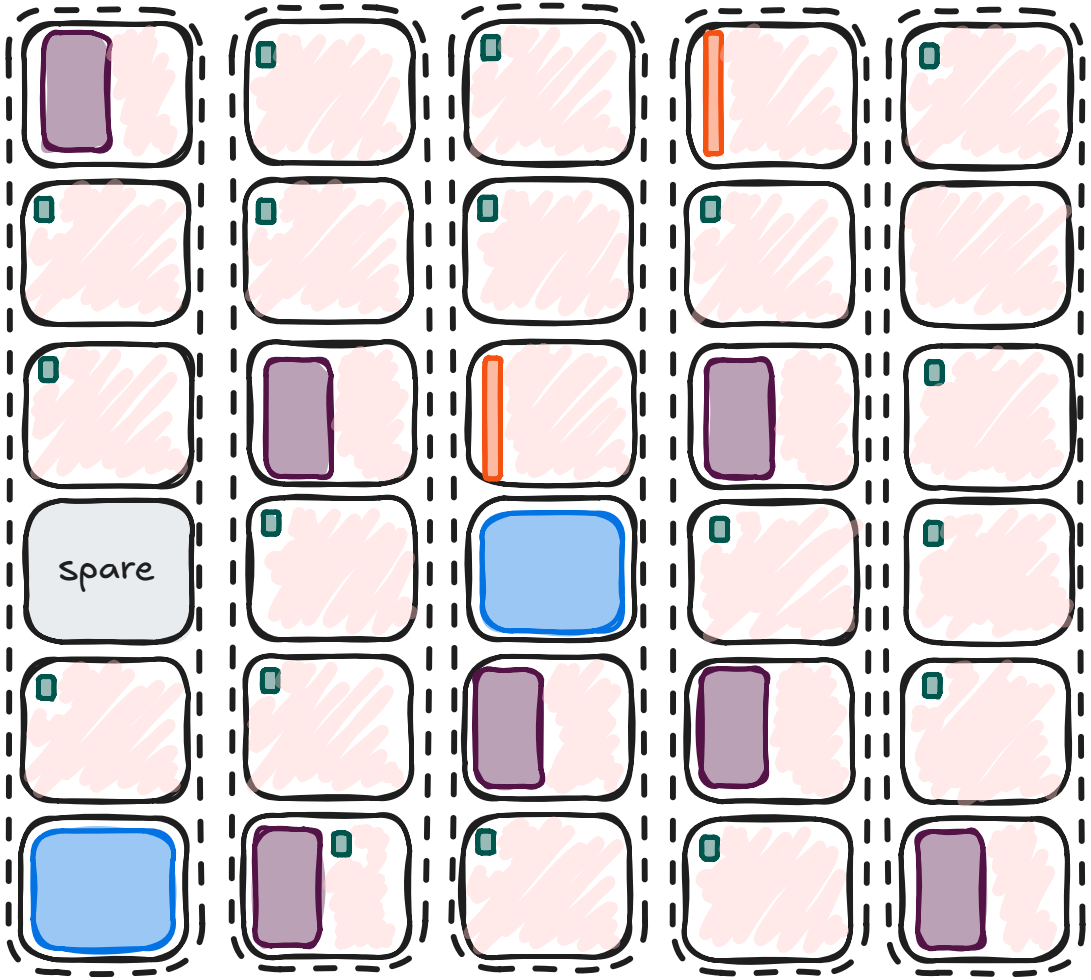
- Requires major upfront investment and planning
- No managed services
- You're responsible for fixing any issues
- Scaling is hard
- Capacity is more expensive than real usage

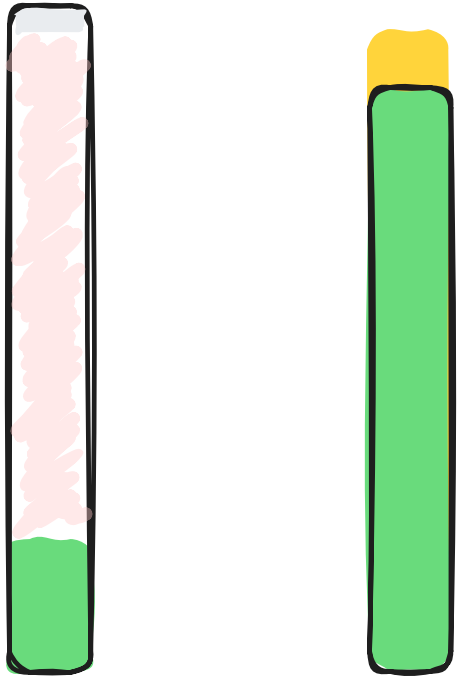
Pros

- It's a heck of a lot cheaper
- You can customize your hardware
- (And data governance can be easier, too)

At The Trade Desk

- About 10 on-prem datacenters
- In most, a Kubernetes cluster between 1,500-3,000 nodes
- 4 main workloads:
  - A - whole node, 25% of DC capacity
  - B - 1/2 of a node, 55% of DC capacity
  - C - 1/5th of a node, 5% of DC capacity
  - D - 1/20th of a node, 15% of DC capacity
- Non-trivial pod constraints





20% utilized

10% over-scale

Problems

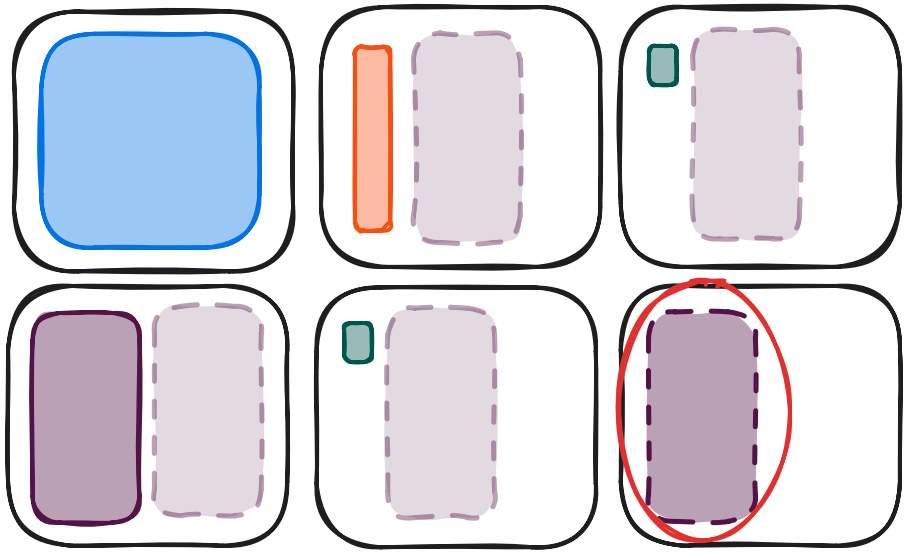
1. Node scoring
2. Deployment over-scaling
3. Fragmentation
4. Right-sizing

1. Node scoring

Which nodes Kubernetes prioritizes for new pods.

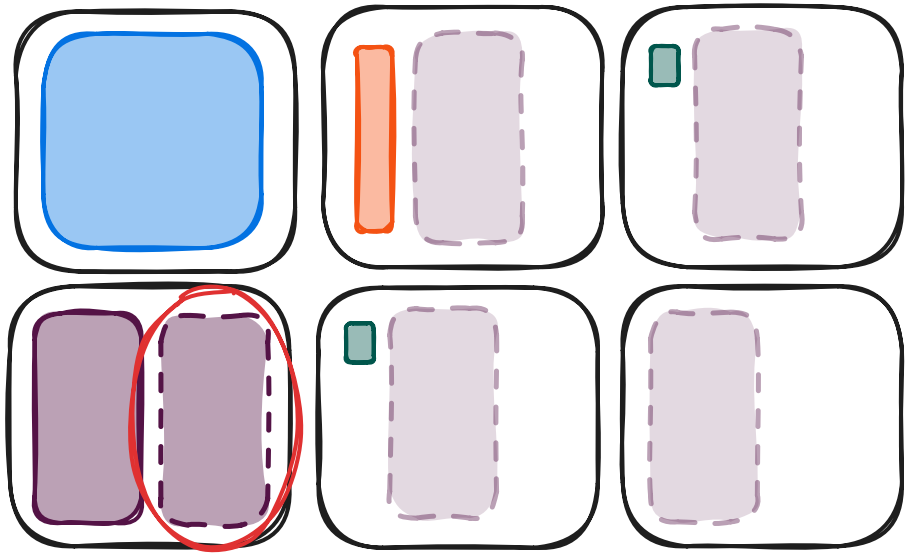
LeastAllocated scoring (default)

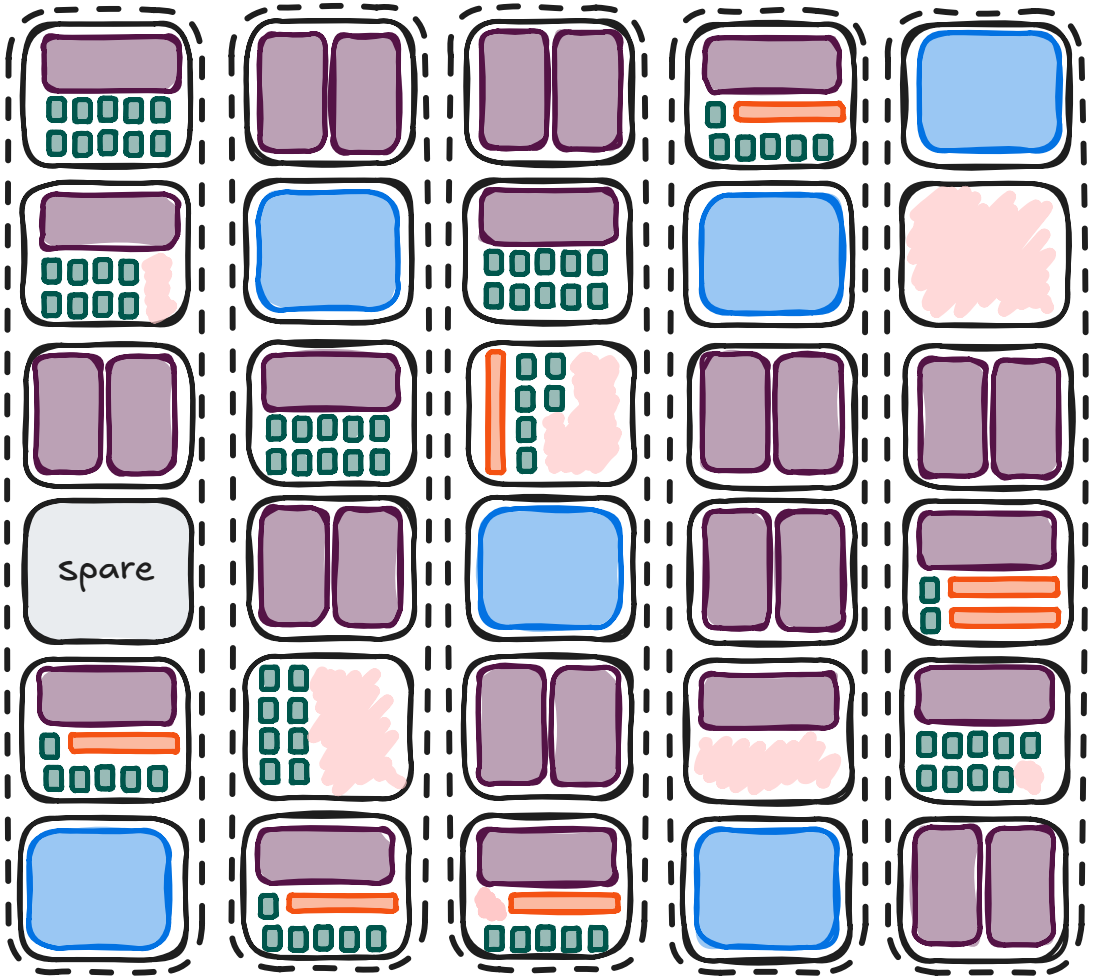
1. NODE SCORING

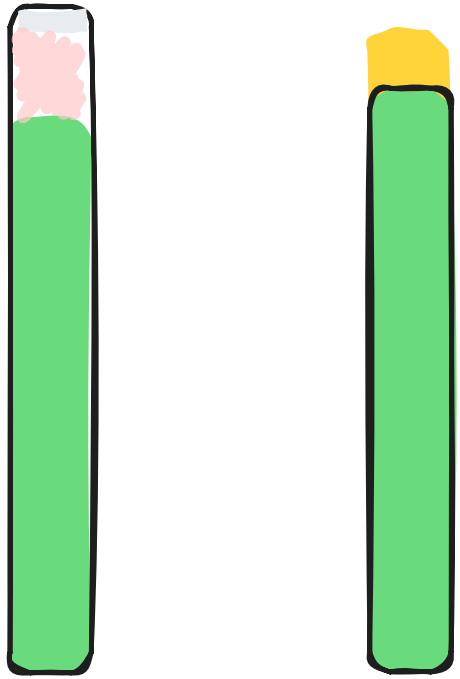


MostAllocated scoring

1. NODE SCORING







85% utilized

10% over-scale

2. Deployment over-scaling

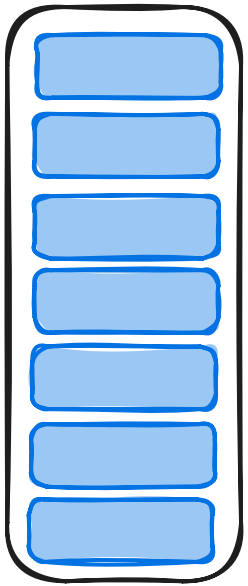
ReplicaSets and Deployments
create extra pods, even with `maxSurge: 0`.

ReplicaSet behavior

2. REPLICASET OVER-SCALING

spec.replicas: 7

= 7 pods

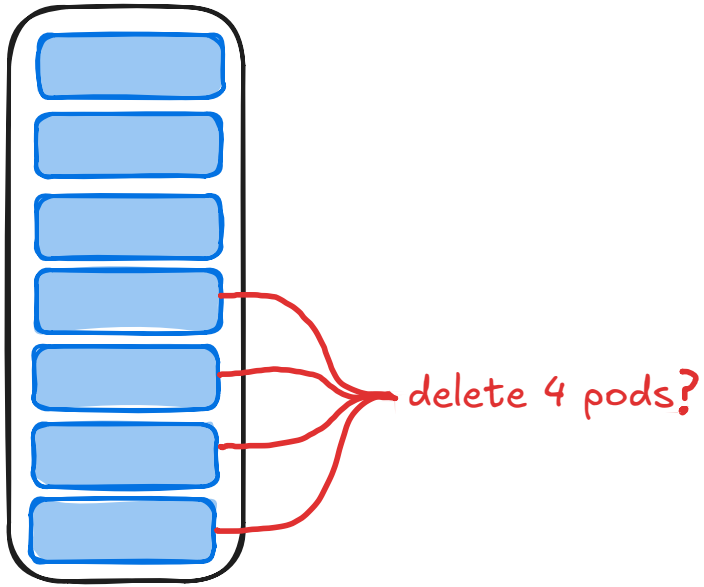


ReplicaSet behavior

2. REPLICASET OVER-SCALING

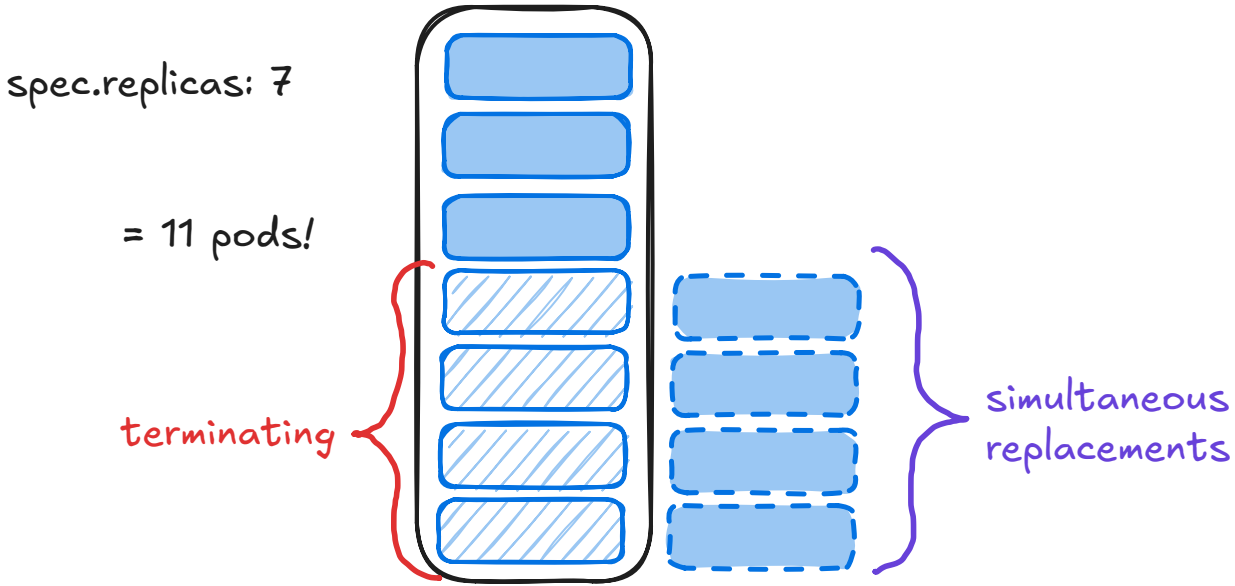
spec.replicas: 7

= 7 pods



ReplicaSet behavior

2. REPLICASET OVER-SCALING

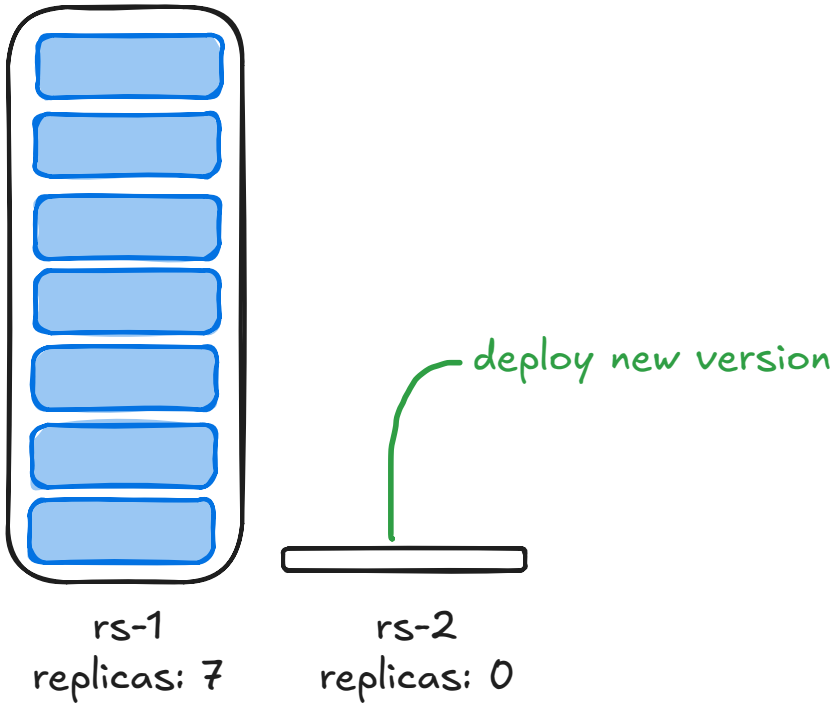


Deployment behavior

2. REPLICASET OVER-SCALING

spec.replicas: 7
maxUnavailable: 3

= 7 pods

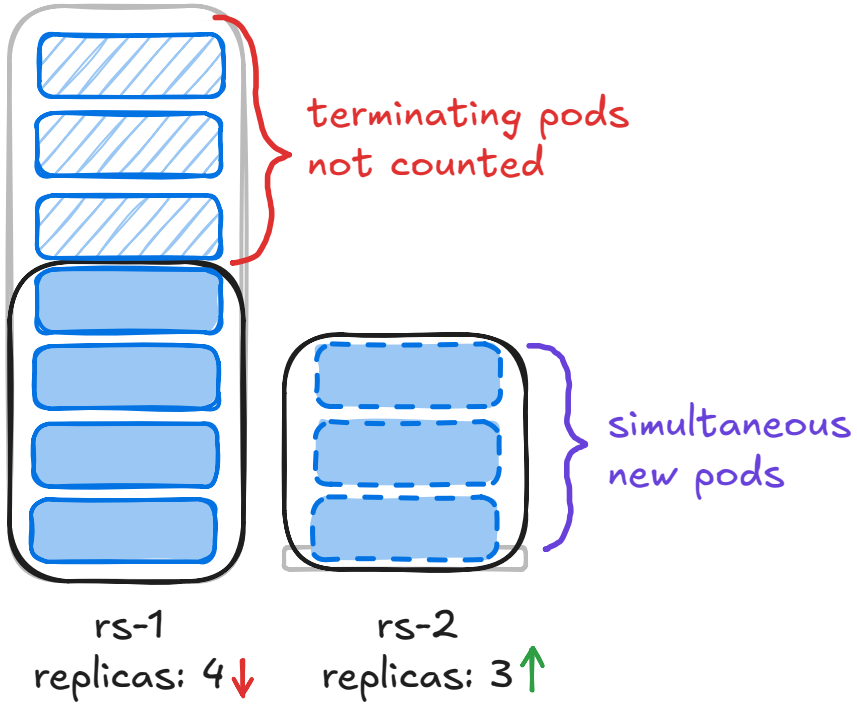


Deployment behavior

2. REPLICASET OVER-SCALING

spec.replicas: 7
maxUnavailable: 3

= 10 pods!

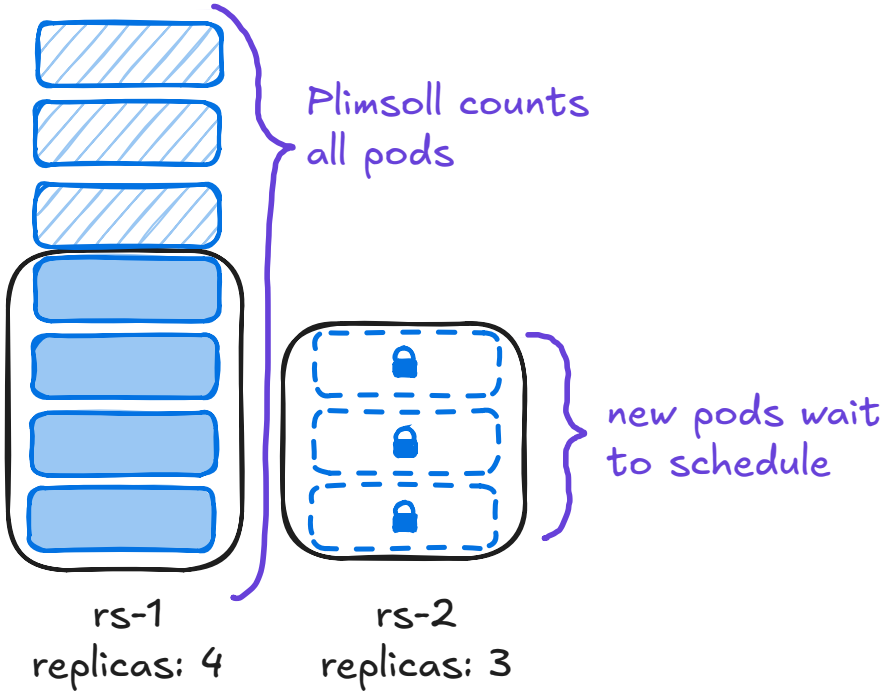


Custom Scheduler Plugin: Plimsoll

2. REPLICASET OVER-SCALING

spec.replicas: 7
maxUnavailable: 3

= 7 pods
allowed to
schedule

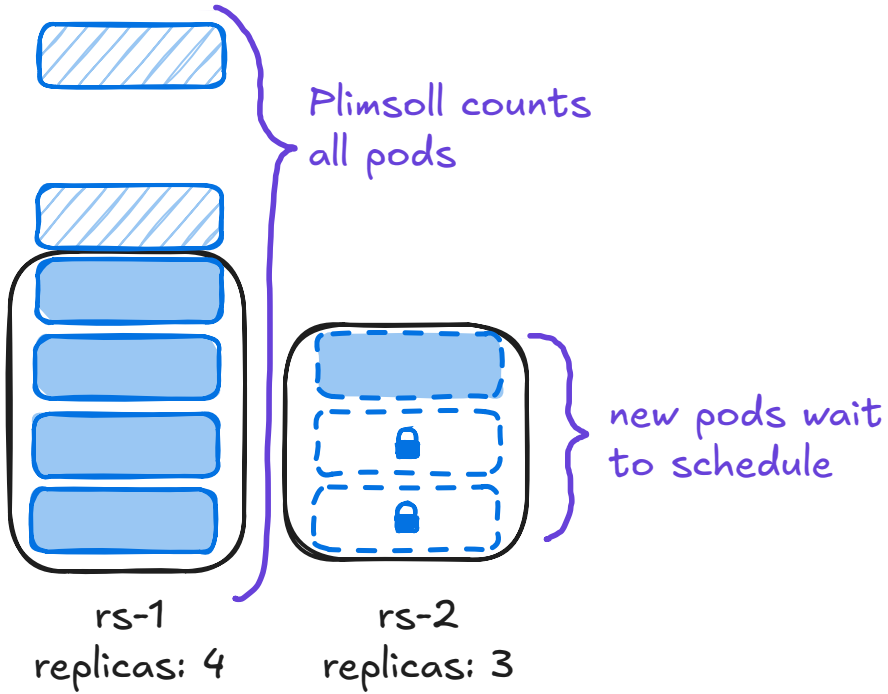


Custom Scheduler Plugin: Plimsoll

2. REPLICASET OVER-SCALING

spec.replicas: 7
maxUnavailable: 3

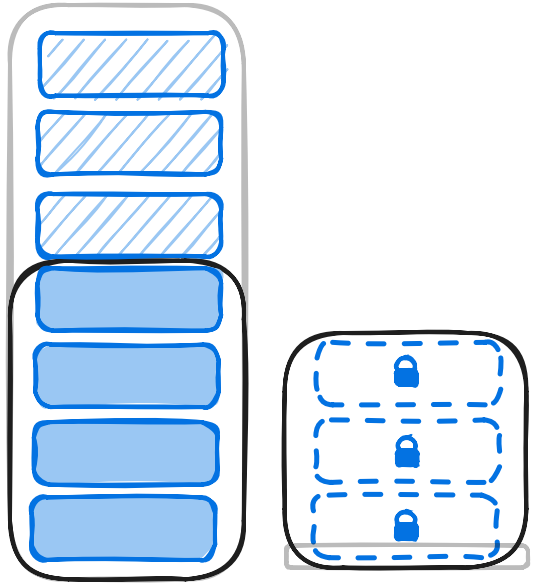
= 7 pods
allowed to
schedule

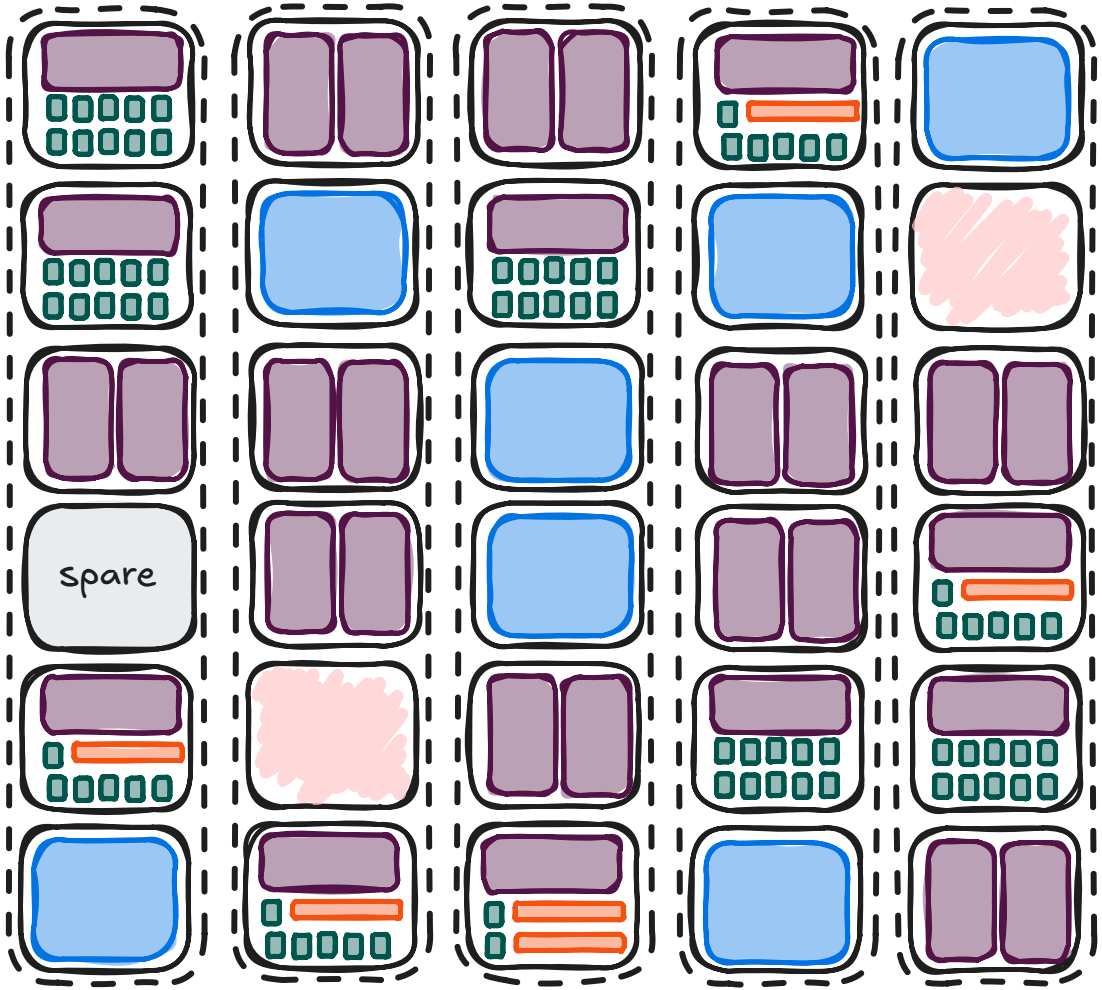


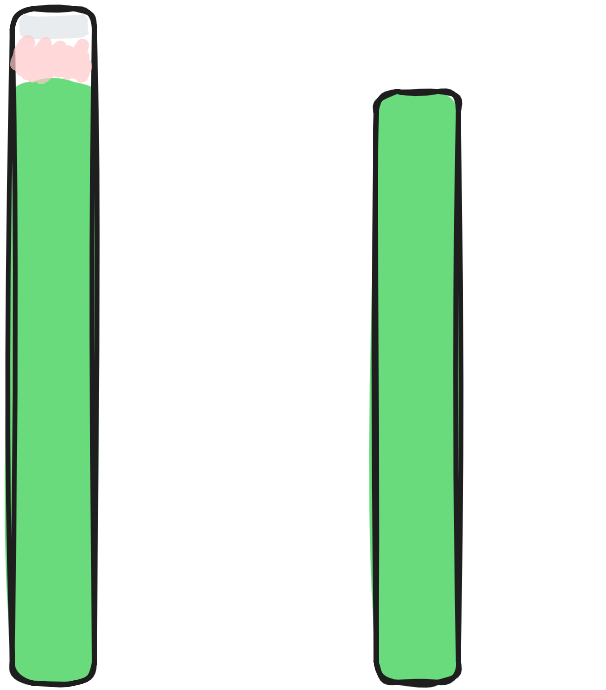
Custom Scheduler Plugin: Plimsoll

2. REPLICASET OVER-SCALING

- Blocks all over-scaled pods from scheduling
- Minimal change to Kubernetes
- Can support out-of-tree controllers like Argo Rollouts




spare


90% utilized
0% over-scale

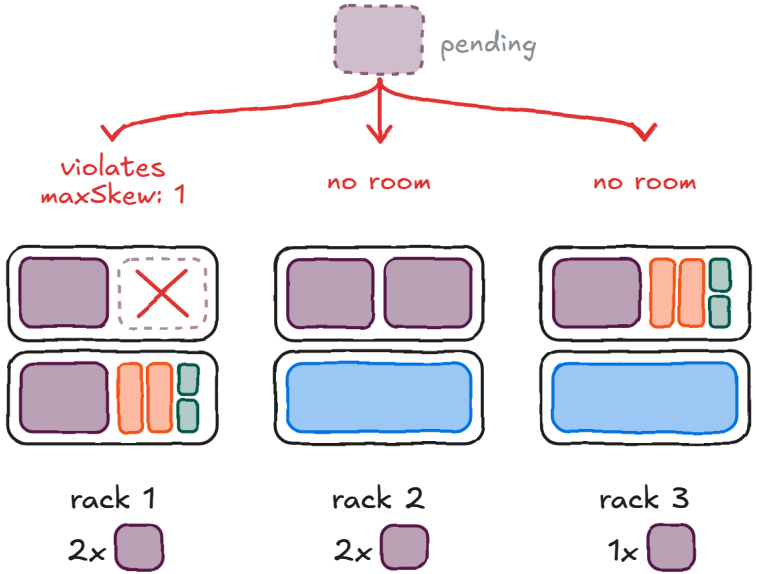
3. Fragmentation

Online scheduling can create suboptimal binpacking.

Causes of fragmentation

3. FRAGMENTATION

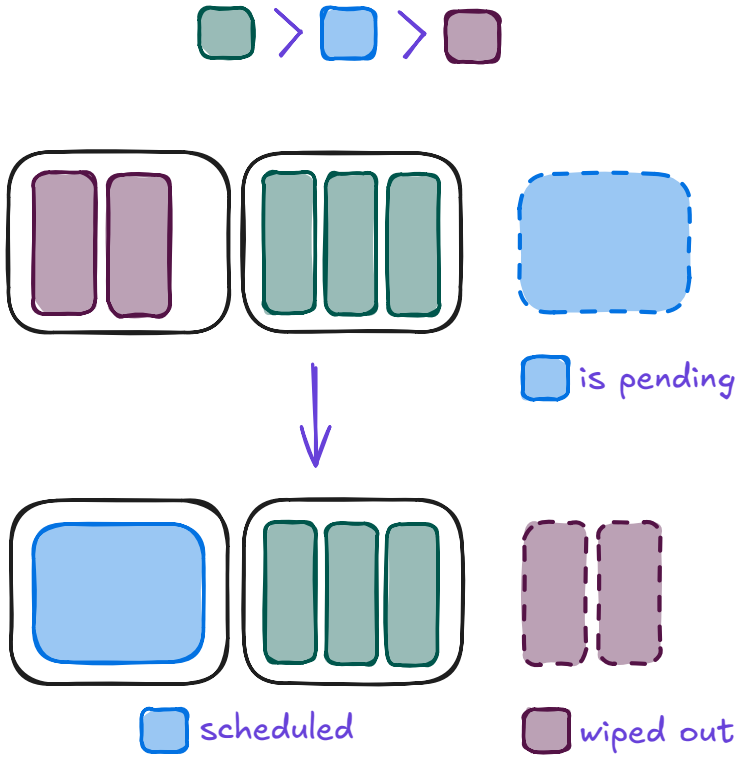
- Pod creation order
- Pod deletion order
- Scale changes
- Complex pod constraints



Why not PriorityClasses?

3. FRAGMENTATION

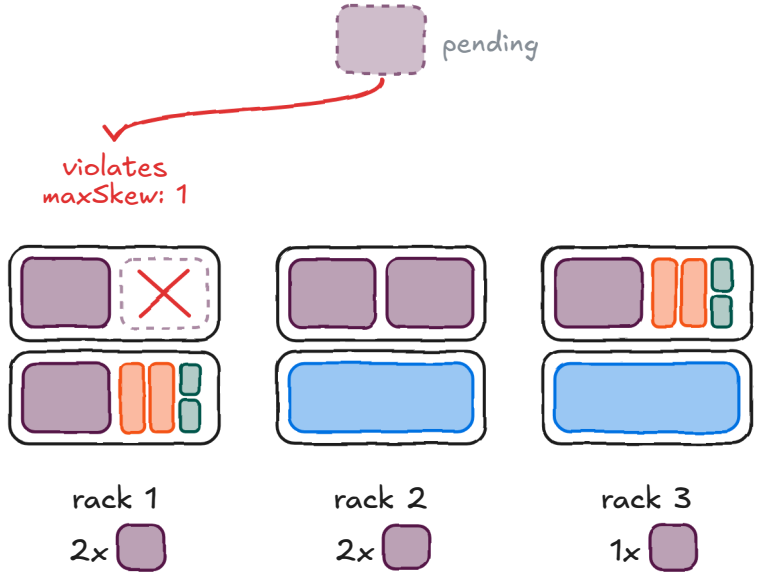
- Preemption only asks "who has lower priority?"
- Exceeding capacity can cause a major outage!



Custom Scheduler Plugin: ResourceShuffler

3. FRAGMENTATION

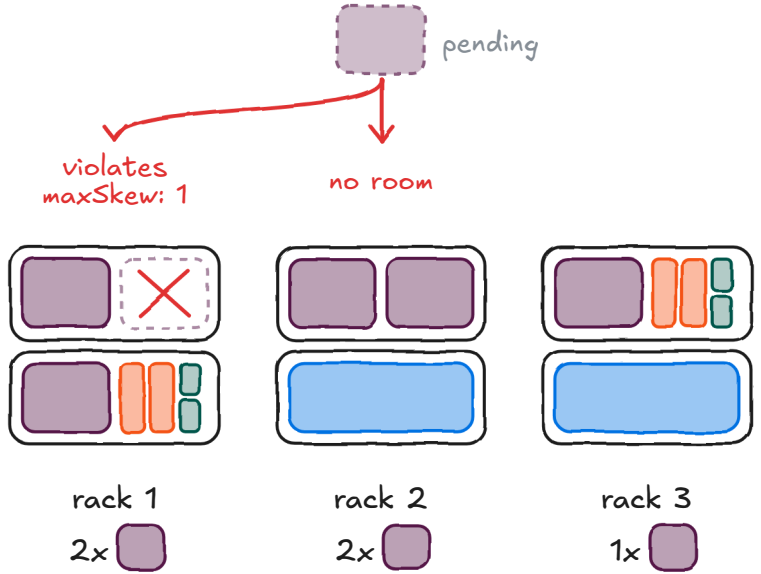
- Runs only when a pod fails to schedule
- Makes room by defragmenting
- Respects PDBs
- Only evicts pods that fit elsewhere



Custom Scheduler Plugin: ResourceShuffler

3. FRAGMENTATION

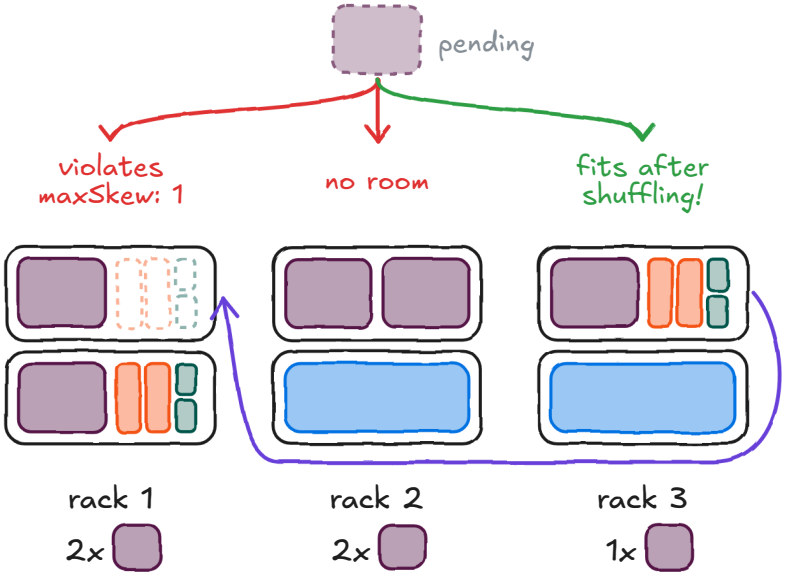
- Runs only when a pod fails to schedule
- Makes room by defragmenting
- Respects PDBs
- Only evicts pods that fit elsewhere



Custom Scheduler Plugin: ResourceShuffler

3. FRAGMENTATION

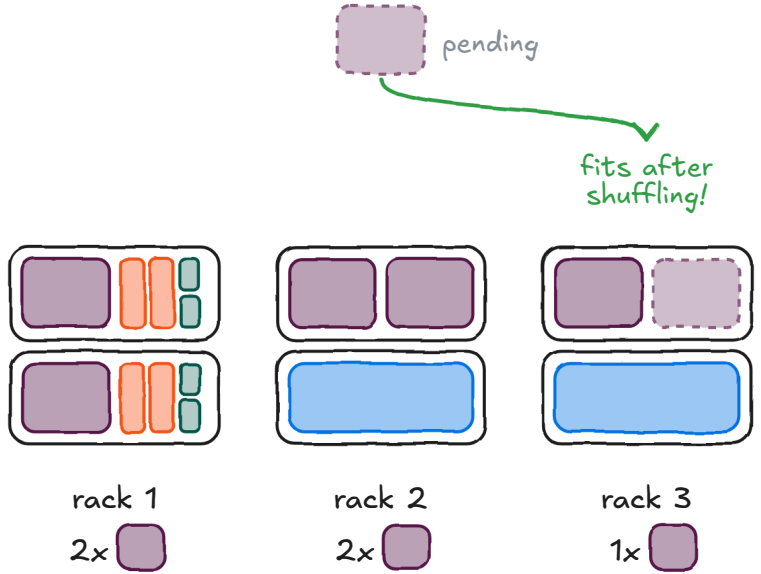
- Runs only when a pod fails to schedule
- Makes room by defragmenting
- Respects PDBs
- Only evicts pods that fit elsewhere



Custom Scheduler Plugin: ResourceShuffler

3. FRAGMENTATION

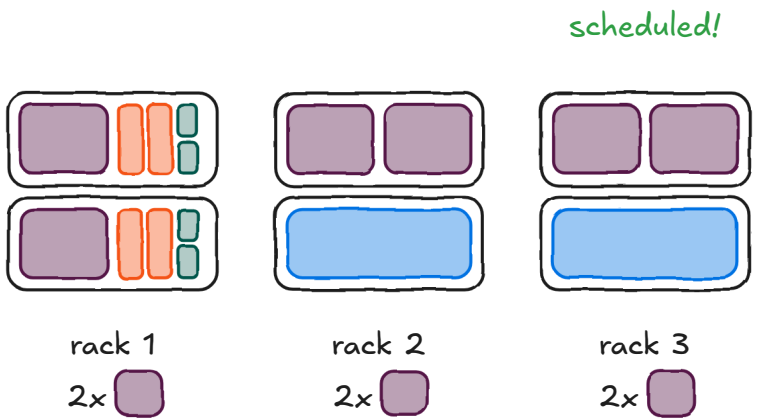
- Runs only when a pod fails to schedule
- Makes room by defragmenting
- Respects PDBs
- Only evicts pods that fit elsewhere

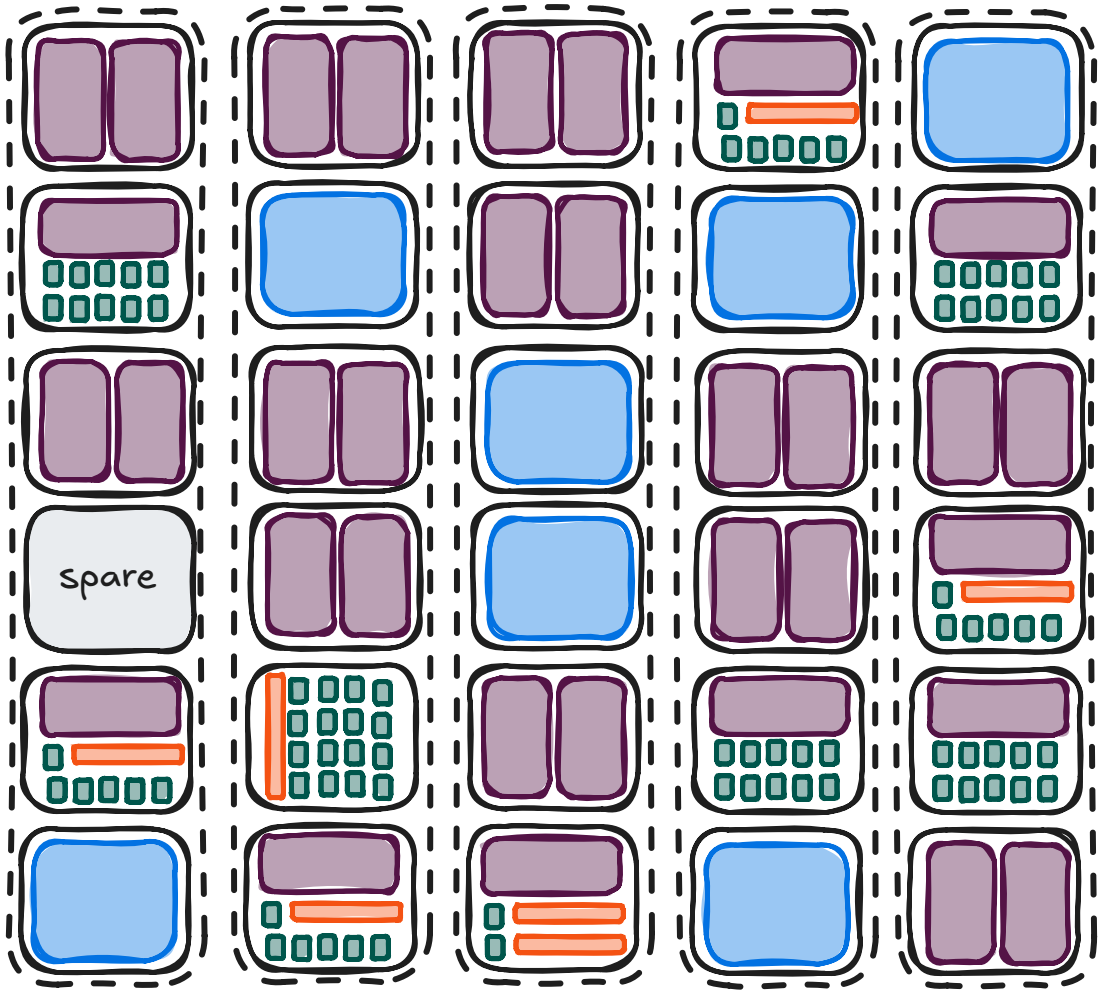


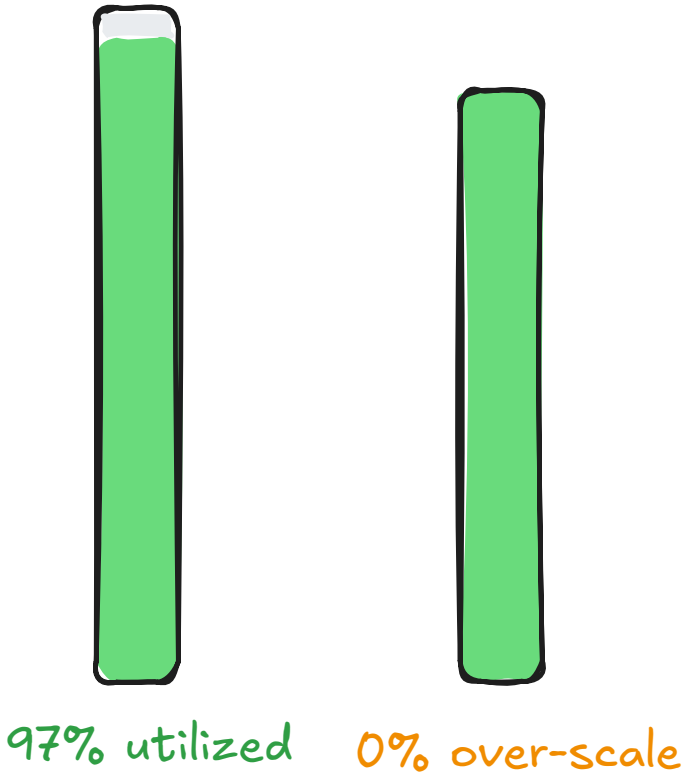
Custom Scheduler Plugin: ResourceShuffler

3. FRAGMENTATION

- Runs only when a pod fails to schedule
- Makes room by defragmenting
- Respects PDBs
- Only evicts pods that fit elsewhere




spare

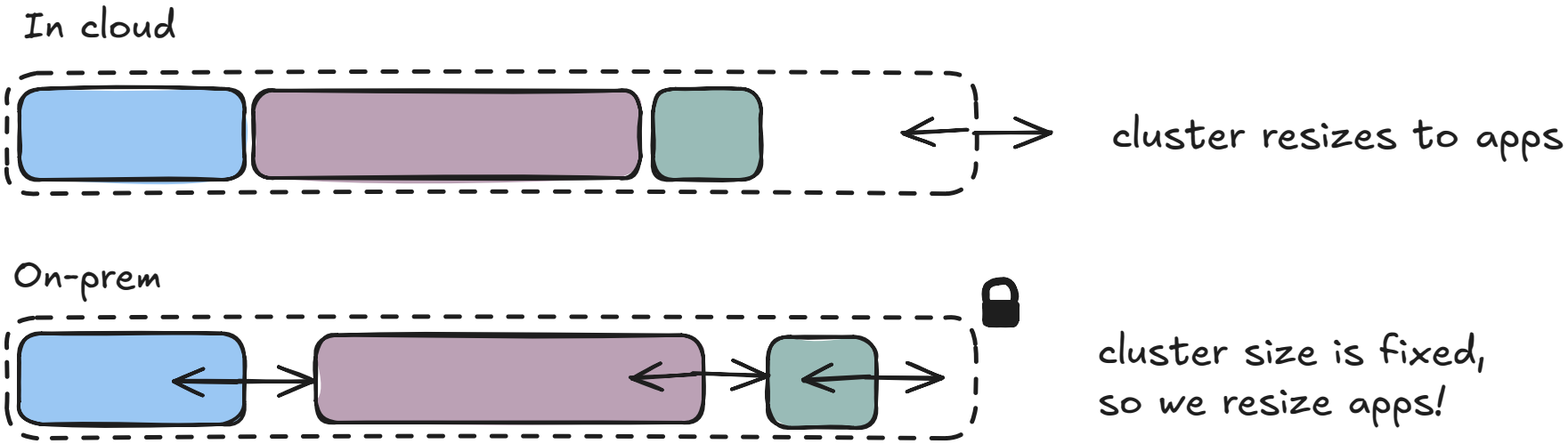

97% utilized
0% over-scale

4. Right-sizing

Adjusting workload and cluster size to
ensure availability and maximize cost efficiency.

On-prem right-sizing

4. RIGHT-SIZING



Problems with manual right-sizing

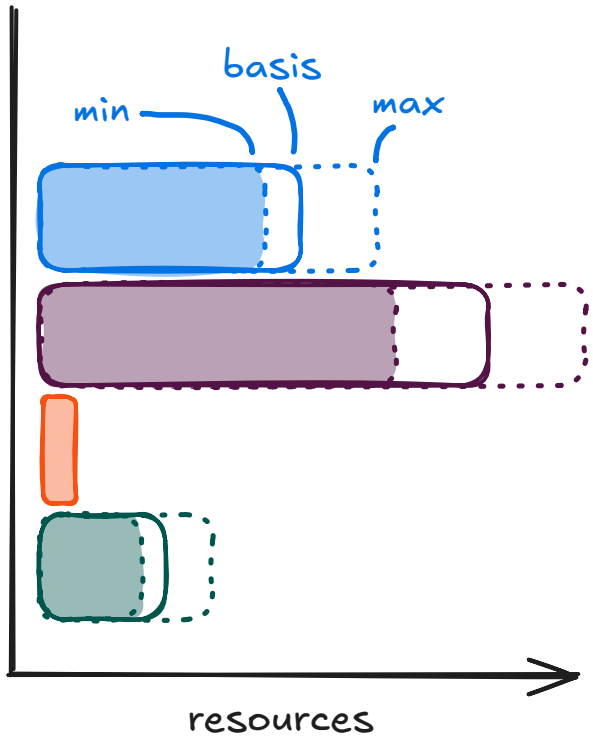
4. RIGHT-SIZING

Uncertain (effective) capacity

- Hardware breaks
- Racks go down for maintenance
- Online defragmentation isn't perfect
- Pod constraints can limit usable capacity

Hard to set workload scales

- Politics between workload owners
- Workload needs change over time
- Must not exceed uncertain capacity



Elastic workload model

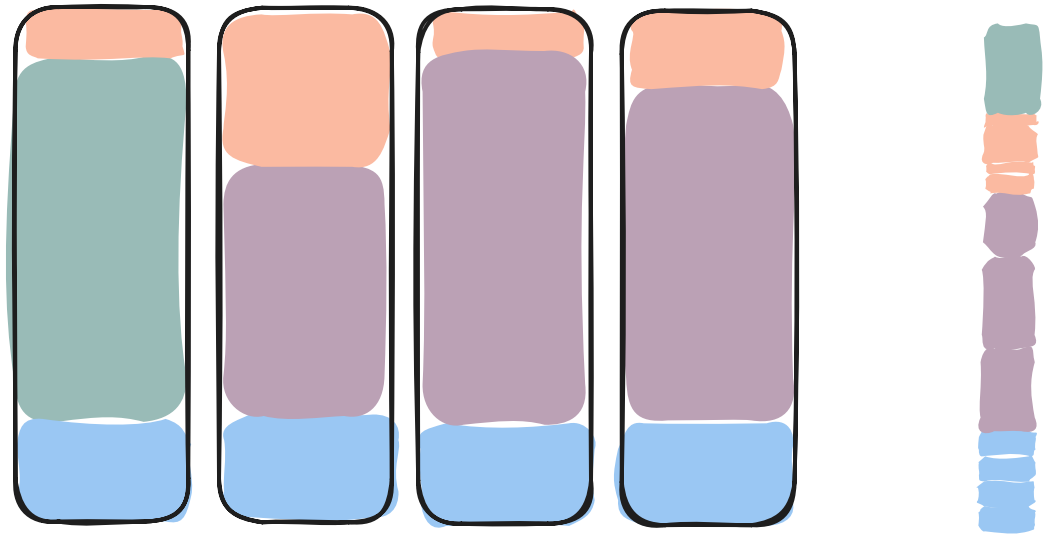
4. RIGHT-SIZING

Inspired by CSS Flexbox:

- `basis` - requested total scale
- `min` and `max` - scale limits
- `shrink` and `grow` (not shown) - willingness to scale away from `basis`

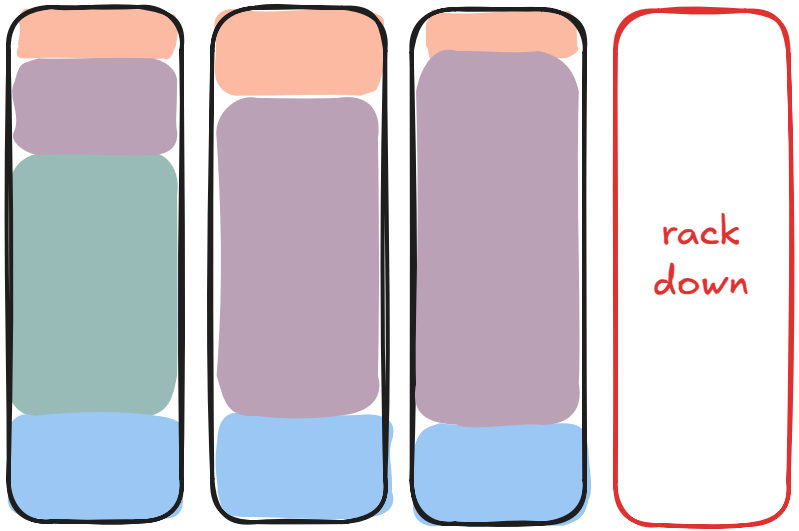
Elastic workloads in action

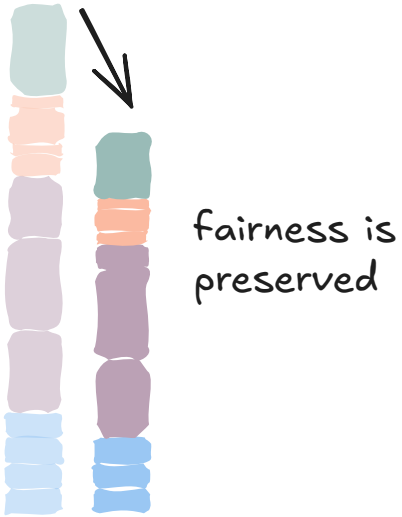
4. RIGHT-SIZING



Elastic workloads in action

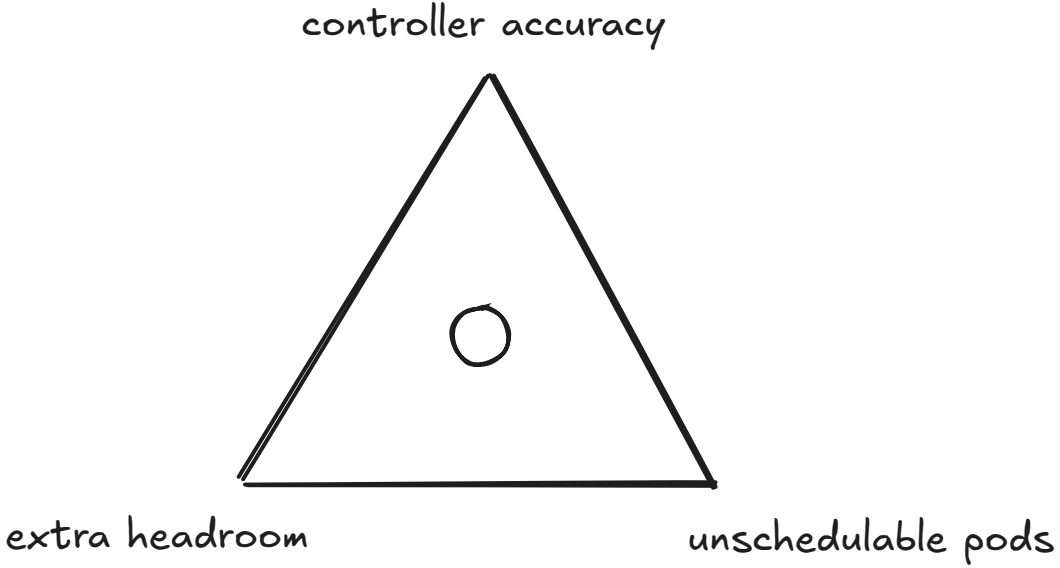
4. RIGHT-SIZING





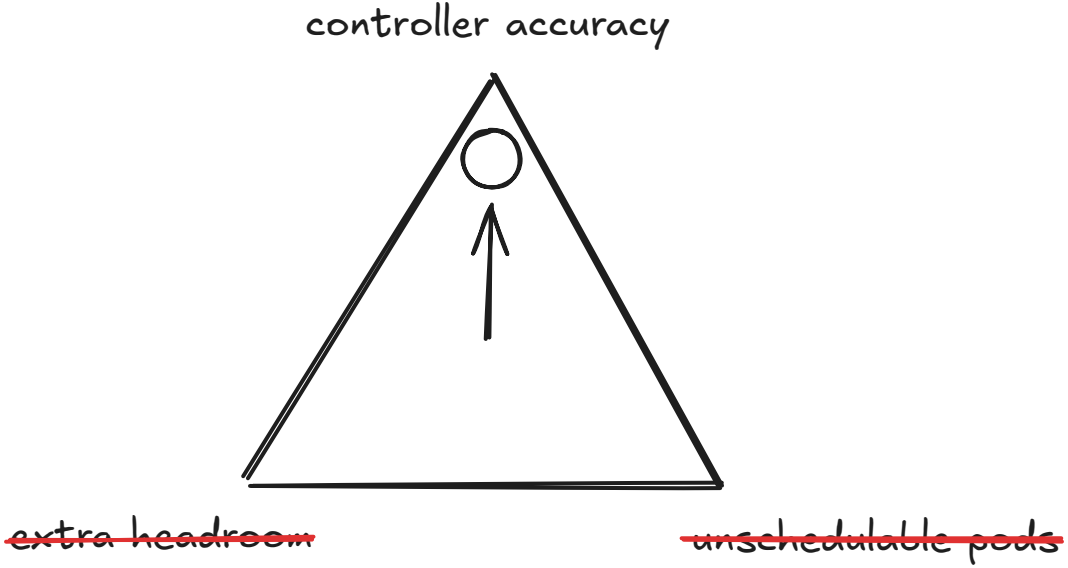
Making a controller

4. RIGHT-SIZING



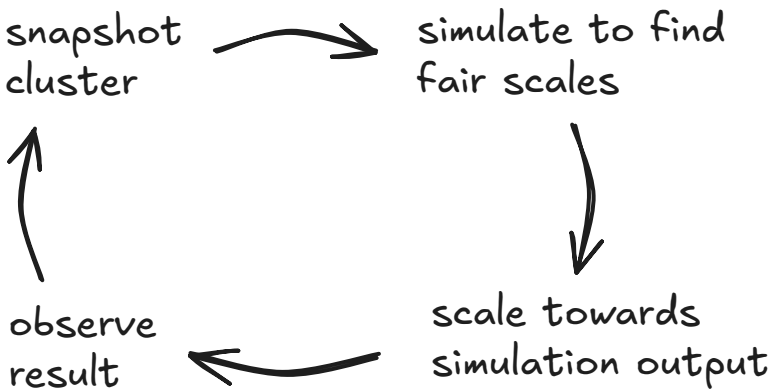
Making a controller

4. RIGHT-SIZING



Making an accurate controller (Bento)

4. RIGHT-SIZING



In simulation:

- Use real Kubernetes scheduler
- Simulate pod deletion order
- Assume as little as possible
- Test minimum changes

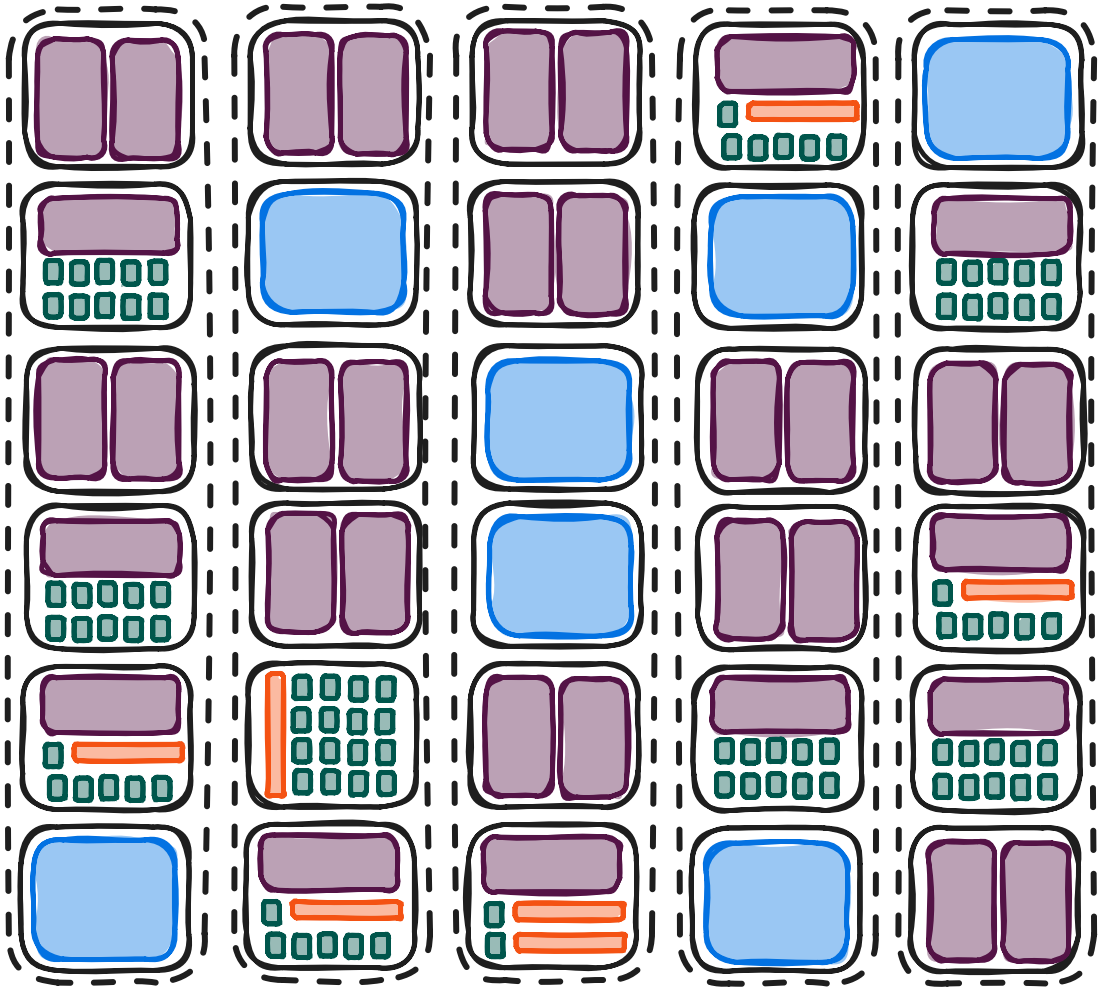
Informing scales

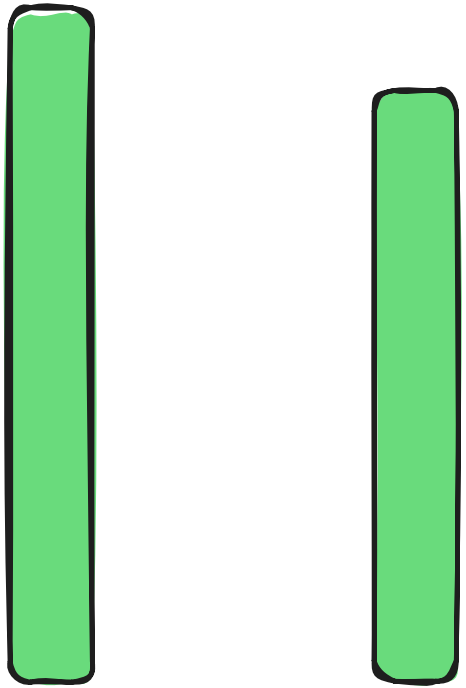
4. RIGHT-SIZING

Simulation can help elsewhere:

- **CI job**: validate workload changes before deployment
- **CLI**: explore allocation scenarios locally

Early feedback for workload owners!





99.9% utilized 0% over-scale

Do you need all this?

1. **MostAllocated** (20% -> 85%) - optimize for big pods, limited space
2. **Plimsoll** (-> 90%) - stop over-scaling during rollouts
3. **ResourceShuffler** (-> 97%) - defragment pod placement
4. **Bento** (-> 99.9%) - right-size workloads to cluster, validate scales

If you can afford the spare capacity, **MostAllocated** is probably enough.

...but we needed it all.

Beyond MostAllocated

If **Plimsoll**, **ResourceShuffler**, or **Bento** seem useful, please reach out at:

dakota.sullivan@thetradedesk.com

I need help convincing my employer these are worth open-sourcing!

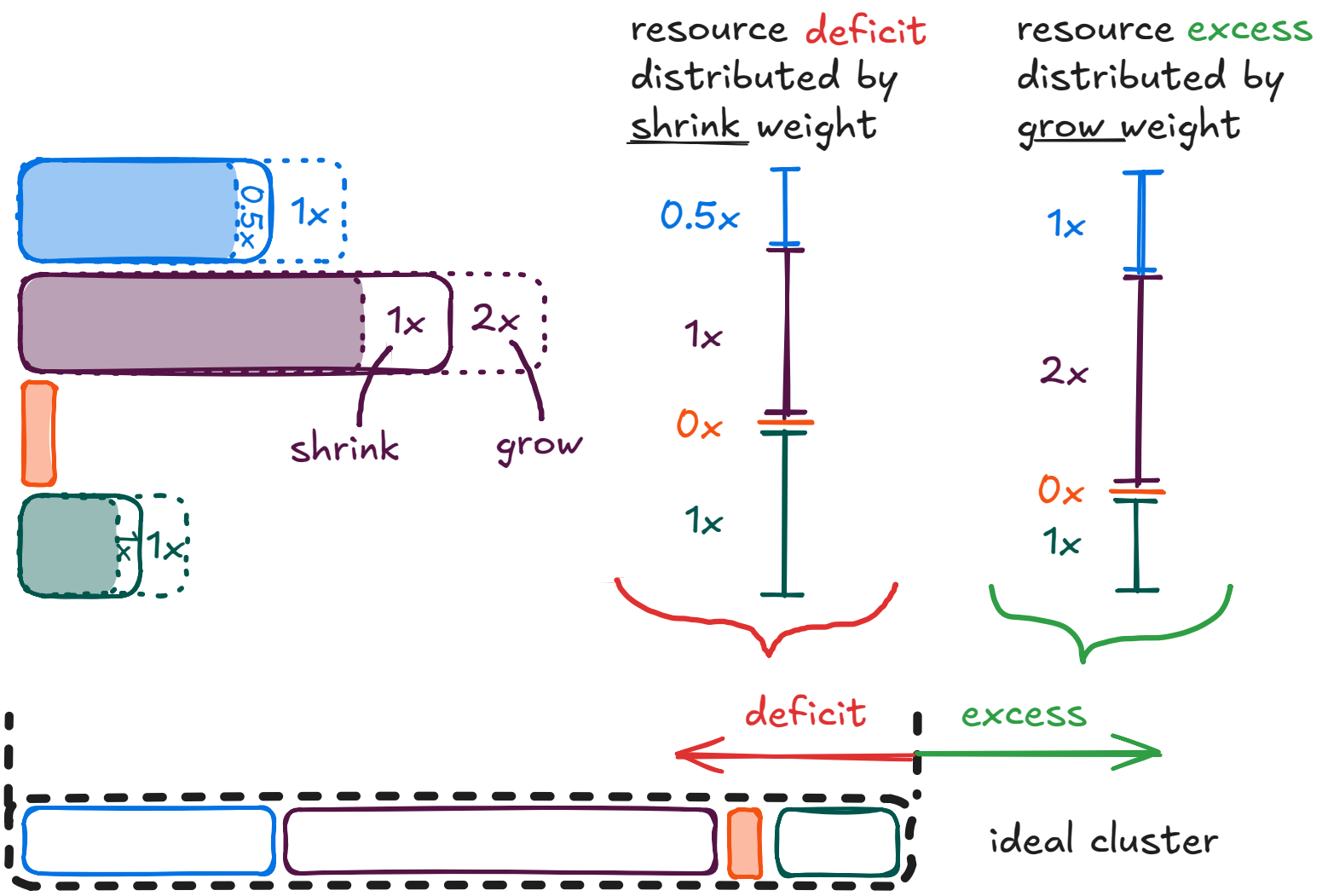
Questions?

(scan QR to give feedback!)

dakota.sullivan@thetradedesk.com
isaac.wilson@elastic.co



Saturated by Design - Maximizing Cluster Allocation in On-Premises Kubernetes


resource deficit
distributed by
shrink weight
resource excess
distributed by
grow weight
0.5x
1x
1x
2x
0x
1x
0x
1x
1x
2x
shrink
grow
deficit
excess
ideal cluster