Demystifying Kubernetes Resource Management

Everything You've Always Wanted to Know… But Were Afraid to Ask.

StormForge

Agenda

01.
Why Resources Matter in K8s

02.
Deep Dive on CPU
+ Live Demo 😅🚀

03.
Deep Dive on Mem
+ Live Demo 😬💥

04.
Getting It Right In Practice

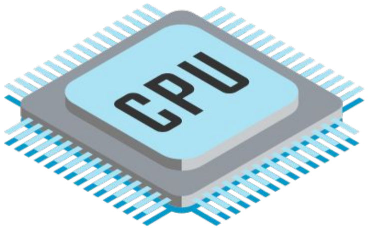
Reid Vandewiele

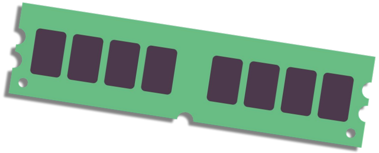
He/Him

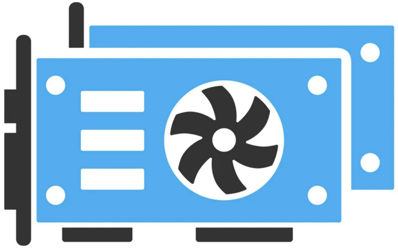
Product Engineer @ StormForge

- 12+ years focus on IT automation
- Tacoma, WA
- Ultimate Frisbee, Mountaineering

The Kubernetes Resources Abstraction

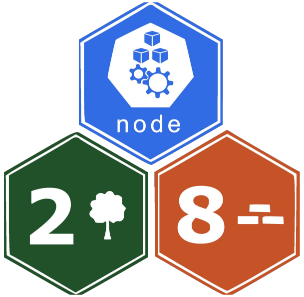






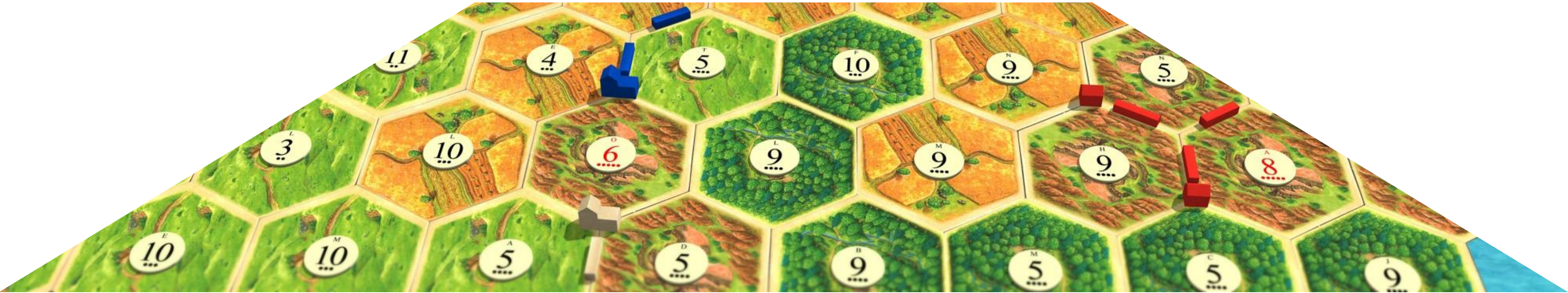
The Kubernetes Resources Abstraction

Nodes have resources



Pods need resources

A cluster has nodes

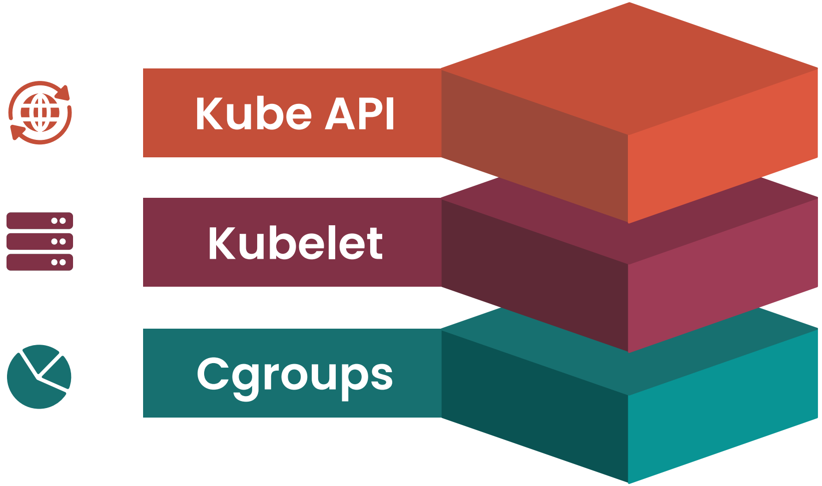


Why Resources Matter

StormForge

Thinking About Resources

Relevant Abstraction or Component Layers



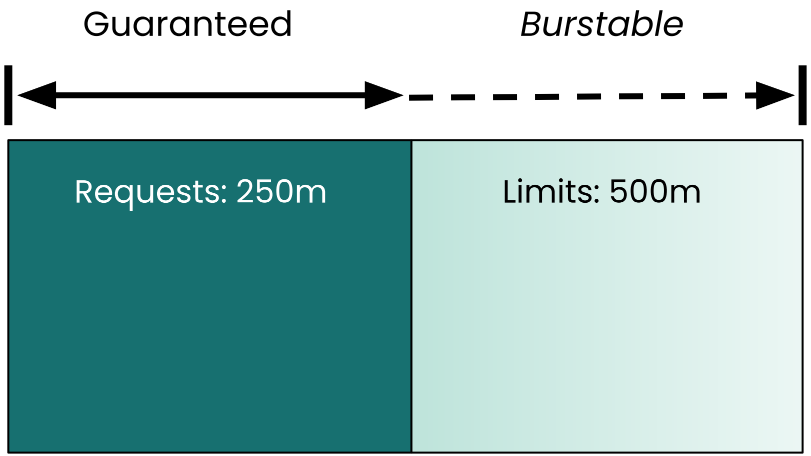
User-facing Abstraction for resource requests and limits

Nodes. Resource **Requests** affect **Pod Scheduling** to nodes

Runtime Implementation of requests & limits abstractions

Resource Basics

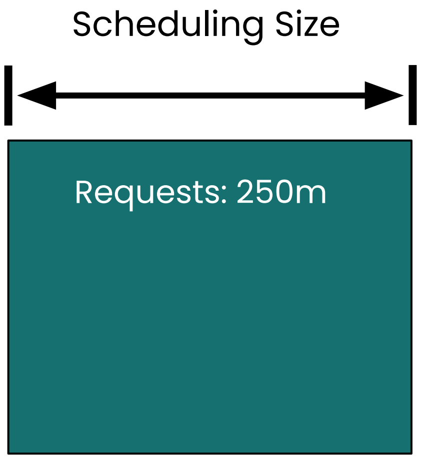
- **Requests** are the minimum resources a container asks for guaranteed access to
- **Limits** are the maximum resources a container is permitted to consume on a node




```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - resources:
      requests:
        cpu: "250m"
        memory: "64Mi"
      limits:
        cpu: "500m"
        memory: "128Mi"
```

Resource Basics

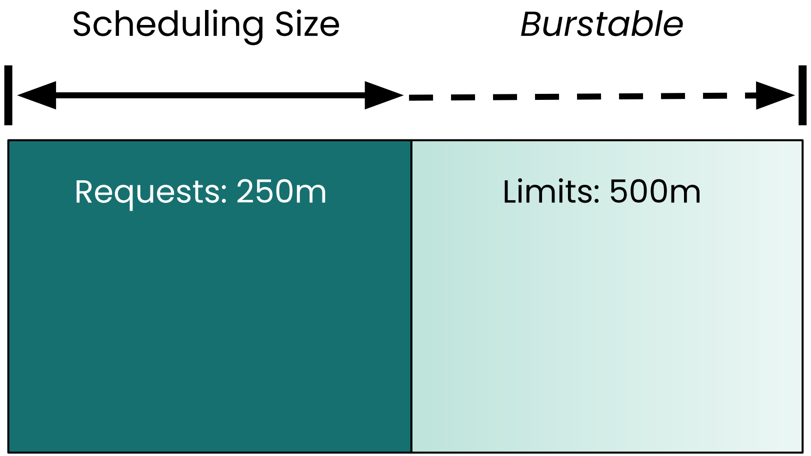
- **Requests** are the the only thing that matters when it comes to Node selection
- The Kubernetes **Scheduler** packs pods onto nodes according to each Node’s available resources and each Pod’s resource requests
- The scheduler **never** overprovisions nodes as measured by Pod resource requests

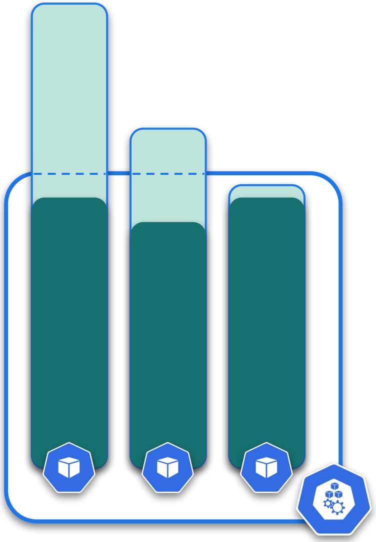


Scheduled Workloads

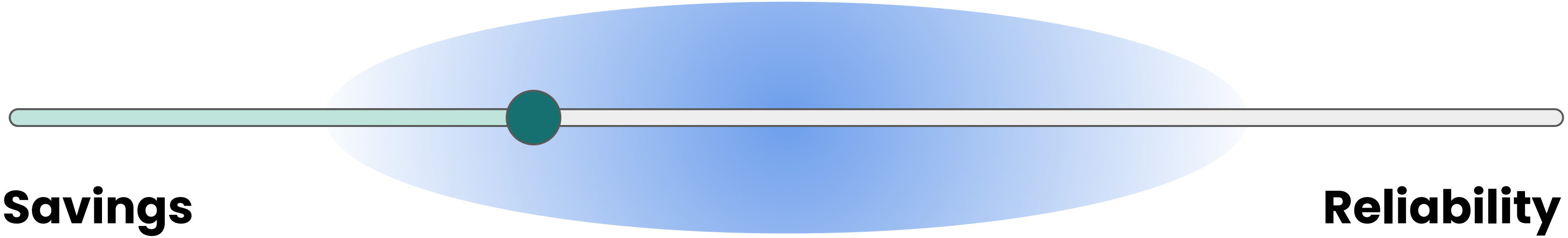
Resource Basics

- **Overprovisioning** of a node’s physical resources is technically possible whenever requests and limits are not equal, including whenever limits are not set
- For many workload types, some degree of overprovisioning* is desirable for cost optimization





To Overprovision, Or Not To Overprovision?

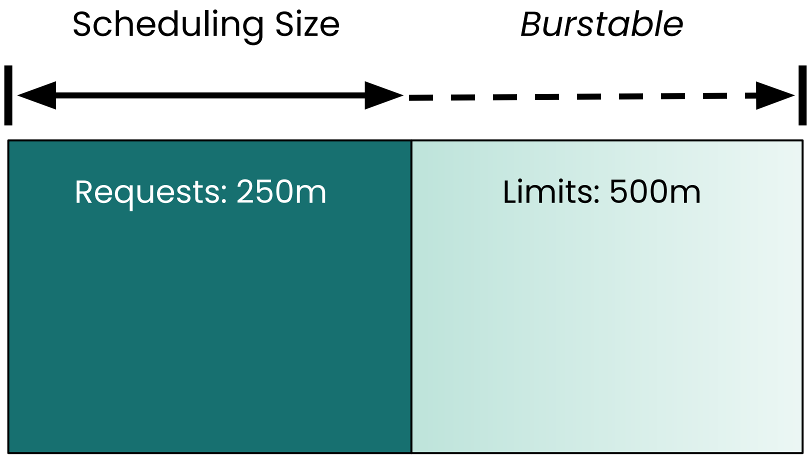


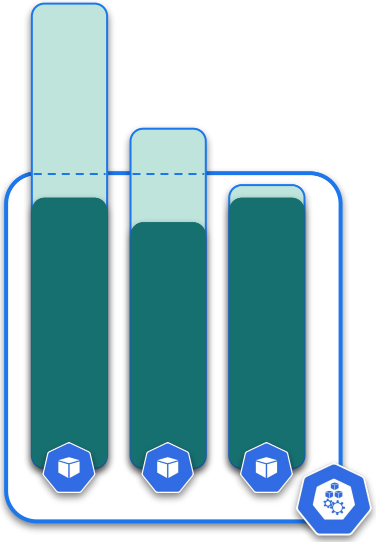
Savings
Overprovisioning
Resource Sharing

Reliability
Overallocation*
Resource Exclusivity

Is Overprovisioning Safe?

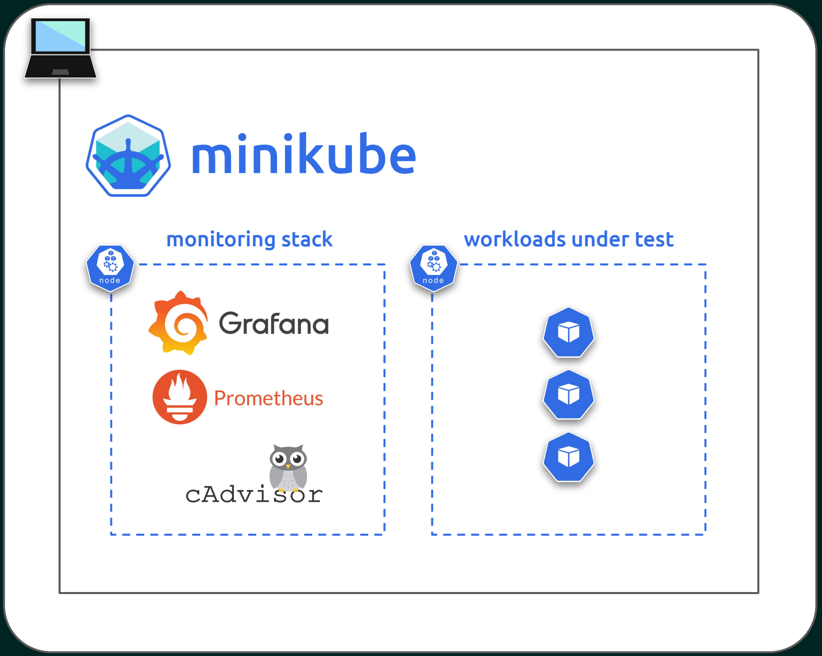
- What are the consequences of overprovisioning for CPU?
- What are the consequences of overprovisioning for Memory?





Scheduled Workloads

Live Experiments Environment Overview



Experiment 1

Resource Settings and CPU Contention

Experiment 1 Review

- ✓ No Requests = No CPU time during contention
- ✓ Usage less than Requests = No interruption during contention*
- ✓ No CPU time *now* = More CPU usage *later* (potentially)

CPU Requests and the Completely Fair Scheduler

"A proportional share scheduler which divides the CPU time (CPU bandwidth) proportionately between groups of tasks (cgroups) depending on the priority/weight of the task or shares assigned to cgroups."

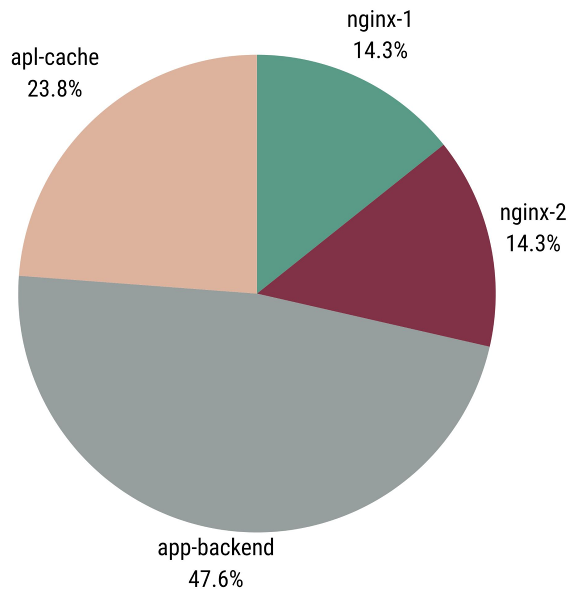
- Kubernetes equates 1 CPU = 1024 CFS shares
- K8s assigns CFS shares to containers based on their CPU requests
- Important for the abstraction mental model:
  No cgroup allocations except by Kubernetes

CPU Requests and the Completely Fair Scheduler

All Containers on a 2-CPU Node

Container	Request	Shares	Effective
nginx-1	150m	153	284m
nginx-2	150m	153	284m
apl-backend	500m	512	953m
apl-cache	250m	256	476m
redis-1	0	-*	-
redis-2	0	-*	-

Proportional CFS Share Assignment



What About The Others?

CPU Limits and the Completely Fair Scheduler

- K8s assigns CFS “quotas” to every container based on the limits
- CFS quotas limit maximum CPU usage but enforcement can have unexpected effects on latency
- **It’s complicated…**

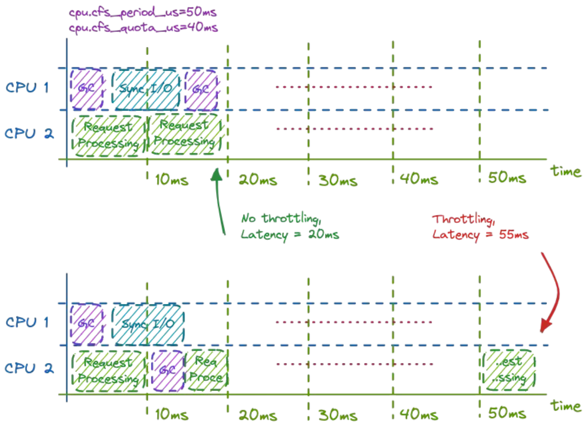
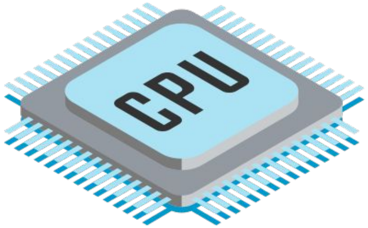


Image credit: JettyCloud @ Medium
“Making Sense of Kubernetes CPU Requests And Limits”
https://medium.com/@jettycloud/390bbb5b7c92

LESSONS LEARNED:

On The Importance of CPU Requests and Limits



Requests are very important.

- Having CPU requests is a minimum guarantee of priority access to the CPU, even during contention*
- Not having any CPU requests makes pods potentially subject to complete CPU starvation

Limits aren't *as* important.

- Limits aren't necessarily needed for Noisy Neighbor reasons **IF** all workloads have reasonable CPU requests (see above)
- Limits are most useful if your requests are wrong for some workloads

Experiment 2

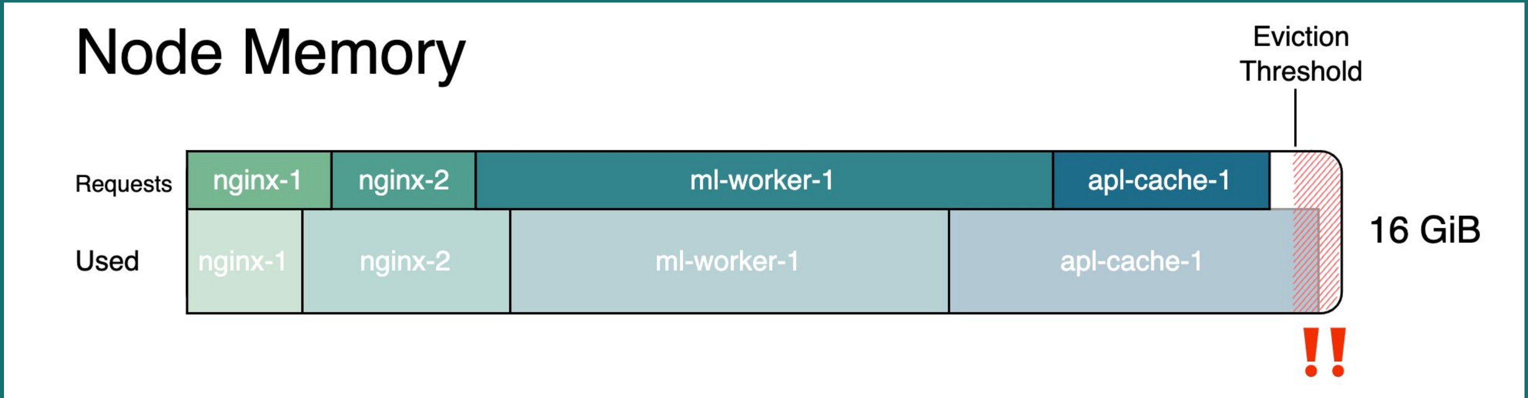
Resource Settings and Memory Contention

Experiment 2 Review

- ✓ Containers with memory limits will be killed if they exceed their individual limit
- ✓ Containers either get the memory they allocate STAT, or something will get OOMkilled
- ✓ What gets OOMkilled? Not deterministic…

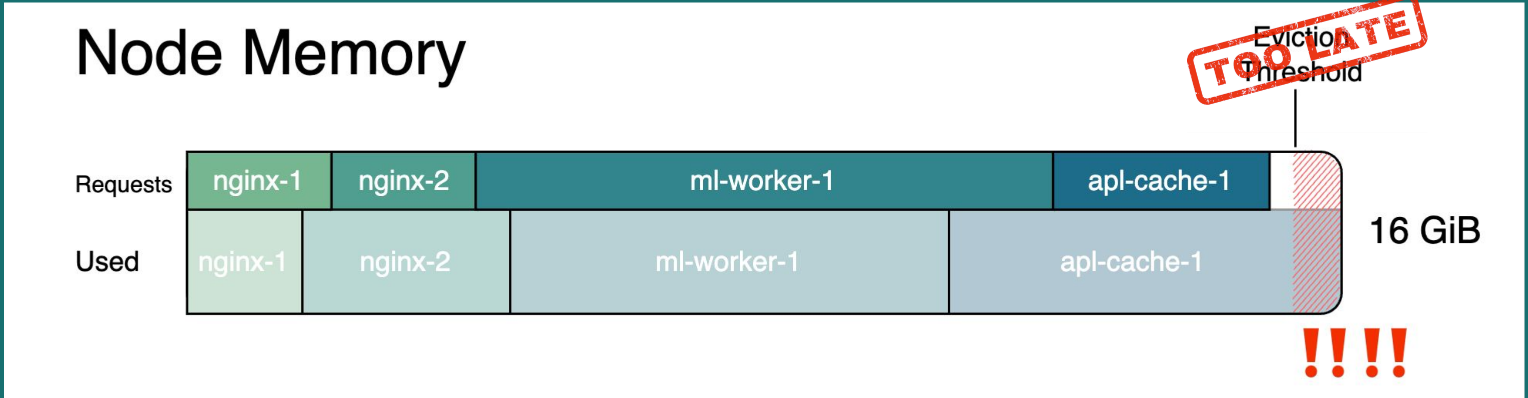
Node Memory Pressure and Eviction

- Node pressure occurs when certain signals exceed thresholds, such as `memory.available`
- Eviction currently applies only to incompressible resources like memory and disk.
- Kubelet will pick and evict pods that are using more resources than they requested. Evicted pods will be rescheduled, probably on other nodes.
- This **did not happen** in the lab (probably)



Node Out-of-Memory (OOM) Behavior

- Linux OOM-Killer will pick processes to terminate if the node runs out of memory (not Kubelet).
- OOM-Killer selection is influenced by Cgroup settings. Lower QoS classes and pods that are using a significant fraction of the memory on the node are towards the front of the line to be OOM-Killed.
- OOM-Killed containers will restart on the same node. They are not rescheduled.
- **This** is what happened in the lab (probably)

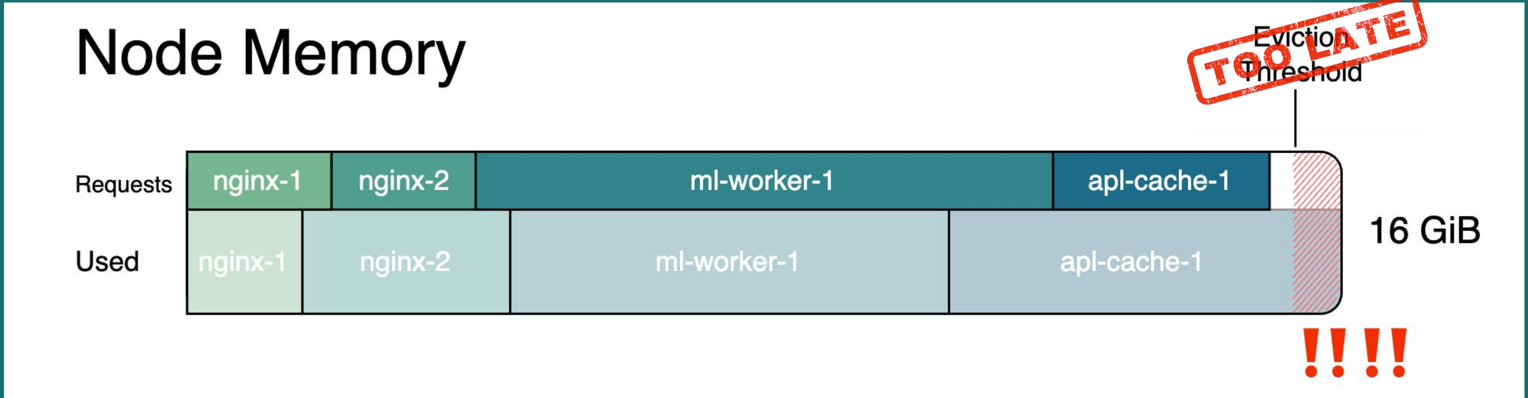


Wait… so, why didn't the Last State say OOMKilled?

Great question.
Some light reading.

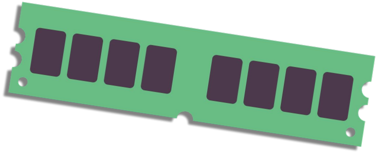
◂ Pod evictions, OOM scenarios and flows leading to them – Mihai Albert

Littledriver – Who murdered my lovely Prometheus container in K8s cluster? ▸



LESSONS LEARNED:

On The Importance of Memory Requests and Limits



Requests are very important.

- Memory isn’t guaranteed by requests*, but having proper requests sets the scheduler up for success when picking which nodes to co-locate workloads on
- You should be more conservative with any over-provisioning of memory, due to non-determinism of what happens when nodes run out of it

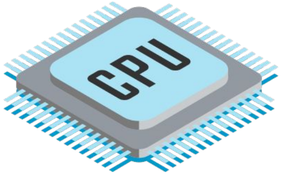
Limits are helpful too.

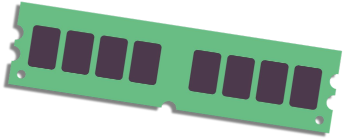
- Limits can help ensure that when a workload uses excessively more memory than it requests, that workload is what is OOMKilled, and not an innocent co-located workload

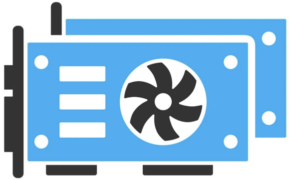
A Note on Cgroup Versions

What About Other Kinds of Resources?

- Ephemeral Storage works kinda like memory, but is often not specified
- Kubernetes permits extending the abstraction for additional resources using Device Plugins
    - GPU is the most commonly used additional resource
    - How overprovisioning works for other resources (i.e. GPUs) is outside the scope of this presentation
- Further control over CPU affinity can be achieved via a Kubelet setting, the static policy for the CPU Manager (more “exclusivity” than “sharing”).







Getting it Right in Practice

Getting it right is hard.

Typical Resource Management Journey

STAGE 1:

Don’t bother setting requests.

Falls down when performance problems become too frequent.

STAGE 2:

One-size fits all approach.

Falls apart when low resource efficiency becomes expensive.

STAGE 3:

Manually tune every workload.

Grueling or irregular; poor use of engineering resources?

OPTION 1:

Influence App Owners to Invest Their Time and Effort

Policies that can influence Developer / Application Owner Behavior

- Use **`LimitRanges`** to lay down resource defaults
- Use **`ResourceQuotas`** to create per-namespace scarcity
- Use Kyverno to define and enforce your own policy requirements

Tools to enable them to be as successful as they can be

- Application Monitoring (APM) Tools and dashboards to show app owners their workloads, and their real-world resource consumption
- Documentation or protocols that tell them what to do with that information

Typical Resource Management Journey

STAGE 1:

Don’t bother setting requests.

Falls down when performance problems become too frequent.

STAGE 2:

One-size fits all approach.

Falls apart when low resource efficiency becomes expensive.

STAGE 3:

Manually tune every workload.

Grueling or irregular; poor use of engineering resources?

Stage 4:

Automate it.

OPTION 2:

How About Automating it?

How Would Automation Work?

- Observe and collect usage data
- Calculate tailored resource settings for every workload
- Apply and keep settings up-to-date autonomously as requirements change over time
- Give human operators policy-level ownership, rather than specific settings ownership

THENEWSTACK

STOP Setting CPU and Memory Requests in K8s

Let Machine Learning and Automation do it for you.

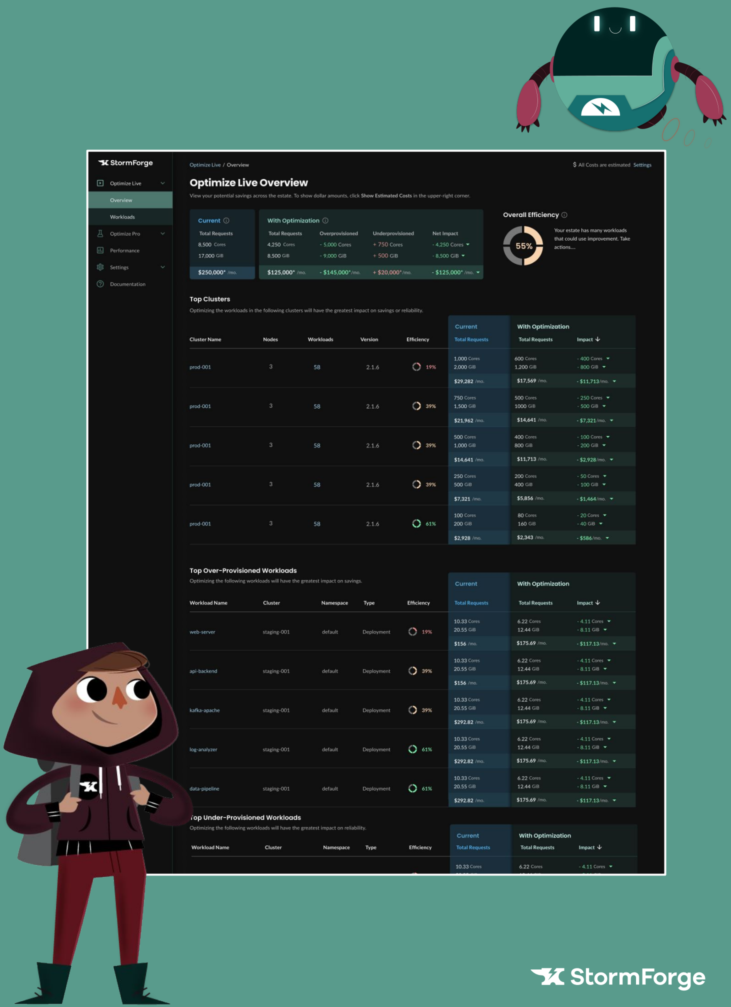
OPTIMIZE **LIVE**

Automatic and **Continuous** Workload Optimization for Kubernetes **At Scale**

Free Trial: app.stormforge.io/signup

Stop Setting CPU and Memory Requests.

Let intelligent automation do it for you.



StormForge

LinkedIn

linkedin.com/in/reidmv

StormForge

OPTIMIZE **LIVE**

Stop Setting CPU and Memory Requests.

Let intelligent automation do it for you.

Q&A

Thank you

Reid Vandewiele
Product / UX / DevRel
reid@stormforge.io