KubeCon | CloudNativeCon

North America 2024

Better Pod Availability

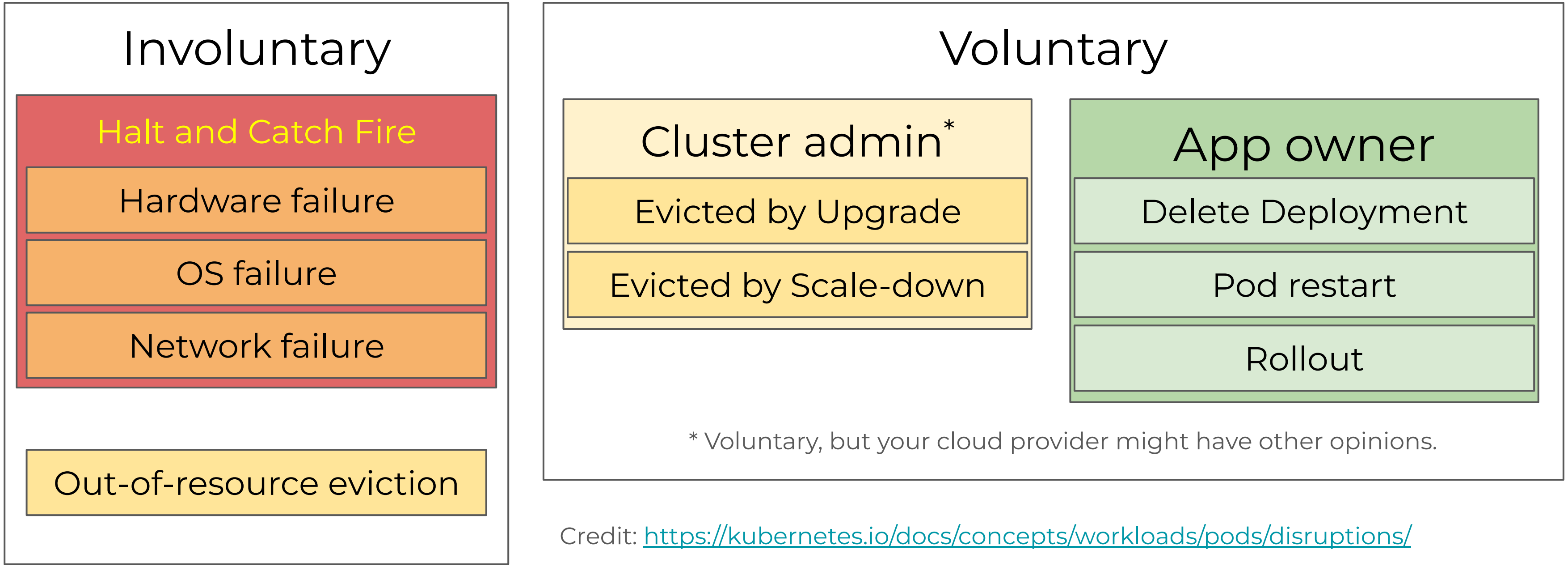
a survey of the many ways to manage workload disruptions

Zach Loafman, Staff SRE, Google

What is Pod Disruption?

Anything that interrupts a Pod before the application exits.

A Taxonomy of Pod Disruptions



Involuntary

Halt and Catch Fire

- Hardware failure
- OS failure
- Network failure

Out-of-resource eviction

Voluntary

Cluster admin[*]

- Evicted by Upgrade
- Evicted by Scale-down

App owner

- Delete Deployment
- Pod restart
- Rollout

[*] Voluntary, but your cloud provider might have other opinions.

Credit: https://kubernetes.io/docs/concepts/workloads/pods/disruptions/

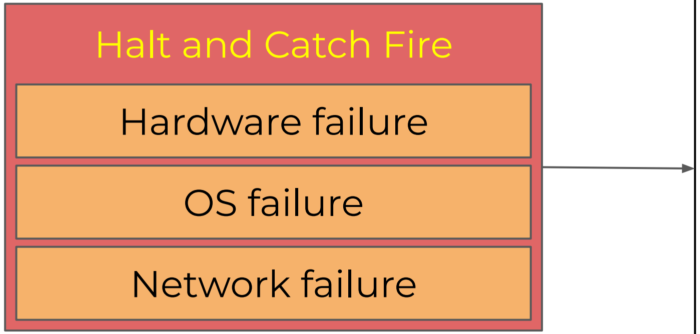
A better taxonomy?

Good Disruption

Pod is interrupted when you want it to be interrupted.

Bad Disruption

Pod is interrupted unexpectedly.



Halt and Catch Fire

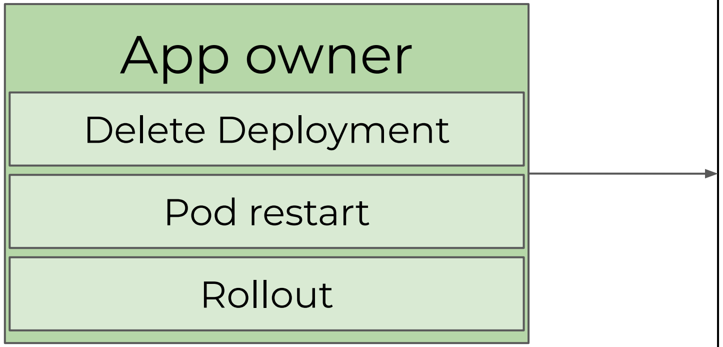
Hardware failure

OS failure

Network failure

Bad Disruption

- Mostly driven by HW/SW quality
- **Best practice: Design for failure**
  - … but that’s another talk
- **How can Kubernetes help?**
  - [Affinity/anti-affinity]
  - [Topology Spread Constraints]
  - etc.



App owner

Delete Deployment

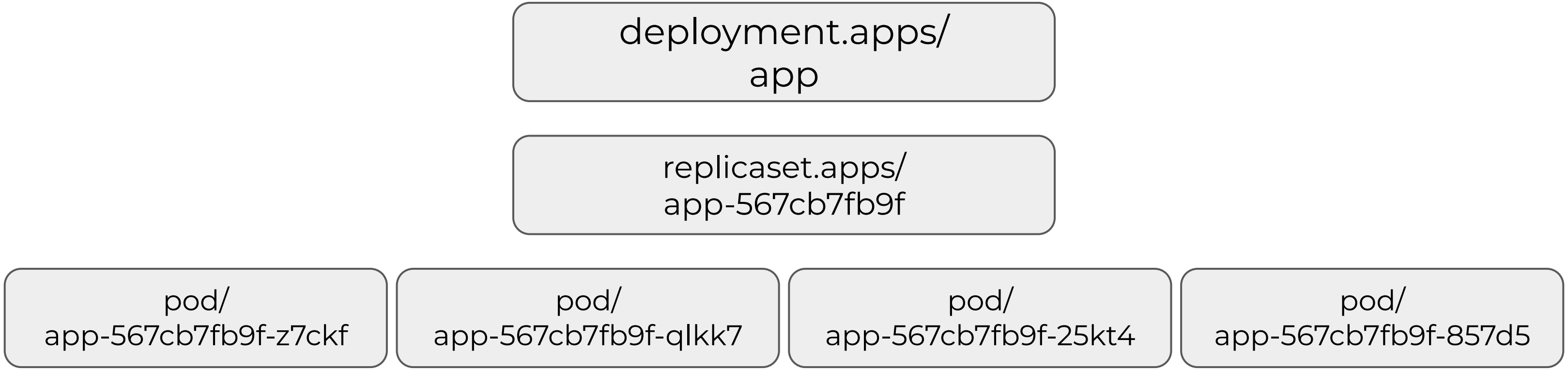
Pod restart

Rollout

Good? Disruption

- Config changes in Kubernetes are *mostly* good disruption.
- **Best practices:**
  - Robust change management: GitOps, multi-party authorization, etc.
  - … but that’s also another talk.
- **How can Kubernetes help?**
  - Deployment rollout strategy
  - `PodDisruptionBudget`

PodDisruptionBudget (PDB)



deployment.apps/
app

replicaset.apps/
app-567cb7fb9f

pod/
app-567cb7fb9f-z7ckf

pod/
app-567cb7fb9f-qlkk7

pod/
app-567cb7fb9f-25kt4

pod/
app-567cb7fb9f-857d5

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: app-pdb
spec:
  maxUnavailable: 1
  selector:
    matchLabels:
      app: app
```

PDB specifies a disruption limit

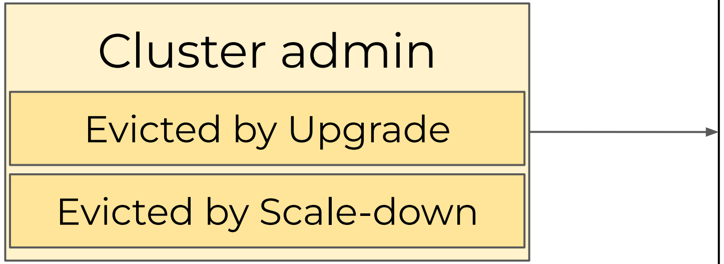
- “at most 1 pod down”, “2 must be available”, “80% must be available”, etc.
- Can be used for any scaled resource
- Honored by *most* voluntary disruptors

If you aren’t using PDBs, you should be.

Out-of-resource eviction

Bad? Disruption

- Not *necessarily* involuntary
- **Best practice:**
  - Tune requests/limits
  - If tuning for reliability, limits==requests
  - If tuning for cost, tune PodPriority
- **How can Kubernetes help?**
  - PodPriority and PriorityClass
- **Warning: Resource eviction honors PDB on best-effort basis**



Cluster admin

Evicted by Upgrade

Evicted by Scale-down

Good? Disruption

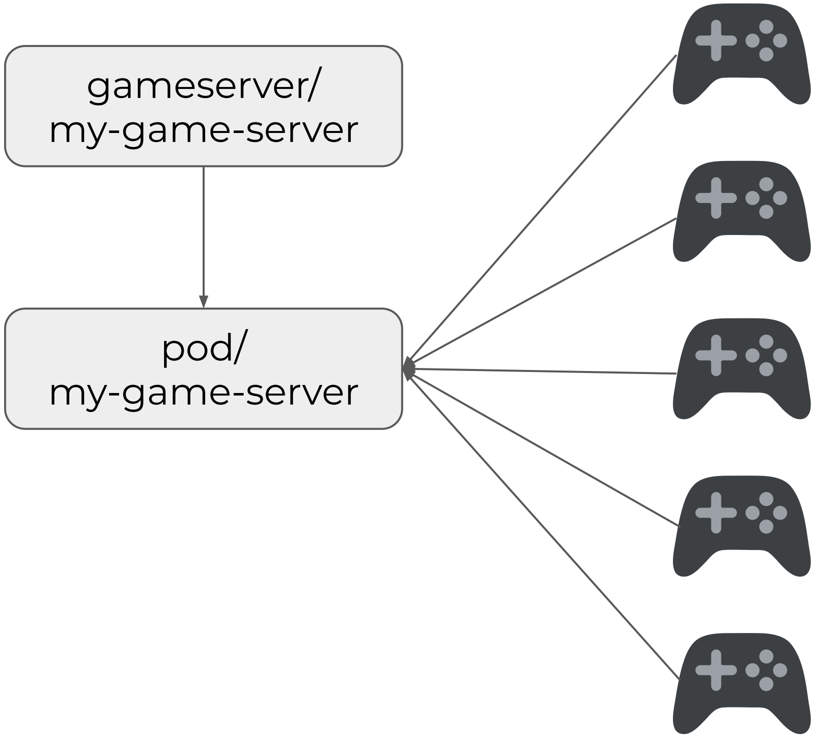
- Good:
  - Keeping your nodes up to date!
  - Saving money!
  - Reduced toil!
- Bad:
  - Without proper config, automation may disrupt your workloads

We’ll focus here for the rest of the talk.

Agones: Dedicated Game Servers

https://agones.dev/

```
apiVersion: agones.dev/v1
kind: GameServer
metadata:
  name: my-game-server
spec:
  ports:
    - name: default
      portPolicy: Dynamic
      containerPort: 7654
  template:
    spec:
      containers:
        - name: game-server
          image: game-server:1.23
          resources:
            requests:
              memory: 64Mi
              cpu: 20m
            limits:
              memory: 64Mi
              cpu: 20m
```



gameserver/
my-game-server

pod/
my-game-server

Agones: A case study in disruption

The Problem:

- Each pod is a game session
  - with its own in-memory state and direct player connections
- A single game session can last from minutes to hours
- Cost/complexity tradeoffs to checkpoint/restart
- Bad pod disruption → **game over** → sentiment/reputation loss

Do you run training jobs or HPC workloads? If so, this may be familiar!

Agones: A case study in pod disruption

As a tradeoff for pods that can’t fail, cluster admins running game servers and similar workloads have a unique set of challenges:

- Halt and Catch Fire → 💀
  - **Mitigation:** Hardware/software quality, monitoring
- Out-of-resource eviction → 💀
  - **Mitigation:** Resource management, PodPriority
- But what about Cluster Admin actions?

Automation: Why do we care?

Cluster Admin actions largely cover “node drain”, either by automation or directly by an admin - **so why allow automation at all?**

- Node repair - with monitoring, possibly avoid Halt and Catch Fire
- Cluster Autoscaler / Karpenter - Bin-pack node resources, save $$$
  - Especially important for workloads like games with diurnal / weekend cycles!
- Node upgrade - keep your software up to date

If you don’t use automation, you are feeding your SREs to the machines. And in the case of repair / autoscaling, you’re probably leaving hard $$ on the table (humans can’t react fast enough).

Node drain: How does it work?

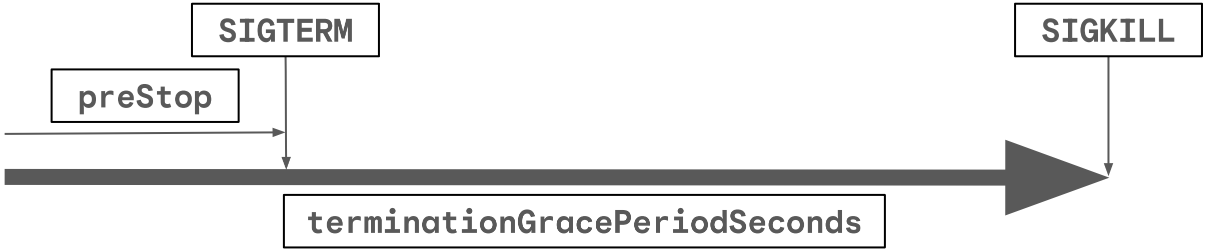
So you or automation want to take a node out of service, what happens?

1. Honors the `PodDisruptionBudget`
2. Gracefully terminates the pod, honoring the `terminationGracePeriodSeconds`

Except these are **lies**, depending on your provider / configuration.

terminationGracePeriodSeconds

- When Kubernetes wants to stop a Pod, kubelet:
  - Runs a **preStop** hook, if defined
  - Sends a **SIGTERM** to the application
  - Waits **terminationGracePeriodSeconds**
  - Kills the Pod, if it's still running
- Effectively, **tGPS** is the "cleanup period" for a Pod.
- Most Pod evictions honor *some* grace period

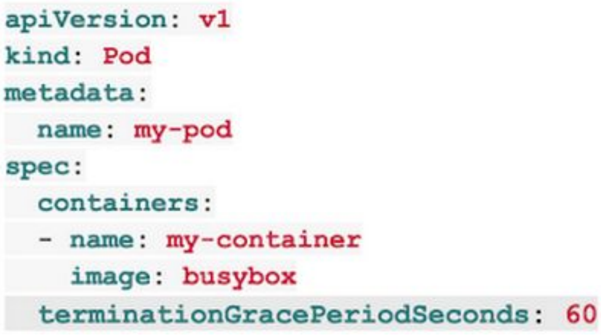


SIGTERM

SIGKILL

preStop

terminationGracePeriodSeconds



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: busybox
  terminationGracePeriodSeconds: 60
```

More reading:

- [Pod Lifecycle]()
- [Lifecycle hooks]()
- [Blog]()

The Cost of Disruption

Fast Yielding

Pod can be evicted in <10m

- Most HTTP/RPC servers
- Most stateful workloads

Slow Yielding

Everything else: Any workload where losing in-memory state is costly (in money, reputation, etc.)

- Session servers (e.g. game servers, voice chat, live transcoding)
- Many AI training jobs
- HPC-ish batch jobs

Disruption by Automation: Node upgrade

Popular cloud providers chose different strategies:

Cloud Provider	Drain timeout behavior (PDB + tGPS)
AKS	Upgrade operation fails after 30m
EKS	Upgrade operation fails after 15m
GKE	Pod disrupted after 1h

Disruption by Automation: Autoscalers

Cluster Autoscaler and Karpenter have different strategies:

	Cluster Autoscaler	Karpenter
Honors PDB	Yes	Yes
Honors tGPS	Max 10m (configurable cluster-wide)	Yes, until configurable per-NodePool maximum
Blocked by?	`cluster-autoscaler.kubernetes.io/safe-to-evict: "false"`	`karpenter.sh/do-not-disrupt: "true"`

Disruption by Automation: Best practices

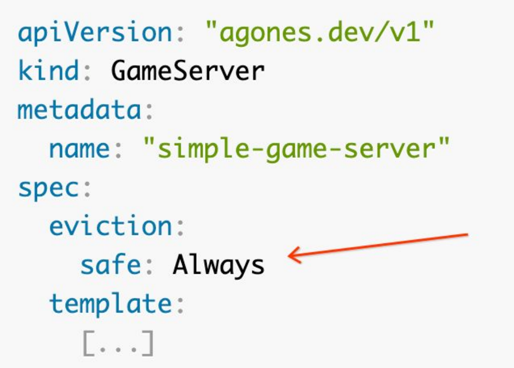
Best practices to support scale-down for slow-yielding apps:

- Ensure your app supports `SIGTERM`
  - If it doesn't, you can use a `preStop` hook to block `SIGTERM`, but it's tricky
- Tune `terminationGracePeriodSeconds` in the pod specification
- Tune maximum `terminationGracePeriod` for your autoscaler
- If you truly can't support scale-down, use a blocking annotation
  - Use sparingly! On a forever-running app, it will prevent scale-down drain
- Affine slow-yielding apps to the same nodes

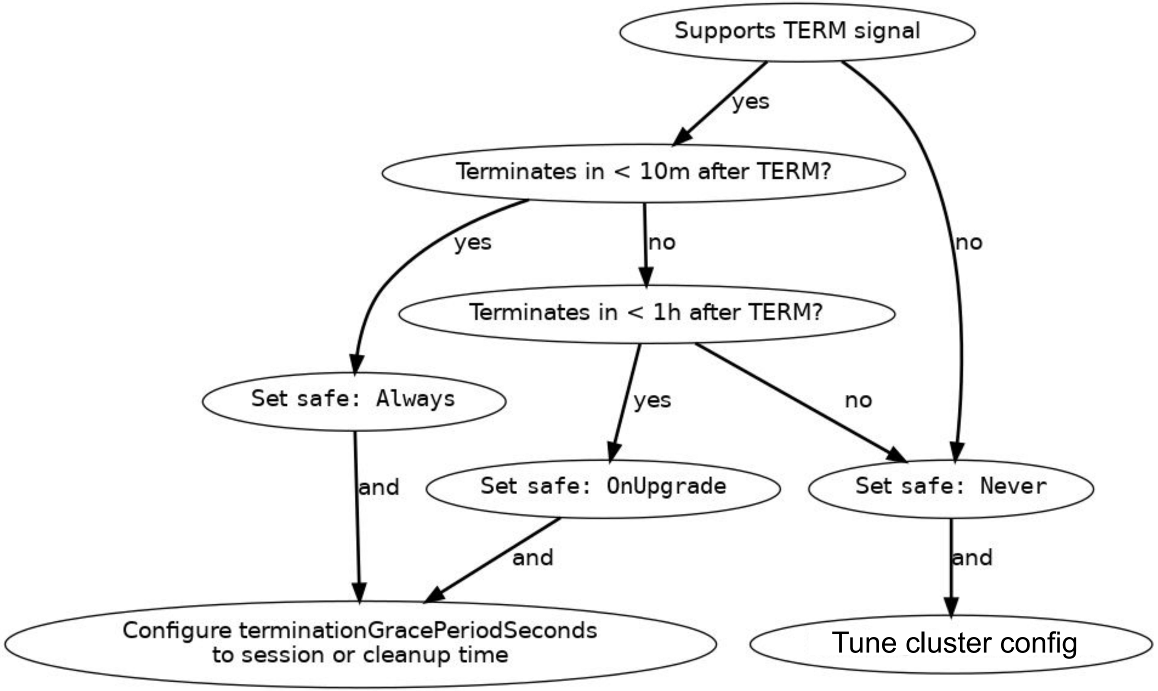
Remember: Supporting scale-down is saving $$$.

Agones eviction API

In Agones, we built an eviction API to try to make this a little easier:



```yaml
apiVersion: "agones.dev/v1"
kind: GameServer
metadata:
  name: "simple-game-server"
spec:
  eviction:
    safe: Always
  template:
    [...]
```



Supports TERM signal

yes

Terminates in < 10m after TERM?

yes

no

no

Terminates in < 1h after TERM?

Set safe: Always

yes

no

and

Set safe: OnUpgrade

Set safe: Never

and

and

Configure terminationGracePeriodSeconds to session or cleanup time

Tune cluster config

Agones eviction API

Goal of eviction API: declared evictable-ness → apply appropriate policies for a given environment (GKE Standard/Autopilot, etc.)

- This goal was only met for GKE, but the scaffolding is there.

A similar API could be constructed w/annotations and a webhook to enforce policies appropriate for your workloads.

- Allows portability between cloud/on-prem, autoscalers, etc.

The two biggest things you need to consider:

- Does the app do the right thing w/SIGTERM?
- How long does it take after SIGTERM before it's safe to terminate?

Conclusion

Takeaways:

- Managing pod disruption on Kubernetes is a series of tradeoffs
  - Multivariable problem of cost, human toil, system complexity, and more
- Think about the **cost of disruption** and engineer appropriately
  - **App owners:**
    - Can you checkpoint, or is it too expensive (in complexity or resources)?
  - **App owners and admins:**
    - With fast-yielding apps, tune your PDBs, consider topology → 🎉
    - With slow-yielding apps, also tune tGPS, tune your automation → 🎉

Questions?