
KubeCon
CloudNativeCon
North America 2024

KubeCon | CloudNativeCon

North America 2024

Navigating Failures in Pods With Devices

Challenges and Solutions

Who we are

Sergey Kanzhelev

Mrunal Patel

SIG Node maintainers. Navigating changes in one of the most complex k8s SIGs.

Motivation

Motivation

- Modern workloads use a lot of new devices beyond traditional CPU/Memory/Network cards.
- AI/ML training require specialized devices and run for weeks to months.
- Device failures could drastically slow down the process.
- Visibility into device failures is crucial for success.

Motivation

Excerpt from Llama 3.1 paper

- During a 54-day snapshot period of pre-training, we experienced a total of 466 job interruptions. Of these, 47 were planned interruptions due to automated maintenance operations such as firmware upgrades or operator initiated operations like configuration or dataset updates. The remaining 419 were unexpected interruptions, which are classified in Table 5. Approximately **78%** of the unexpected interruptions are attributed to confirmed hardware issues, such as GPU or host component failures, or suspected hardware-related issues like silent data corruption and unplanned individual host maintenance events. GPU issues are the largest category, accounting for **58.7%** of all unexpected issues.

Overview of AI/ML workloads on k8s

Why AI/ML runs on k8s?

Kubernetes is catching up with the AI/ML needs with some rough edges.

But still being an orchestration of choice for many:

- Containers are the standard in packaging and deployment workloads
- K8s declarative management and ecosystem works for AI/ML
- Ability to share hardware and mix workloads
- Reliability, security, observability investments cannot be underestimated

Type of AI/ML workload

Oversimplified view on AI/ML types of workload:

1. **Training**
   a. Runs as a job with restartPolicy=Never
   b. Runs on many nodes with interconnected devices.
   c. Takes all devices of used nodes
   d. Need all pods to run at once.
   e. Failure of one pod disrupts the whole “step” of a job.
2. **Inference**
   a. Runs as a pod with restartPolicy=Always
   b. Requires the model and model weights - large chunk of data
   c. Initialization - loading weights - and may take time
   d. Depending on a size can run multi-node (just a few) or on subset of node devices.

Pods are pets, not kettle

Breaking past assumptions

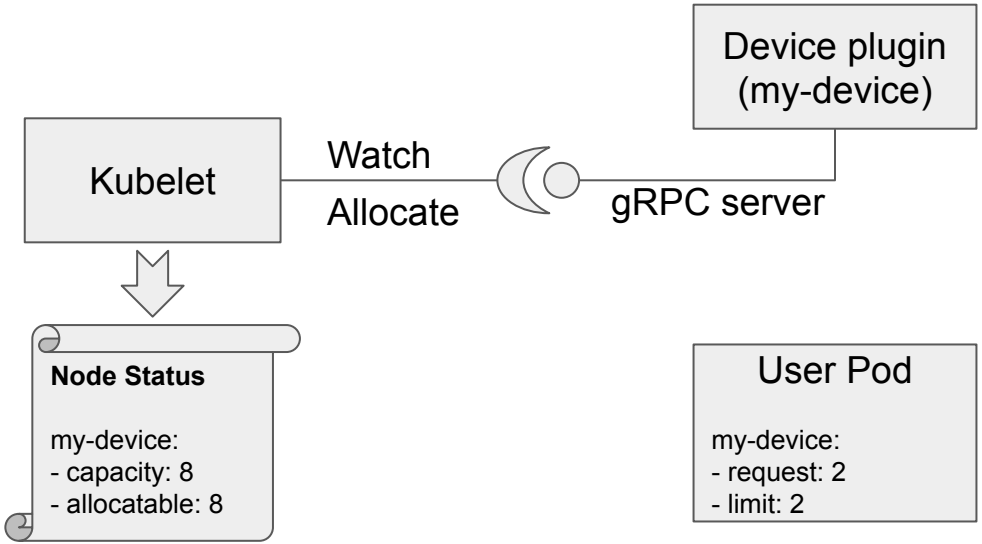
1. Can get a better CPU and app will work faster.
2. When something doesn't work, just recreate it.
3. Any node will work. No need to coordinate between Pods
4. Each Pod can be plug-and-play replaced if failed
5. Images are slim and easily available
6. Long initialization can be offset by slow rollout

1. Require a specific devices (class of devices) to run.
2. Allocation is expensive, re-allocation is expensive
3. Scheduled in special way - devices often connected in a cross nodes topology
4. Part of larger app. Lifecycle of an entire app depend on each Pod
5. Images are heavy
6. Initialization may be long

The existing failure model on k8s has no knowledge about devices and very expensive

Failure modes: k8s infra

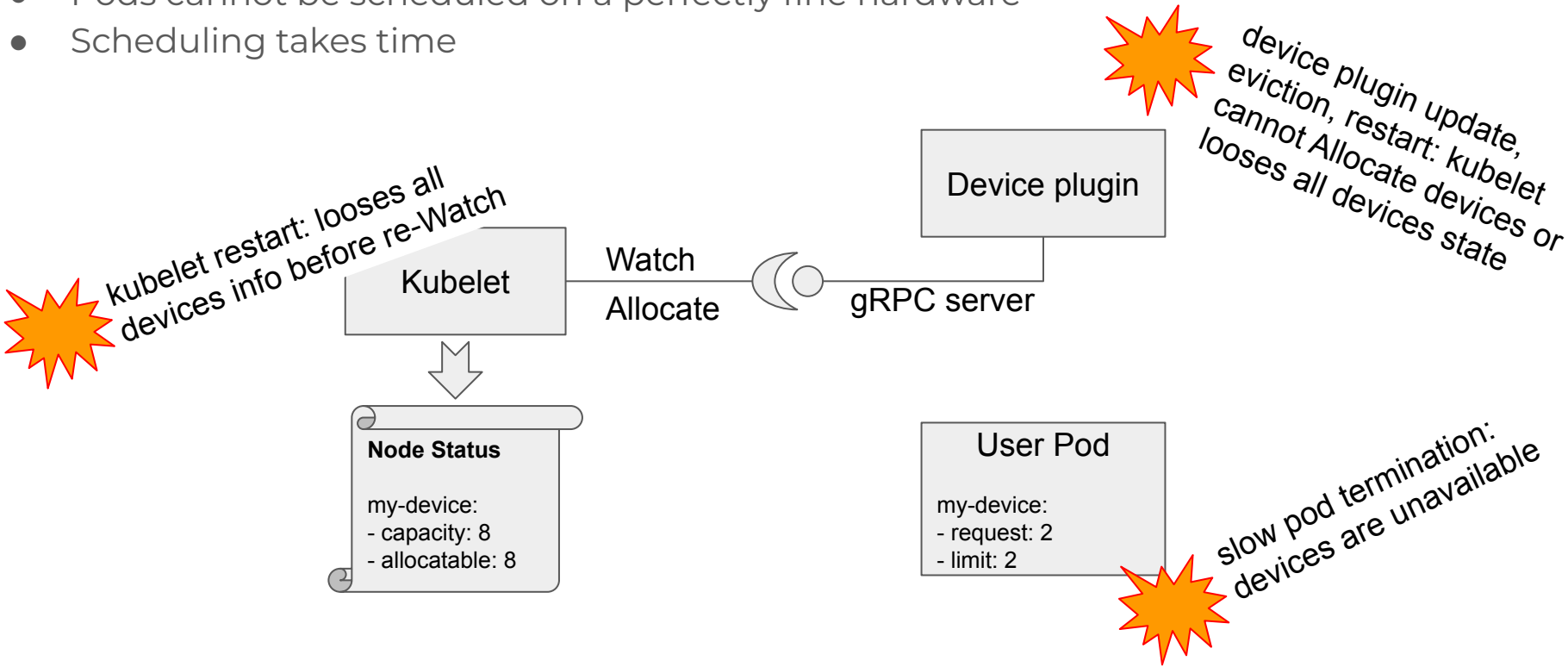
How Device Plugin works



1. Device plugin is scheduled
2. Device plugin is registered with the kubelet
3. Kubelet watches for devices and updates capacity
4. Scheduler places a User Pod on a Node
5. Kubelet asks Device plugin to Allocate
6. Kubelet creates a Pod with the allocated devices

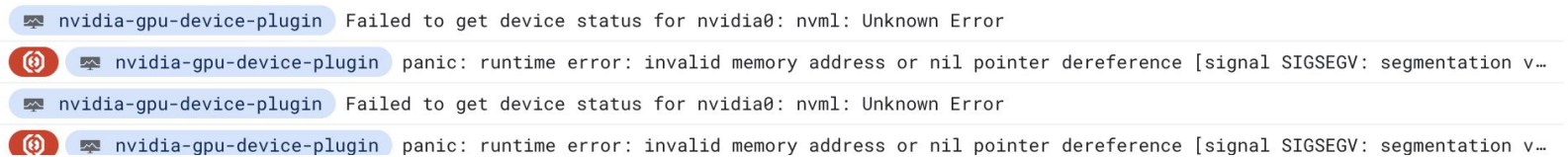
How Device Plugin works: failures and races

- Pods are failing admission
- Pods cannot be scheduled on a perfectly fine hardware
- Scheduling takes time



Work around device plugin issues

- Configure and restart kubelet and containerd as early as possible to not interrupt the workload
- Monitor device plugin health, plan carefully for upgrades
- Do not overload the node with less-important workloads
- Configure user pods tolerations to handle node readiness flakes
- Configure and code graceful termination logic carefully



A bug in Device plugin can render your node unusable

Device driver version mismatch

Another k8s infra-related issue is a driver issues.

Driver version:

- Must match the hardware
- Be compatible with an app
- Must work with other drivers (like nccl, etc.)

```
GPU driver auto installation is disabled.
Waiting for GPU driver libraries to be available.
GPU driver is installed.
InitContainer succeeded. Start nvidia-gpu-device-plugin container.
device-plugin started
Reading GPU config file: /etc/nvidia/gpu_config.json
Failed to parse GPU config file /etc/nvidia/gpu_config.json: unable to read gpu config file /etc…
Falling back to default GPU config.
There is no Xid config specified
Using gpu config: { 0 { 0} []}
```

Best practices handling driver versions

- Monitor driver installers health
- Plan upgrades of infrastructure and Pods to match the version
- Have a canary deployments whenever possible

Failure modes: device failed

Health controller

In many cases failed device will result in unrecoverable and very expensive nodes doing nothing.

Simplest solution is a nodes “health controller”.

- Controller checks Allocatable == Capacity
- If Capacity is greater, than start a timer
- Once timer reaches threshold, kill and recreate a node

Problems with the “health controller”:

- Root cause is not known
- Not workload aware
- Failed device might not be in use and you want to keep other devices running
- Sometimes it is too slow
- Node may be part of a hardware slice and simply cannot be deleted

Can end up with custom controllers with a lot of config

```
kind: ConfigMap
metadata:
 name: xid-config
 namespace: kube-system
data:
 HealthCriticalXid: "32,79,74"
```

xid 79 could be a software issue (2 year back).

Pod failure policy

Per-pod reaction on failed device is another DIY solution.

1. Pod can define special error codes for device failures
2. Pod failure policy can handle device failure in a special way

See Handling retriable and non-retriable pod failures with Pod failure policy

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: job-pod-failure-policy-failjob
spec:
  completions: 8
  parallelism: 2
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: main
        image: docker.io/library/bash:5
        command: ["bash"]
        args:
        - -c
        - echo "Hello world! I'm going to exit with 42 to simulate a software bug."
  backoffLimit: 6
  podFailurePolicy:
    rules:
    - action: FailJob
      onExitCodes:
        containerName: main
        operator: In
        values: [42]
```

Problems with the “Pod failure policy”

- Error codes must be coded carefully
- Only works with Jobs with restartPolicy=Never
- There is no “device failed” condition

Kubernetes just keeps a pod assigned to a device, even if device is reportedly unhealthy.

Custom pod watcher

DIY pods watcher is needed to handle device failures for inference workloads.

Why it is needed:

- Pod will keep being assigned to the failed device forever
- There is no “unscheduling” for pod with “restartPolicy=Always”
- No built-in controllers that delete Pods in CrashLoopBackoff

Problems with the “Custom pod watcher”

- Custom solution
- Impossible to distinguish device failure from the pod logical error reliably

Failure modes: container code failed

Container code failures

AI/ML pods are better rescheduled locally or even in-place.

AI/ML pods are often interconnected and need to be restarted together.

Solution is recreate everything on failure. But it is expensive for AI/ML workloads.

Failure modes: device degradation

Not all device failures are “terminal”

- One device that is lagging behind can slow down the whole training job.
- Device performance may be affected by numerous components in hardware stack.

Solution: Metrics and perf measurement to detect performance deviations and act on it.

Monitoring

Pod Resources API

- Node local kubelet grpc API (/var/lib/kubelet/pod-resources/kubelet.sock)
- Exposes device assignment information
- Used by monitoring agents such as Nvidia DCGM
- Used by Multus CNI
- Requires running a privileged daemon set

Future

Future improvements

Stand-walk-run approach

- K8s have a lot of extensensibility points to implement DIY failures handling
- Many error handling techniques are working, just not efficient
- Drastically changing failing devices handling may break back compat and many set ups

Failure modes: k8s infra

- DONE: integrate kubelet with the systemd watchdog · Issue #127460

Future:

- DRA: detect stale DRA plugin sockets · Issue #128696
- Support takeover for devicemanager/device-plugin · Issue #127803
- Kubelet plugin registration reliability · Issue #127457
- Recreate the Device Manager gRPC server if failed · Issue #128167
- Retry pod admission on device plugin grpc failures · Issue #128043

Failure modes: device failed

Step 0: Device Health in Pod Status: [Add Resource Health Status to the Pod Status for Device Plugin and DRA · Issue #4680 · kubernetes/enhancements · GitHub]

- It is a new extensibility point for many DIY solutions

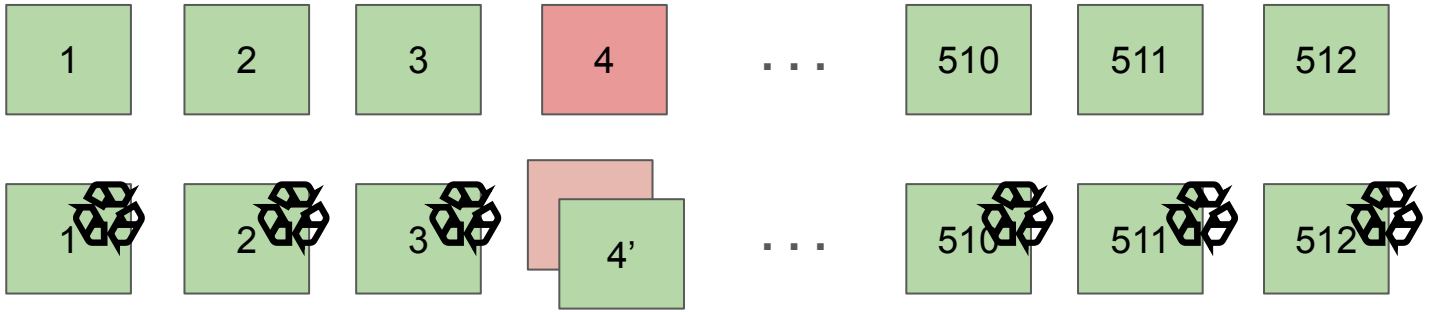
Next steps ideas to be tested:

- Integrate device failures into Pod Failure Policy
- Pod-level retry policies, including Failing a Pod
- Unscheduling pod with restartPolicy=Always to get a new device

Failure modes: container code failed

Coming soon: In-Place container restart KEP

The idea: allow Pod to tell about itself that it needs to be “recycled”. For example, sidecar container watches the progress of a training job and in case one of the Pods failed and being recreated, it recycles own Pod in-place.



Failure modes: device degradation

Devices benchmarking with the workload interruption

We are looking for ways to get performance characteristics from the runtime metrics.

Thank you!