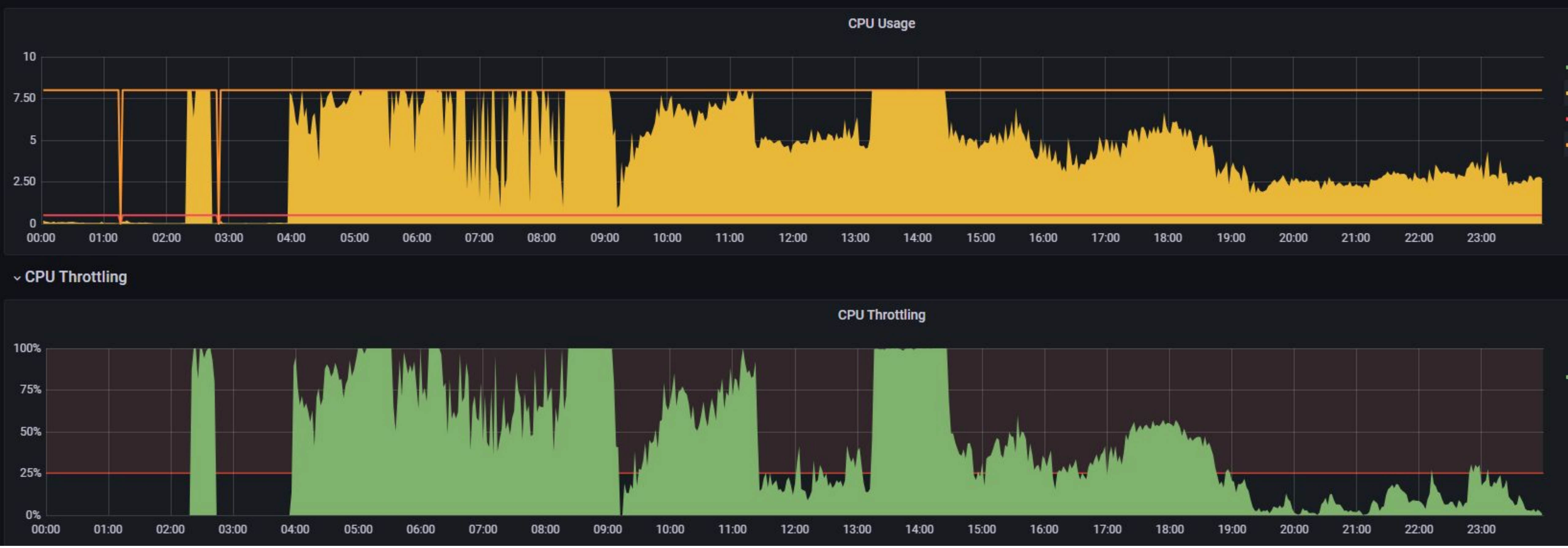


CPU Throttled?

- Do You like it..?
- If No, let's dig into it…

Kubernetes CPU Requests, Limits, and Throttling

Impacts on Application Performance, some Best Practices

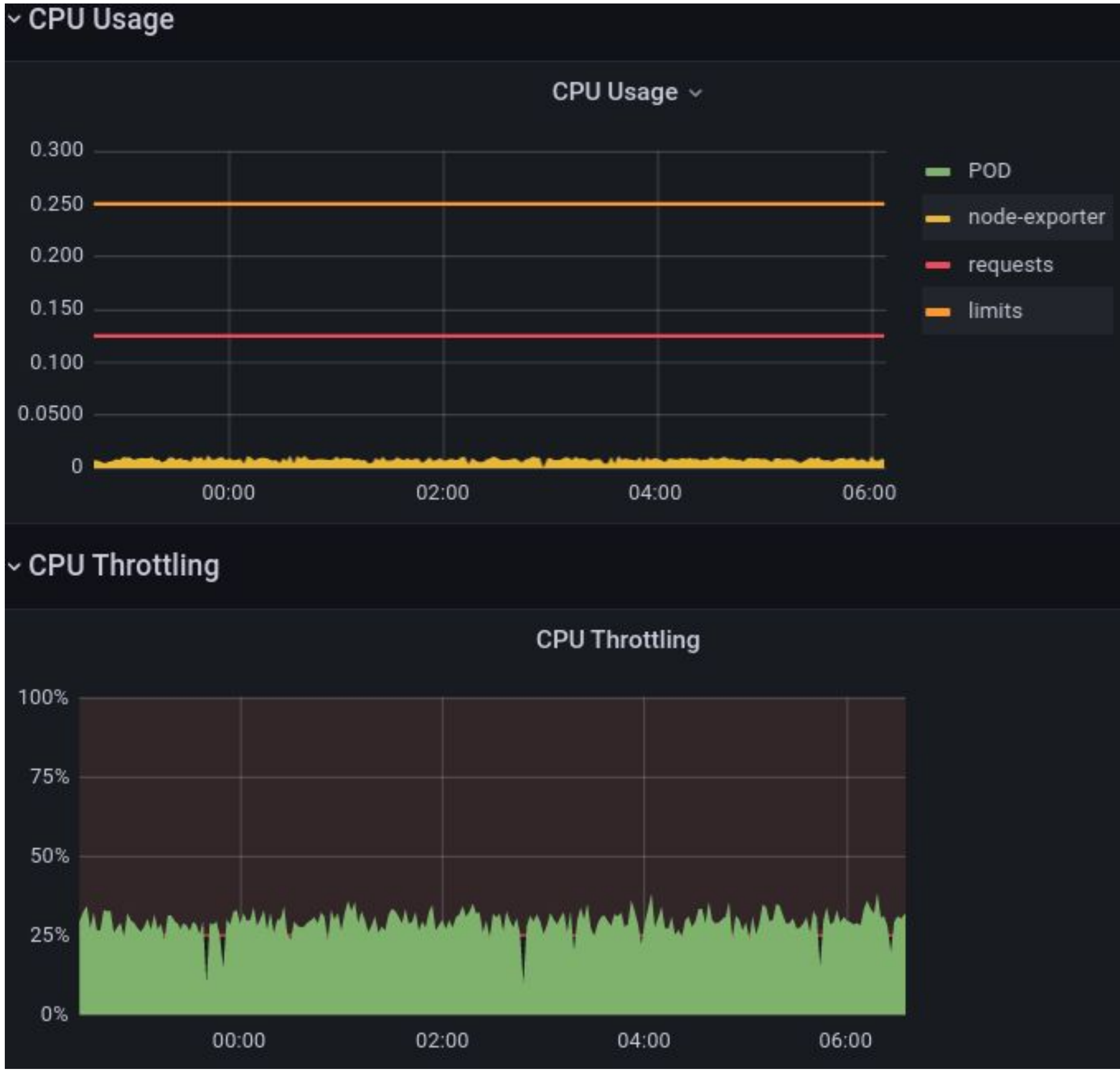
Alexandru Lazarev

Role

Date: 2025-01

CPU Usage vs CPU Throttling

- A random but real example: Grafana CPU Usage vs Throttling reports for a K8s container.
- How to explain low CPU Usage, even much below request, but high CPU Throttling?
  - … and this is commonly faced case.
  - What does those Throttling mean?
  - How is Throttling measured?
- All answers *(I'll try to address)* below…



Agenda

- Understanding CPU Requests and Limits
- Linux Kernel Scheduler (CFS) and CPU Bandwidth Control
- Throttling Mechanism: How and Why
- Impacts on Applications
- Best Practices and Community Insights
- Monitoring and Observability Tools
- Key Takeaways

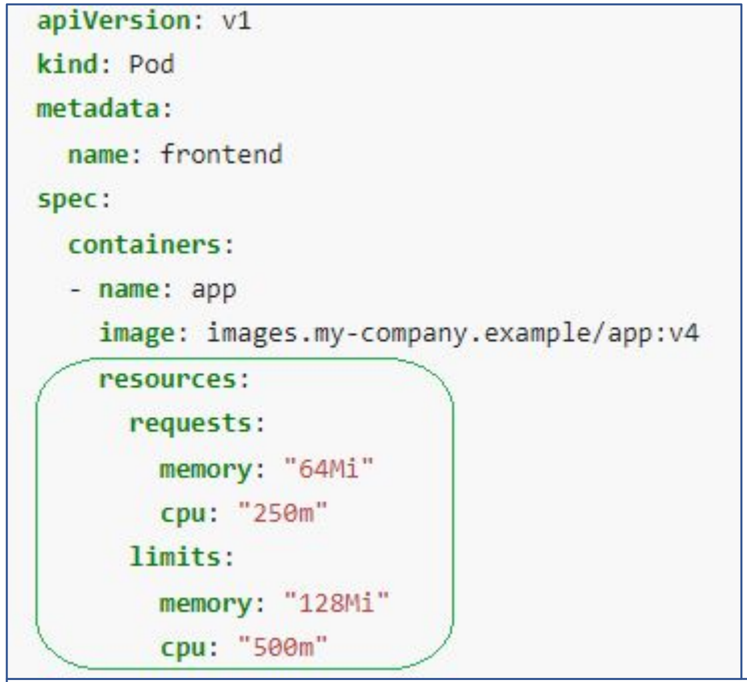
High Level Overview

CPU Management in Kubernetes: Limits, Requests, Throttling

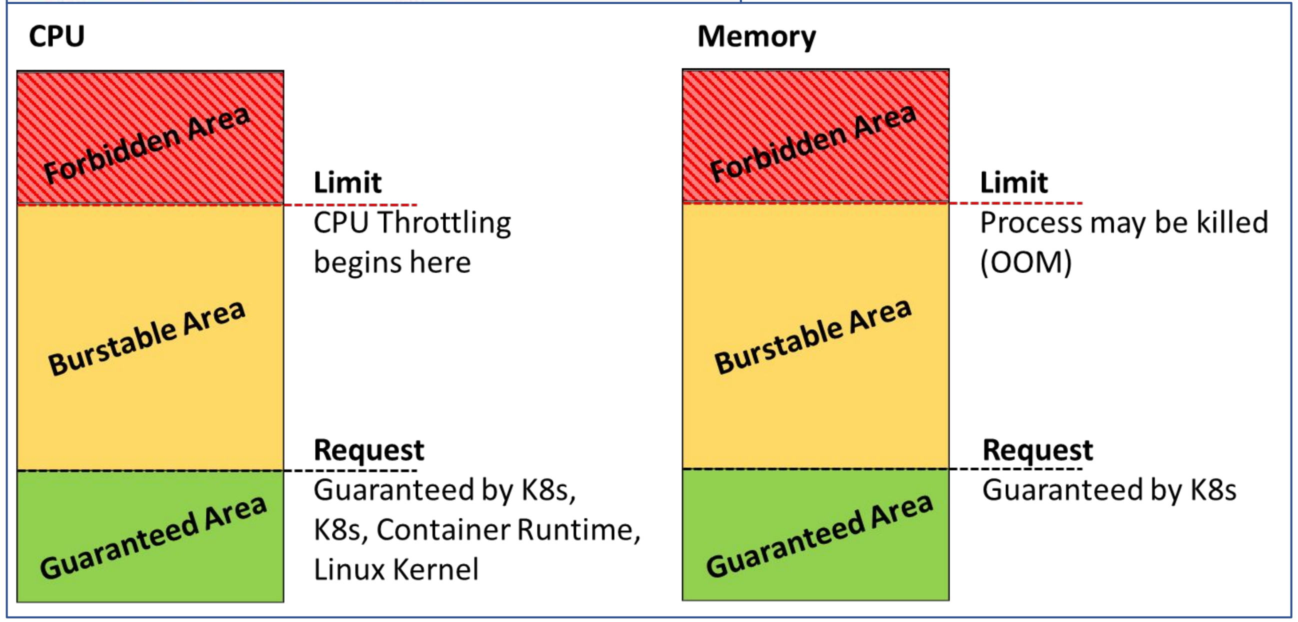
CPU Requests vs. Limits

- **Requests**: Guarantee resources for workloads.
    - In K8s, they affect how Pods are scheduled on nodes.
    - Play role as CPU tasks “weight” (priority) during CPU contention on host machine.

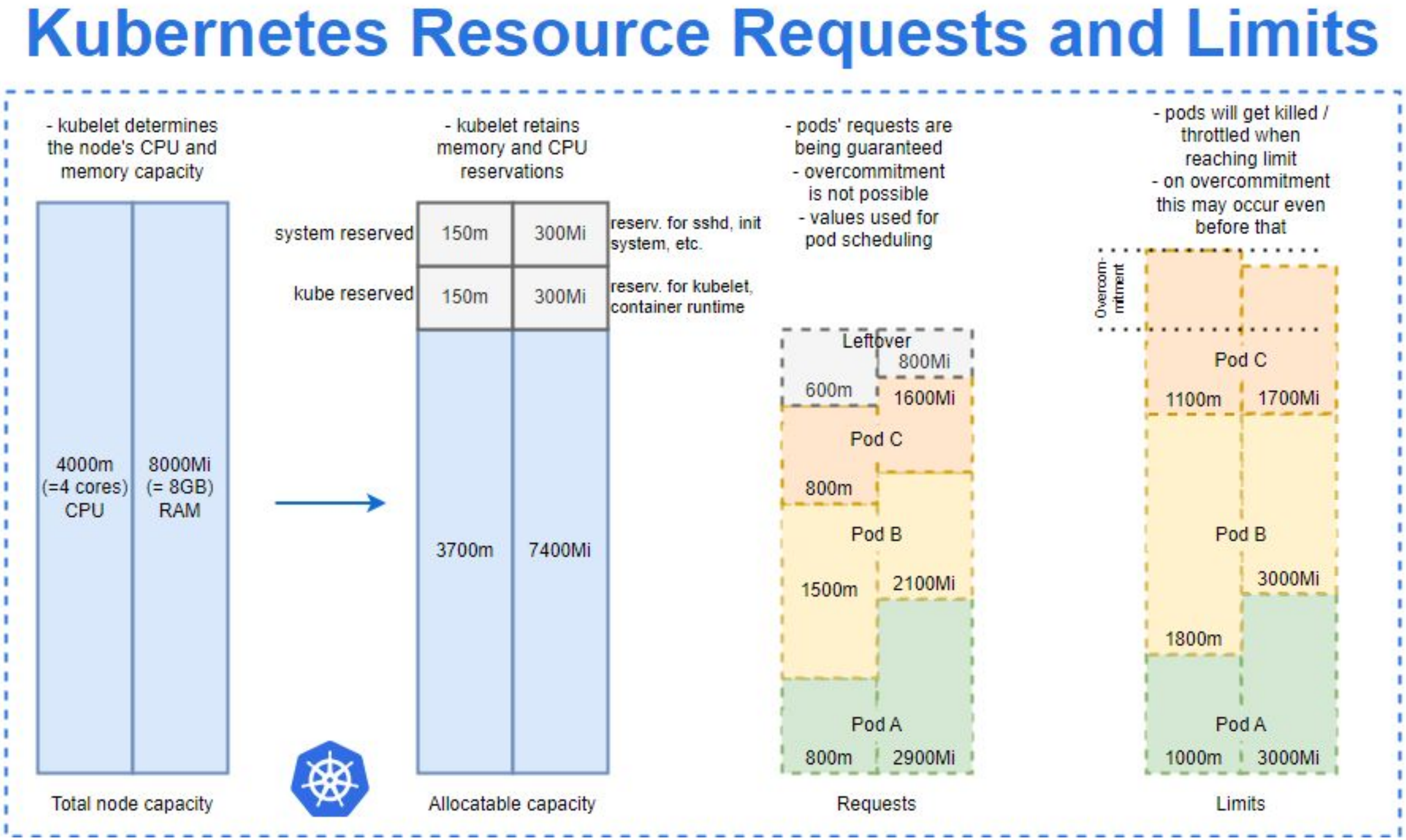
- **Limits**: Cap CPU usage to avoid over-consumption.



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```



K8s CPU and MEM Resources packed onto Node


Kubernetes Resource Requests and Limits
- kubelet determines the node's CPU and memory capacity
4000m (=4 cores) CPU
8000Mi (= 8GB) RAM
Total node capacity
- kubelet retains memory and CPU reservations
system reserved
150m
300Mi
reserv. for sshd, init system, etc.
kube reserved
150m
300Mi
reserv. for kubelet, container runtime
3700m
7400Mi
Allocatable capacity
- pods' requests are being guaranteed
- overcommitment is not possible
- values used for pod scheduling
Leftover
600m
800Mi
Pod C
800m
1600Mi
Pod B
1500m
2100Mi
Pod A
800m
2900Mi
Requests
- pods will get killed / throttled when reaching limit
- on overcommitment this may occur even before that
Overcom-mitment
Pod C
1100m
1700Mi
Pod B
1800m
3000Mi
Pod A
1000m
3000Mi
Limits

src: https://shipit.dev/posts/kubernetes-overview-diagrams.html

Key Terms and Concepts: CPU Throttling - Intro

- **CPU Throttling:**
  The process of **limiting CPU usage** when tasks or containers exceed their allocated quota (limit), forcing them to pause and wait until the next scheduling period.

Key Terms and Concepts: Intro

Further in this Doc:

- **CPU**: A unit of processing power, which can represent a **physical CPU**, a **CPU core**, or a **Logical CPU** (*e.g., a physical core with Hyper-Threading enabled is considered as 2 logical CPUs*). In Kubernetes, it's often referred to as a **vCPU**, equivalent to a share of processing power.

- **vCPU** (virtual CPU) in cloud and/or virtualized environments – *out of scope of this DOC* but, in short: is a unit of CPU capacity assigned to a container or pod in Kubernetes. It abstracts the underlying physical hardware, allowing Kubernetes to allocate and manage processing resources in a platform-agnostic way.
    - In most environments, 1 vCPU corresponds to **1 physical CPU** or **1 logical CPU** (e.g., 1 thread on a hyper-threaded core).
    - On physical hardware, a single physical CPU core with Hyper-Threading enabled may back **2 vCPUs**.
    - In virtualized environments (e.g., VMs or cloud instances), multiple vCPUs can be **over-committed** and mapped to the same physical CPU core. For instance:
        - **N:1 mapping**: 4 vCPUs could share 1 physical core, with each vCPU getting a fraction of the core's processing power (e.g., 25% if evenly divided).

Key Terms and Concepts: Intro

- **CPU Time:** The **total cumulative time** a task or group of tasks spends executing on one or more CPUs. This includes time spent running tasks **in parallel** (on multiple CPUs) or **concurrently** (on the same CPU) over a defined period.

- **CPU Usage:** Represents the **percentage of CPU capacity** consumed by a task or group of tasks over a monitored interval of wall-clock time. It’s a **rate** (e.g., "50% of one CPU" or "2 CPUs used at 80% i.e. 160%"), typically measured in real-time or averaged over intervals.

- **CPU Usage** and **CPU Time** are related, they are not identical.

Key Terms and Concepts: Intro

- **"wall-clock"** refers to **real-world time** as you would measure on a regular clock or stopwatch. It tracks the total elapsed time from start to finish, including everything (e.g., waiting, idle, and active time), not just CPU processing time.

Key Terms and Concepts: Intro

- **Context switching:** the process where the **Kernel Scheduler saves the state of a running task** (e.g., CPU registers, program counter) and **restores the state of another task**, allowing the CPU to switch between tasks efficiently. This enables multitasking by sharing CPU time among multiple processes or threads.

Key Terms and Concepts: Intro

Task Types in Relation to CPU Access

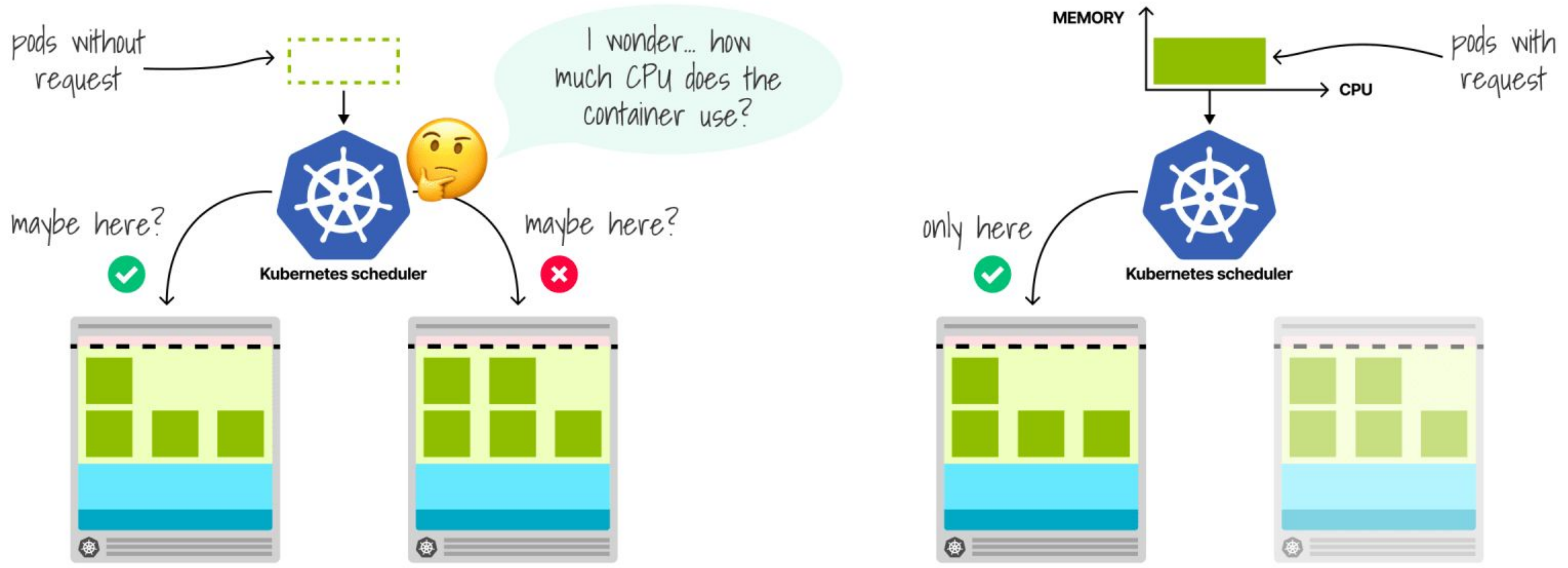
- **CPU-Bound Tasks:** Spend **most of their time on the CPU**, continuously running until their processing is complete. These tasks use the CPU intensively and benefit from more CPU time. The scheduler gives them longer CPU slices, but they are preempted if other tasks have higher priority.
- **I/O-Bound Tasks:** Spend **less time on the CPU** and frequently yield it to wait for I/O (disk, network). They use short bursts of CPU and quickly return to a waiting state, allowing the scheduler to switch to other tasks.
- **Memory-Bound Tasks:** Performance depends on memory access; they alternate between short CPU bursts and waiting for data from RAM, using the CPU sporadically.
- **Hybrid Tasks:** Some workloads exhibit a mix of characteristics (e.g., web servers handling both CPU-intensive request processing and I/O-heavy database access).

High Level Overview, again: credits

- This hi-level overview is based on following sources by Daniele Polencic from learnk8s.io:
  - https://itnext.io/cpu-limits-and-requests-in-kubernetes-fa9d55948b7c
  - https://x.com/danielepolencic/status/1632717409051181056
  - https://x.com/danielepolencic/status/1863580670066647194

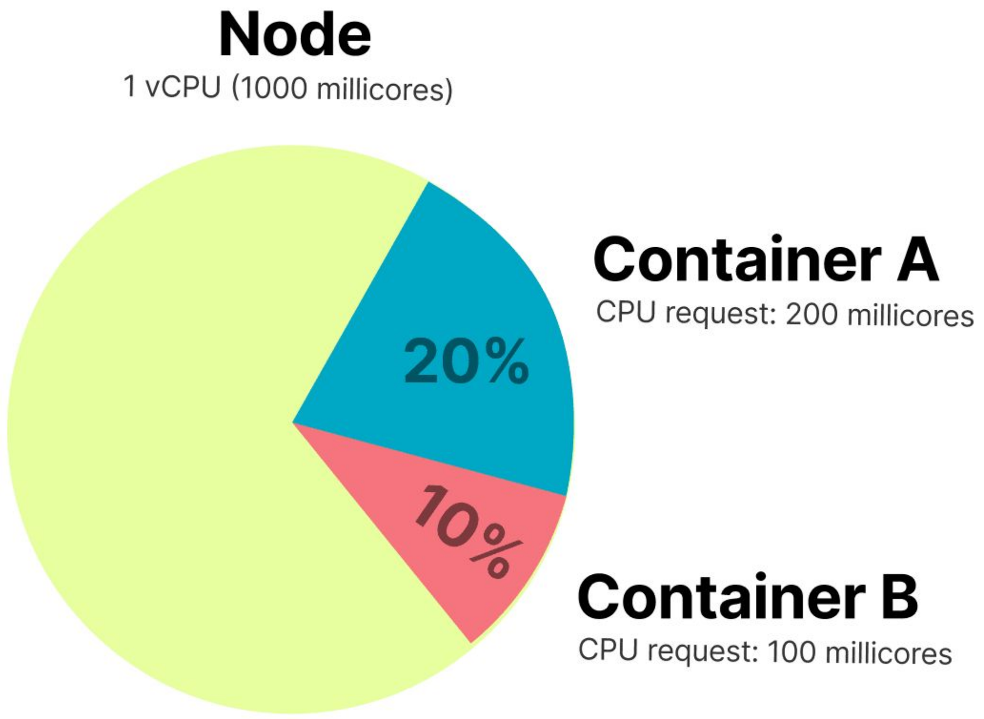
High Level Overview: CPU Requests vs K8s Scheduler

- Resources (like CPU, MEM) Requests hint K8s Scheduler where (on which node) to bind the pod.



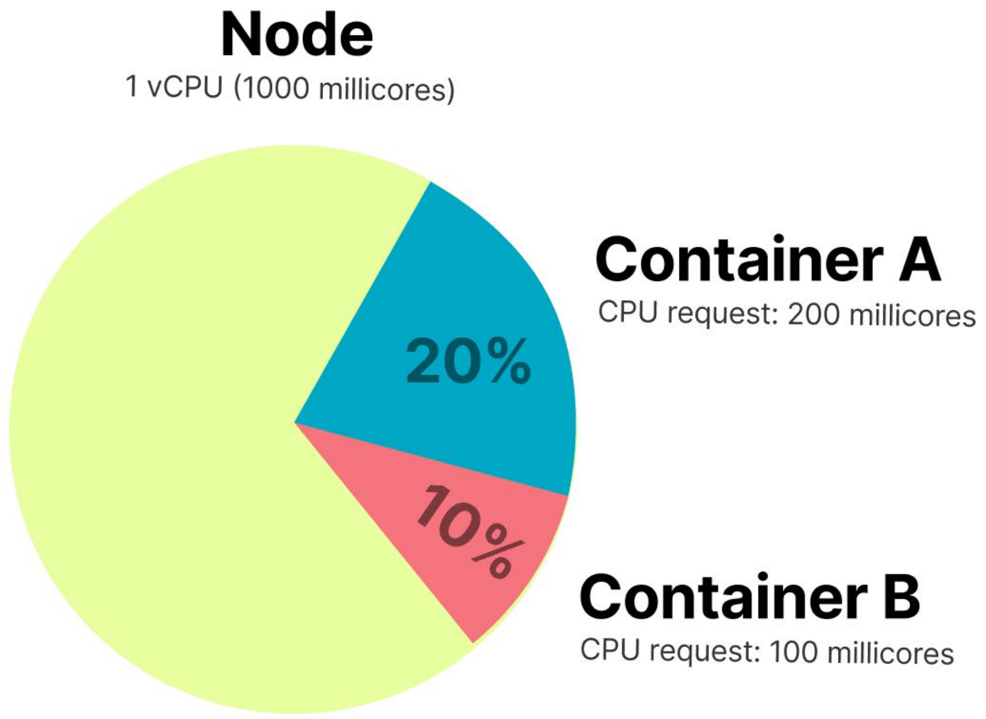
High Level Overview: CPU Requests vs Node

- Following image may be interpreted as CPU Requests booking on the node:
  - Container A booked 200m CPU or 20% of node Allocatable capacity,
  - Container B - 100m CPU or 10%,
  - For other containers remain 700m CPU or 70% of node Allocatable CPU resource.



High Level Overview: CPU Requests vs Usage

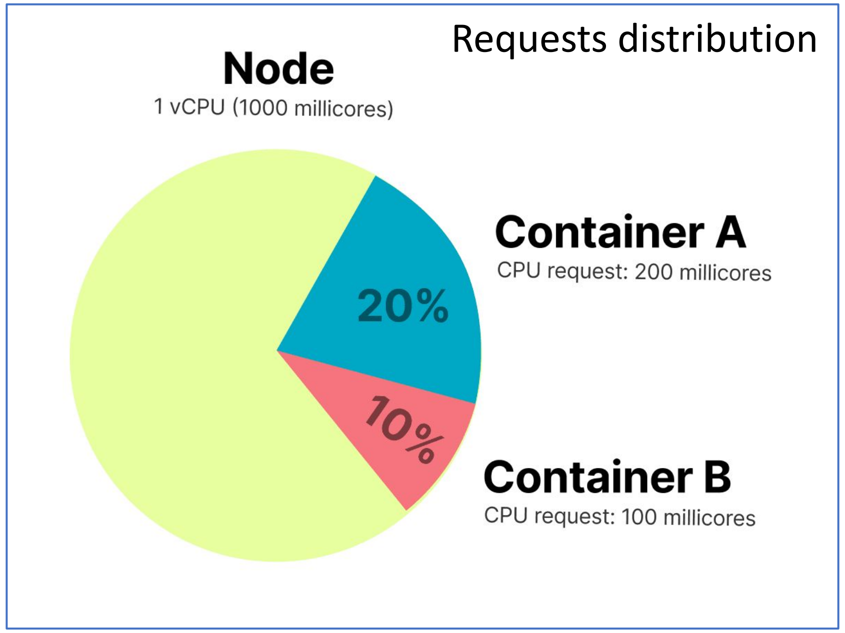
- Let interpret pie slices like Real CPU Usage:
  - Container A and B use exactly amount of CPU they requested.
  - What if both wants all the node CPU at the same time?

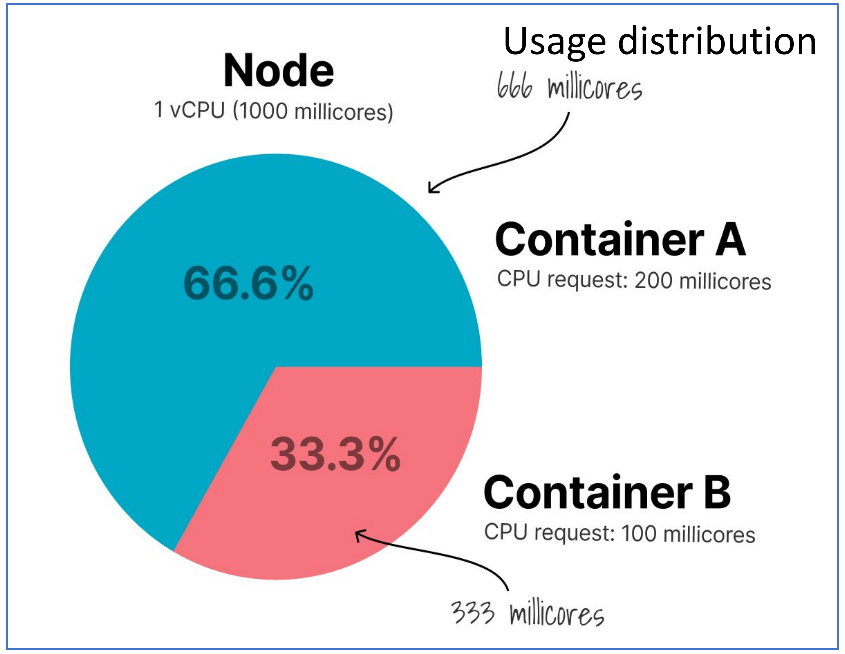


High Level Overview: CPU Requests vs More Usage

Both containers wants all the node CPU at the same time:

- Containers will compete for the same CPU but
- K8s will distribute available CPU to them proportionally by their requests (weight).

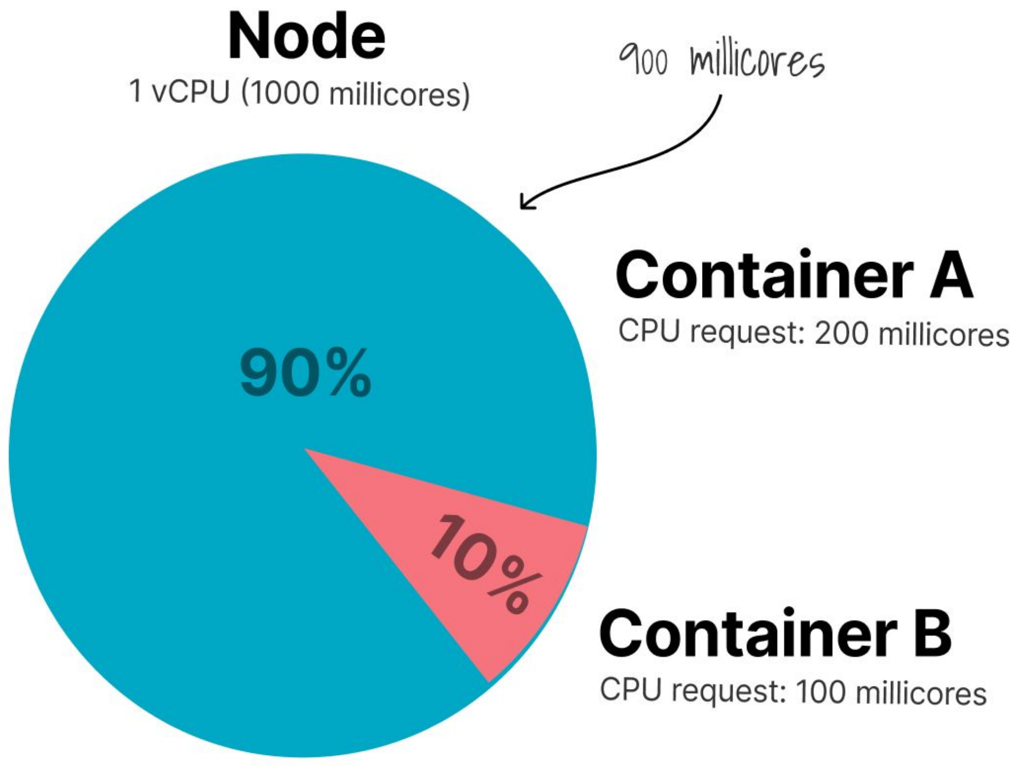


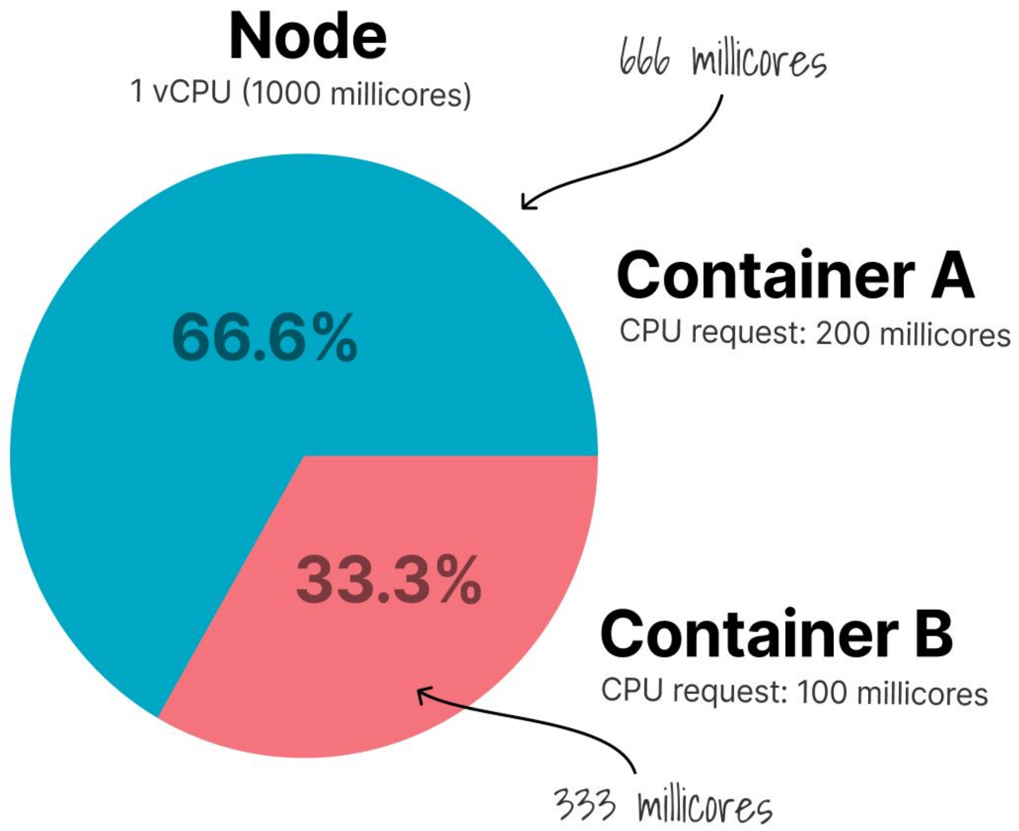


High Level Overview: CPU Requests vs more++ Usage

Use case 1: **A** is bursting but **B** is needing now only 100m CPU, so A can use the whole remaining spare CPU, until…

Use case 2: **B** is also bursting, so K8s will distribute all available CPU sparely and proportionally to their requests between them…

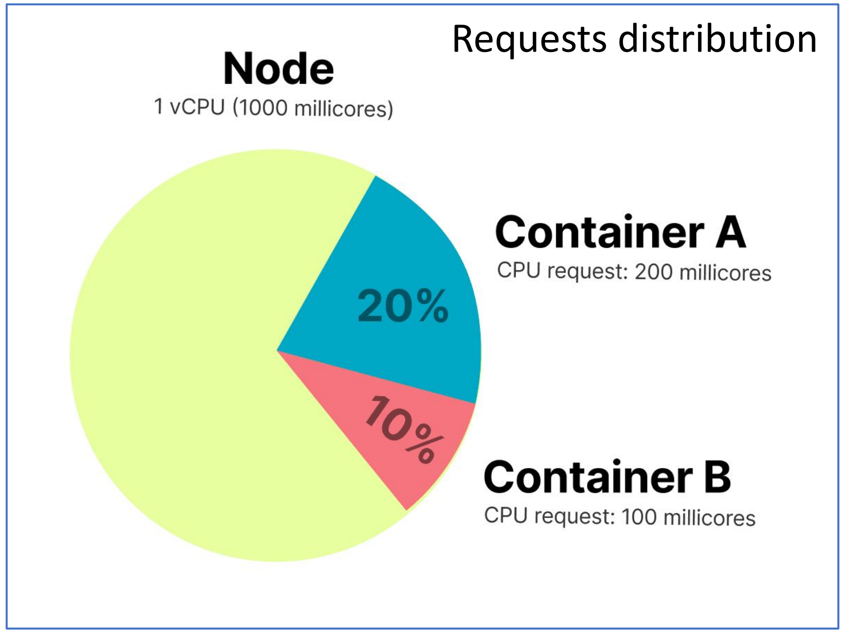


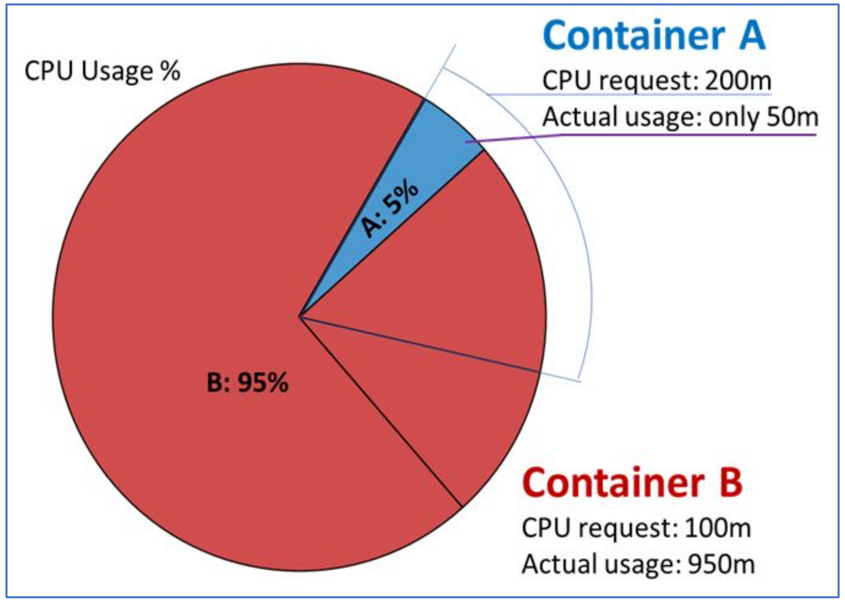


High Level Overview: CPU Requests vs More++ Usage

Other use-case: “**borrowing**”

- Container **A** requested 200m (20% of CPU capacity) but it is using now only 50m (5%).
- Container **B**, requested 100m (10%), but it is bursting now and need all CPU, so it may and is using now what is available: 95%
    - It uses 10% of it’s request + 70% of unused and unreserved + 15% borrowed from A which is reserved/requested by A but not used now, so 95%.

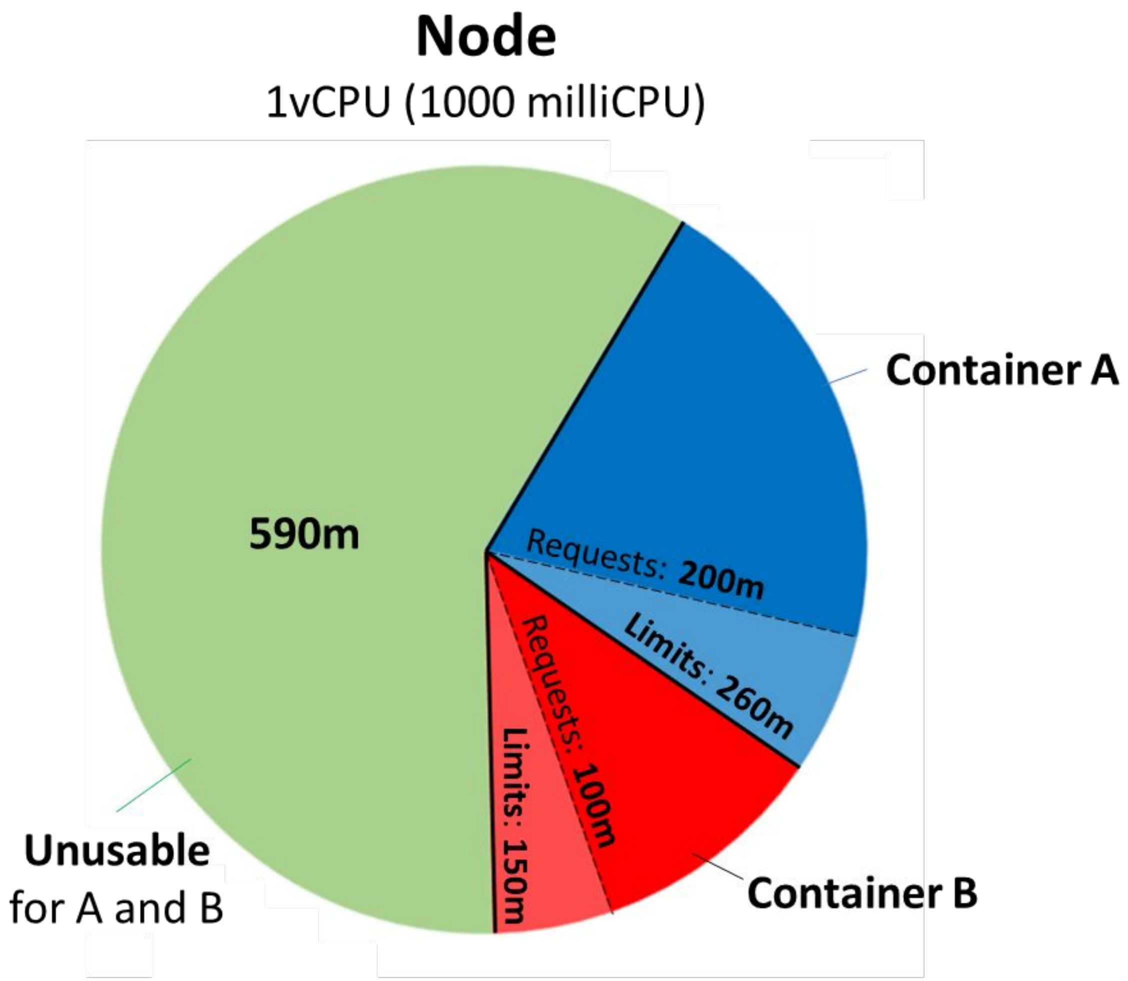




High Level Overview: CPU Requests vs Limits

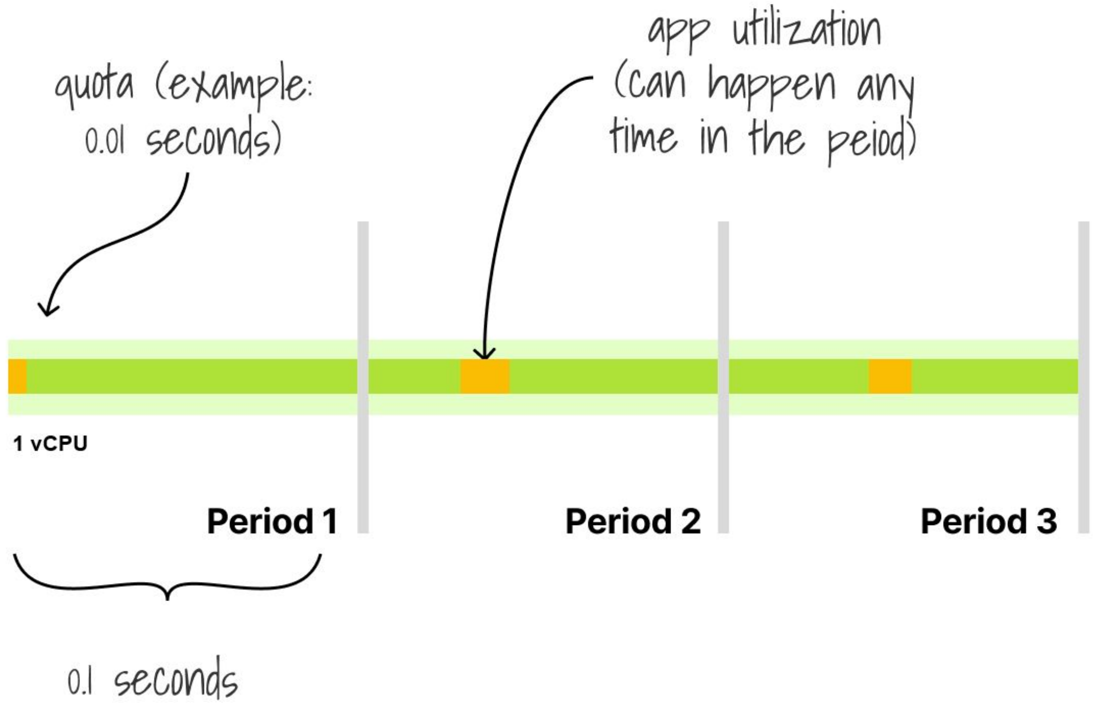
Let introduce limits for **A** and **B**:

- **A** CPU: Requests = 200m, Limits = 260m
- **B** CPU: Requests = 100m, Limits = 150m
- Containers may burst above their requests, but no more then defined limits, beyond which they will be CPU throttled, even if there is a plenty of idle CPU available.
- Usage slices above requests up to limits are possible only if there are available unused CPU resources on the node.



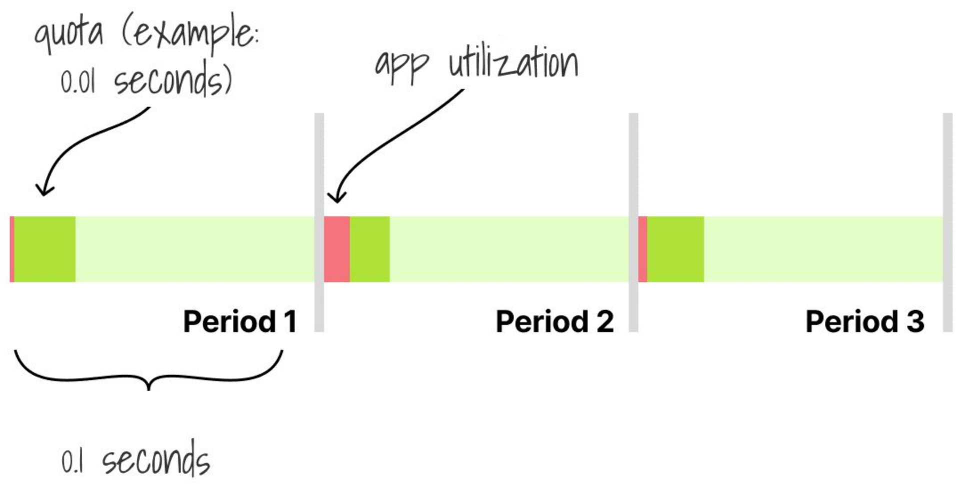
High Level Overview: CPU Limits

- <u>**CPU limits**</u> **translate into a time quota** for CPU usage within a defined accounting period.
- CPU Limits = 100m = 0.1 CPU may be translated as:
    - Use 10% of a single CPU capacity/time during a period, or
    - Use as much CPU as needed, but only for 10% of the time within a period.
- Example:
    - Use the CPU only for 1 second every 10s, or, ***closer to reality***:
    - Use the CPU for 10ms in every 100ms period


quota (example: 0.01 seconds)
app utilization (can happen any time in the peiod)
1 vCPU
Period 1
Period 2
Period 3
0.1 seconds

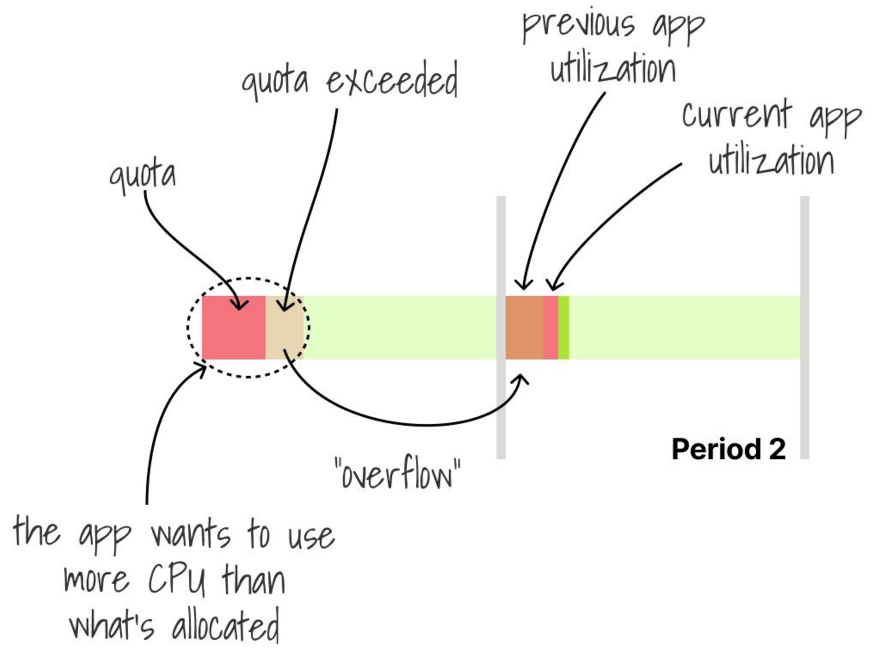
High Level Overview: CPU Limits vs Throttling

1) Application has only small (below quota) CPU bursts each period:



2) Application has bigger CPU bursts each period, and CPU demand exceeds defined quota, so:

- CPU Time will be limited to quota
- Remained time to execute is transferred to next period.



High Level Overview: CPU - Conclusions

CPU Requests!

- **Always set CPU Requests** to realistic and production-like performance testing proven values.
- They should reflect normal (not minimal) CPU Usage of application, or even p99 of observed maximum, depending by case.

- They guarantee requested resources.
- They allow spare and proportional to requests resources usage when:
  - Is needed higher amount of resources then requested,
  - Resources contention on node,
  - Others…

High Level Overview: CPU - Conclusions

CPU Limits?

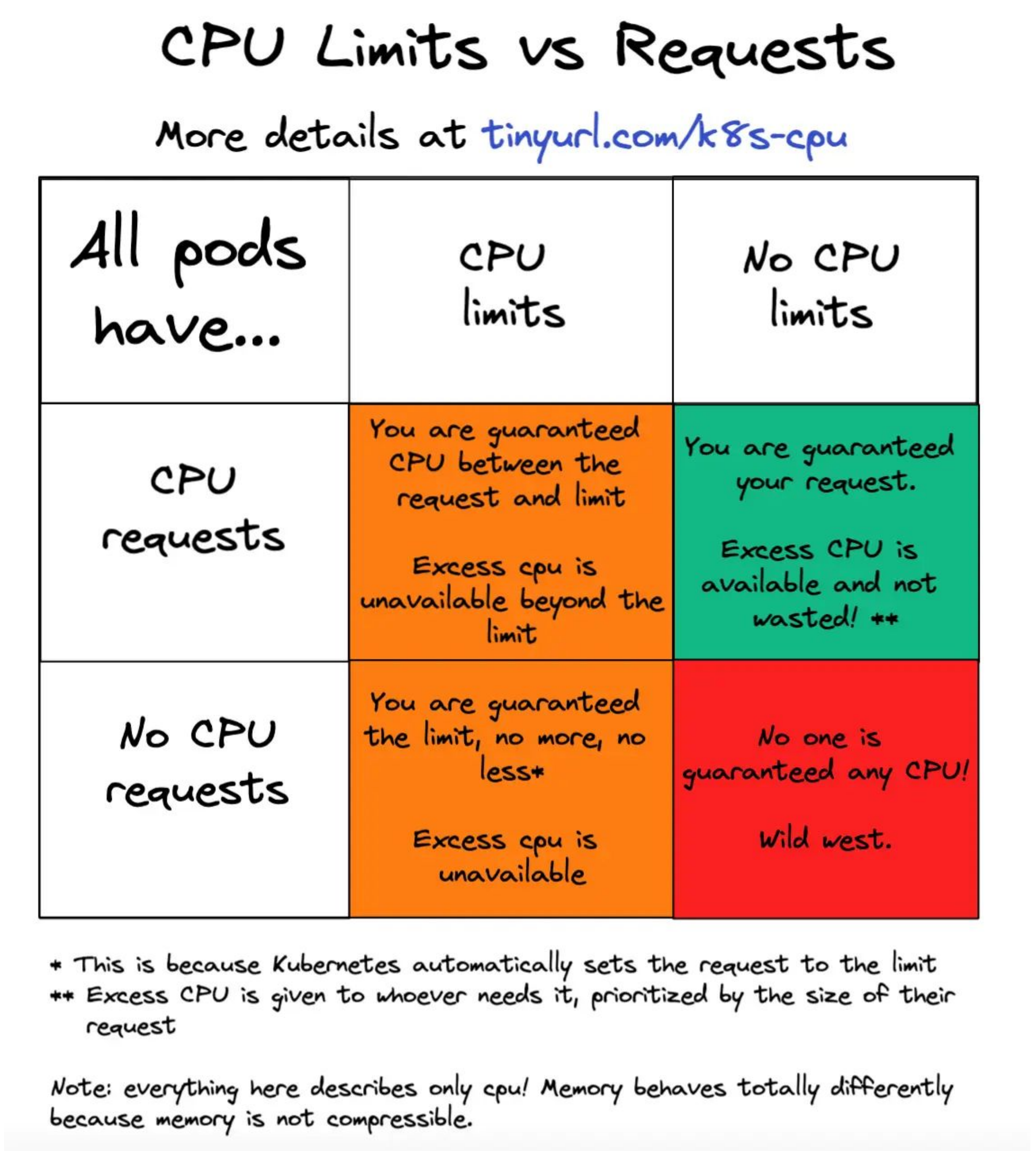
- This is tricky, controversial subject in community.
- They may cause application’s big, unexpected and unjustified delays, but they also may …
- Cap greedy CPU Usage of some buggy crazy noisy neighbor.

So…?

- Set them in function of use case.
- Setting them requires lots of careful, iterative performance tests!
- In production set them very high or remove at all for response time critical workloads, especially when there are lots of available CPUs.
- Set them higher (30-50% to 200-300%) then CPU Request for other cases or when you are enforced to set them.
- Set Requests = Limits for narrow use cases, like:
    - Performance Tests to determine workloads sizing and demands
    - When need “Guaranteed QoS” Pod classes
        - Especially Static CPU Binding use cases (narrow usage)
    - Others…

High Level Overview: In a picture

src: https://home.robusta.dev/blog/stop-using-cpu-limits



CPU Limits vs Requests

More details at tinyurl.com/k8s-cpu

All pods have...	CPU limits	No CPU limits
CPU requests	You are guaranteed CPU between the request and limit Excess cpu is unavailable beyond the limit	You are guaranteed your request. Excess CPU is available and not wasted! **
No CPU requests	You are guaranteed the limit, no more, no less* Excess cpu is unavailable	No one is guaranteed any CPU! Wild west.

* This is because Kubernetes automatically sets the request to the limit
** Excess CPU is given to whoever needs it, prioritized by the size of their request

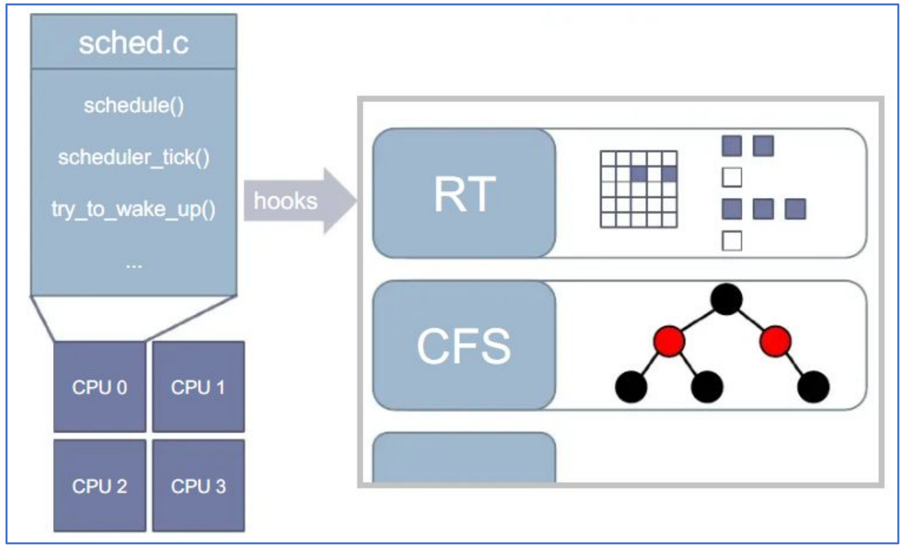
Note: everything here describes only cpu! Memory behaves totally differently because memory is not compressible.

Deep dive into implementation details

Linux Kernel, Tasks Scheduler, cgroups

Key Terms and Concepts: CPU Schedulers

- Kernel may manage simultaneously several tasks runqueues, each one with dedicated scheduler, for different schedulable types:
  - SCHED_NORMAL (also called SCHED_OTHER) for regular tasks – this is of interest in this demo. The rest are out of scope of this presentation, but just worth mentioning:
  - SCHED_RT – for Real-Time Tasks
  - SCHED_BATCH – long running lower priority batch tasks
  - SCHED_IDLE – even weaker
- CFS uses as runqueue an RB-Tree (Red-Black self-balancing tree) of CPU ready-to-run sched_entities (for now "entity" i.e., tasks and/or cgroups), ordered by their "vruntime"
  - "vruntime" is cumulative time spent by entity on CPU and adjusted by weight.
- CFS tries to give each queued entity fair amount of runtime on CPU by peaking left-most task from queue (with lowest runtime, but also considering other candidates priority)
- If peaked entity is a cgroup, CFS assigns CPU Time to the left-most real task within that cgroup, which has its own RB-Tree runqueue.



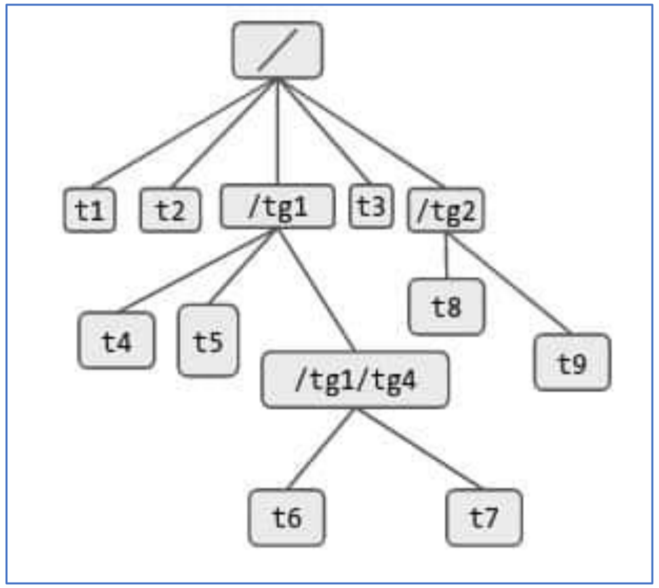


Fig 1. Tasks in a cgroup at different hierarchy levels.
src: https://blogs.oracle.com/linux/post/cfs-group-scheduling

Key Terms and Concepts: CFS

K8s enforces CPU Requests and Limits via **Linux CFS** and **cgroups**.

CFS or "Completely Fair Scheduler" is the CPU Tasks Scheduler, *currently used by Linux Kernel (* from v.2.6.23 (October 2007) but may change in Kernel v >= 6.6x)*.

CFS features:

- **Fair and Proportional Scheduling:** Distributes CPU time fairly to all CPU requesters, aiming to share available CPU cycles **proportionally** based on task weights (priority/niceness).
- **Spare Time Allocation:** Attempts to **assign any spare CPU time** to tasks that need it, ensuring efficient CPU utilization.
- **"sched_entity" Units:** Operates at the level of **sched_entity**, which can represent a **process, thread, or cgroup**.
- **Hierarchical Scheduling:** Treats **cgroups** as single sched_entities and **distributes CPU time internally** among tasks within each group.
- CFS maintains tasks runqueue (RB-Tree) per CPU Core.
- Doc: [CFS Scheduler — The Linux Kernel documentation](https://docs.kernel.org/scheduler/sched-design-CFS.html) (https://docs.kernel.org/scheduler/sched-design-CFS.html)

Key Terms and Concepts: cgroup

“**cgroup**” or Control Group:

- A **cgroup** is a Linux kernel feature that **groups tasks (processes/threads)** for **resource management** (CPU, memory, I/O).
- It allows **hierarchical control**, treating groups as single **sched_entities** under **CFS** for scheduling.
- Used by **Kubernetes** to enforce resource **requests and limits** at the container level.
- wiki: [cgroups – Wikipedia](https://en.wikipedia.org/wiki/Cgroups) (https://en.wikipedia.org/wiki/Cgroups)

Key Terms and Concepts: Cont’d

Task:

- A **task** in Linux is the **basic unit of scheduling**, representing either a **process** (single-threaded) or a **thread**.
- **Processes** are independent execution units with their own memory space, while **threads** share memory within the same process.
- **Tasks** are what the **CFS scheduler** manages and schedules on the CPU.

CFS sched_entity:

- Represents a **scheduling unit** in **CFS**—can be an **individual task** or an **entire cgroup**.
- Ensures **fair CPU sharing** based on vruntime *(accumulated and adjusted spent on CPU runtime)*, weight and priority.

Key Terms and Concepts: Cont’d

- **Noisy Neighbors:** tasks or workloads running on the same CPU or node that **demand high CPU resources simultaneously**, creating contention and competing with other tasks for CPU time, often causing performance degradation.

Key Terms and Concepts: CFS params

- **Targeted Latency (**`sched_latency_ns`**)** or **scheduling period:** the **ideal timeframe** within which the scheduler aims to allow all runnable tasks to execute at least once. It defines the period over which tasks are scheduled to ensure responsiveness.

- **Minimum Granularity (**`sched_min_granularity_ns`**):** the **smallest time slice** a task is permitted to run before being preempted. This prevents excessive context switching by ensuring each task runs for a minimum duration, enhancing system efficiency.

- **Task Time Slice:** the **allocated CPU time** for a task within a scheduling period, determined by dividing the targeted latency by the number of runnable tasks, ensuring fair CPU distribution. If the calculated time slice is below the minimum granularity, the scheduler enforces the minimum granularity to maintain system performance.

These parameters are crucial in the **Completely Fair Scheduler (CFS)** for balancing fairness and performance in task scheduling.

Linux **CFS CPU BWC (Bandwidth Control)**

- **CPU Limit** is enforced per cgroup by accounting **CPU Quota** per **CFS Period**.
- **CPU Quota** = CPU Limits = allowed up to use "CPU BW" (Bandwidth, or CPU Time) by a cgroup, during a **quota period**.
- CFS BWC **Quota Period**: wall-clock time = 100ms by default.

- So, to convert from K8s vCPUs Limits to CFS CPU Quota we use formula: ***vCPU * quota_period***,
  - where quota_interval = 100_000 microseconds by default or 100ms.

- **CPU BW** is accounted globally per cgroup (*in Kernel runtime structures*) as sum of all cgroup's tasks CPU Time spent on all available CPUs.

- If quota exceeded, then cgroup's tasks are throttled, i.e., delayed until next period – this is known as "**CPU Throttling**".
- Unused quota is not transferred/accumulated for next periods
  - *with exception of advanced [Burst feature](https://docs.kernel.org/scheduler/sched-bwc.html#burst-feature), not implemented by standard K8s for now (as of version 1.32) but are by some cloud providers (e.g., [Alibaba](https://www.alibabacloud.com/help/en/ack/ack-managed-and-ack-dedicated/user-guide/cpu-burst)).*
    - *Burstable feature:* https://docs.kernel.org/scheduler/sched-bwc.html#burst-feature
    - Alibaba Cloud implementation: https://www.alibabacloud.com/help/en/ack/ack-managed-and-ack-dedicated/user-guide/cpu-burst

Key Terms and Concepts: CPU Throttling - Recap

- **CPU Throttling (Kubernetes Workloads):**
  A process where Kubernetes and Container Runtimes **restricts a container's CPU usage** to stay within its assigned limits, often causing tasks to wait for the next scheduler quota period when they exceed their allowed CPU time.

- **CPU Throttling (Linux Kernel):**
  A mechanism in the **CFS Bandwidth Control** where the kernel **prevents a cgroup from exceeding its allocated CPU quota**, forcing tasks to stop running until the next quota period.

Linux CFS and cgroups v2

- **CPU Requests** are implemented via assigning priorities to containers' processes, i.e., assigning CPU weight and/or "nice" value to cgroups (files "cpu.weight" or "cpu.weight.nice")
  - Doc: https://docs.kernel.org/admin-guide/cgroup-v2.html#cgroup-v2-cpu
  - Known in community terms also as "***Soft Limit***".

- **CPU Limits** are implemented via "CFS Bandwidth Control" (CFS BWC) and cgroups "cpu.max" interface file.
  - CFS BWC Doc: https://docs.kernel.org/scheduler/sched-bwc.html
  - Known in community terms also as "**Hard Limit**".

Linux CFS and cgroups v2. Details, Cont’d

•Cgroup (v2) cpu config files:

- **cpu.max:** “$MAX $PERIOD”, default “max 100000”
  - “$MAX” is Quota = CPU Time in microseconds a cgroup’s task(s) can consume during a period of “$PERIOD” microseconds. $MAX=max -> allows unlimited CPU usage.
  - “max 100000” means the cgroup’s tasks can consume an **unlimited amount of CPU time** if spare CPU capacity is available, during a scheduling period of **100,000 microseconds (100ms).**

- **cpu.weight** – “$weight” – a number in range [1..10000], default = **100**.
  - Determines CPU competing “weight” vs other tasks.
  - If Task A having weight 200 competes for **<u>contended</u> CPU** with Task B having weight 100, then Task A will receive more of available CPU, in proportion of 2:1, than Task B.
  - It corelates with task niceness.
- **cpu.weight.nice** – “$nice” – a number in range [-19..20], default: 0
  - “nice”-ness is a kind of task priority – lower “nice” is higher priority.
  - This file is just another “nice” interface to “cpu.weight”
    - With ascending change of CPU weight from 1 to 10000, the niceness in changed descending from 20 to -19
      - CPU weight = 100 corresponds to nice = 0

CFS CPU BWC Quota Period: **Advanced Note**

- **$PERIOD** from file cpu.max ("*$MAX **$PERIOD***"), defaulted to 100_000usec, is configurable per cgroup in pure Linux administration, by (re-)writing value in cpu.max file directly.

- **In Kubernetes** it is possible to re-configure this value globally per cluster and/or per node, in **kubelet** settings: fields "cpuCFSQuota", "cpuCFSQuotaPeriod" and "CustomCPUCFSQuotaPeriod" feature gate.
  - https://kubernetes.io/docs/reference/config-api/kubelet-config.v1beta1/
  - https://kubernetes.io/docs/tasks/administer-cluster/kubelet-config-file/

- **DISCLAIMER:** Reconfiguring it in K8s is not common practice and is used only in specific narrow cases.

Linux CFS and cgroups v2. Details, Cont’d

cgroup v2 tips & trick:

- cgroup’s config and stats files are located on path:
  - “`/sys/fs/cgroup/$CG_HIERARCHY_PATH`”
- “`/$CG_HIERARCHY_PATH`” can be found in /proc filesystem, for specific Process ID (PID):

```
cat /proc/943567/cgroup
0::/kubepods.slice/kubepods-burstable.slice/kubepods-burstable-podc8015755_6b4f_4051_a184_f4ad9912b8bb.slice/cri-containerd-39e75eaa5b7c0457a58e912831b856254301bd92be88214a11a543f1865323ad.scop
e
```

  - where “0::” stands for unified hierarchy under “`/sys/fs/cgroup`” and the rest “`/kubepods.slice/…`” – from root cgroup to full hierarchical path.

CFS and cgroups v2. Details, Cont’d, Examples

Cgroup v2 “**cpu.max**” examples (applicable to containers too):

- **Limit = 1 CPU:** “100000 100000” i.e., cgroup may use 100_000 microseconds or 100 ms (milliseconds) during a CFS Quota interval of 100 ms, or MAX CPU Usage = 100%.
    - It may be interpreted as running fully on single CPU Core for entire CFS Period or e.g.,
    - Running, sequentially and/or parallelly, on multiple CPUs e.g., 50 millis on CPU0, 15 millis on CPU1 and 35 millis on CPU3
- **“400m” = 0.4 CPUs:** “40_000 100_000” i.e., only 40 millis (40%) of CPU Time during a Period (*it not necessarily be the same single CPU*).
- **“12345m” = 12.345 CPUs:** “1_234_500 100_000” may use up to ~12.3 CPUs (or more, if aggregated) simultaneously during 100 ms CFS Quota Period

CFS and cgroups v2. Details, Cont’d, Examples

- **CPU weight** (equivalent to task “nice”-ness) = 100 by default (or nice = 0) for regular Linux tasks (non-containerized e.g., running in root cgroups).
- **For containerized cgroups** it depends on **CPU Requests** and uses some implementation-dependent heuristics (e.g., on Kernel, Container Runtime, Container Orchestrator, etc.)

Examples on a K8s setup, where I’ve observed that 1 CPU ~= 40 weight units:

- **No CPU Requests (and no Limits):** weight=1 (nice=19).
  - It has the minimal weight so will have the lowest priority vs other regular or properly sized workloads when competing for CPU Time
- **0.5 CPUs:** weight= 20 (nice=7)
- **1 CPUs:** weight= 39 (nice=4)
- **2 CPUs:** weight= 79 (nice=1) *// closer to defaults (regular processes)*
- **3 CPUs:** weight=118 (nice=-1) *// higher CPU priority than other regular processes*

CFS, cgroups v2, throttling. Details, statistics.

- "**cpu.stat**" file reports CPU usage, throttling stats:
  - **usage_usec**: Total CPU time used by the cgroup (microseconds).
  - **user_usec**: Time spent in user space.
  - **system_usec**: Time spent in kernel space.
  - **nr_periods**: Number of periods cgroup threads were running.
  - **nr_throttled**: Number of periods where CPU usage was throttled.
  - **throttled_usec**: Total time the cgroup was throttled.
  - **nr_bursts** and **burst_usec**: related to Burst feature

- Widely adopted **CPU Throttling Formula** is:
  - $Throtlled\ \%\ (Periods) = 100\% * \frac{\Delta nr\_throttled}{\Delta nr\_periods}$
    - Δ - delta-time.
  - It is widely adopted by monitoring stacks like Prometheus + Grafana
    - Also known as Dave Chiluk's formula
  - It reflects throttling frequency, but not quantitatively throttled time

```
# cpu limit = 14 of 16 available

$ cat cpu.stat
usage_usec 648812623954
user_usec 590367341252
system_usec 58445282702
nr_periods 16356166
nr_throttled 2032
throttled_usec 2869459826
nr_bursts 0
burst_usec 0
```

CFS, cgroups v2, throttling. Alternative formula.

- “**cpu.stat**” file reports CPU usage, throttling stats:
  - **usage_usec**: Total CPU time used by the cgroup (microseconds).
  - **nr_periods**: Number of periods cgroup threads were running.
  - **nr_throttled**: Number of periods where CPU usage was throttled.
  - **throttled_usec**: Total time the cgroup was throttled.

```
# cpu.stat
usage_usec 648812623954

nr_periods 16356166
nr_throttled 2032
throttled_usec 2869459826
```

Alternative formula – Throttling Severity:

$$Throtlled\ \%\ (Time) = 100\% * \frac{\Delta throttled\_usec}{\Delta nr\_periods\ *\ quota\_period}$$

- where quota_period = 100ms, by default.
- It reflects throttling severity or i.e., percentage of time a cgroup spent throttled, due to quota exceeded, relative to its allowed CPU Time.

CFS, cgroups v2, throttling. Some heuristics:

- "**cpu.stat**" file reports CPU usage, throttling stats:
  - **usage_usec**: Total CPU time used by the cgroup (microseconds).
  - **nr_periods**: Number of periods cgroup threads were running.
  - **nr_throttled**: Number of periods where CPU usage was throttled.
  - **throttled_usec**: Total time the cgroup was throttled.

```
# cpu.stat
usage_usec 648812623954

nr_periods 16356166
nr_throttled 2032
throttled_usec 2869459826
```

Below formulas are experimental heuristics and provide very approximate KPIs on Throttling impact.

Their accuracy decreases in multi-threading parallel tasks execution use-cases.

Desired CPUs:

$$Desired\ CPUs = \frac{\Delta usage\_usec + \Delta throttled\_usec}{\Delta nr\_periods * quota\_period}$$

- Estimates the required CPU to avoid throttling by normalizing workload demand over the CFS quota period.

CPU Time Efficiency:

- $CPU\ Efficiency\ \%\ (Time) = 100\% * \frac{\Delta usage\_usec}{\Delta usage\_usec + \Delta throttled\_usec}$

- Estimates used CPU Time vs eventual demand (execution + throttled time)

Throttling Explained

Agenda

- How CPU limits lead to idle time in CFS periods i.e., **throttling**.
- Examples: Single-thread vs. Multi-thread workloads.
- Example#1: Pod/Container with CPU limit=0.4 and single-thread.
- Example#2: Pod/Container with CPU limit=various, single-thread and continuous computation.
- Example#3: Pod/Container with CPU limit=2 and multithreading.
- Conclusions

Throttling Explained: Examples

For simplicity, following examples consider that there are missing … or make abstraction of:

- context switching
- noisy neighbors
- CFS target timeout, minimum task runtime, etc.

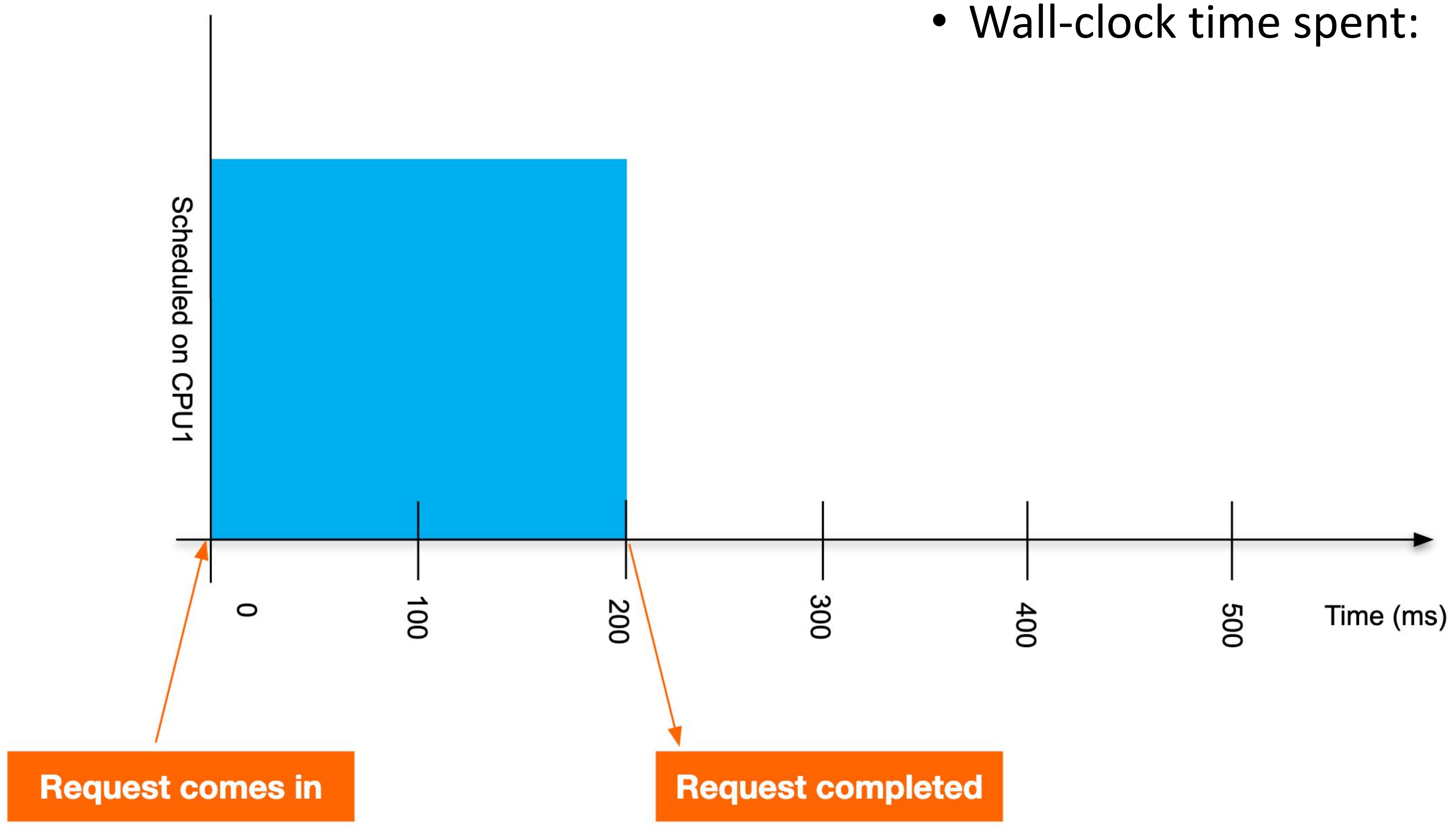
Throttling Explained. Example#1: Single-Task

- Suppose single task (single-threaded process), single CPU Core, no other “noisy neighbors” (other tasks requiring CPU parallelly) to compete with.
- A CPU-bound Task needing 200 ms of CPU Time to complete a request.
    - It may be an http/grpc request to a server and maybe important one like liveness probe or real-time request, or Garbage Collector thread in JVM, or data processing request between main DB and replica, etc.

- ***©NOTE:*** This example explanations (images and others) are credited to **Dave Chiluk** - Linux Kernel contributor and following sources:
    - “CPU Limits. Throttling” KubeCon presentation: https://www.youtube.com/watch?v=UE7QX98-kO0.
        - Related presentation: https://static.sched.com/hosted_files/kccncna19/dd/Kubecon_Throttling.pdf
    - Blog posts:
        - P1: CPU Throttling - Unthrottled: Fixing CPU Limits in the Cloud (or medium link)
            - Link: https://engineering.indeedblog.com/blog/2019/12/unthrottled-fixing-cpu-limits-in-the-cloud/
            - Medium: https://medium.com/indeed-engineering/unthrottled-fixing-cpu-limits-in-the-cloud-a0995ede8e89

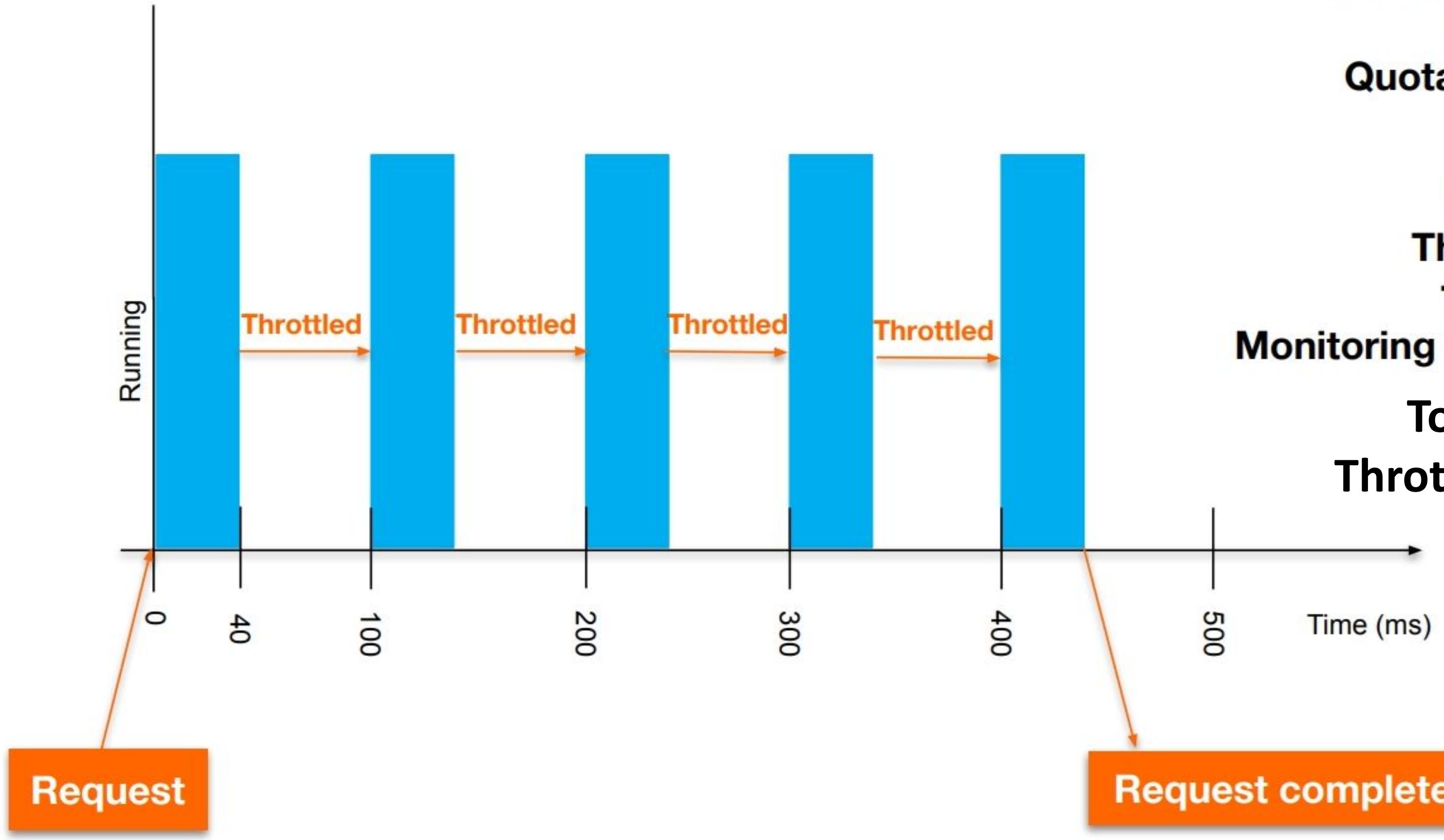
Throttling Explained. Example#1
Conceptual model: No CPU Limit

- CPU time Required: 200ms
- Wall-clock time spent: 200ms



Throttling Explained. Example#1

Conceptual Model: CFS Bandwidth Control



CPU time required	200ms
CPU quota	.4 CPU
Quota per period	40ms
CPU Usage	200ms
Throttle Time	240ms
Throttled %	80%
Monitoring CPU Usage	.4 CPU
Total periods	5
Throttled period	4

Throttling Explained. Example#1. Calculus

Calculus page format (as on next page, incl. for other examples):

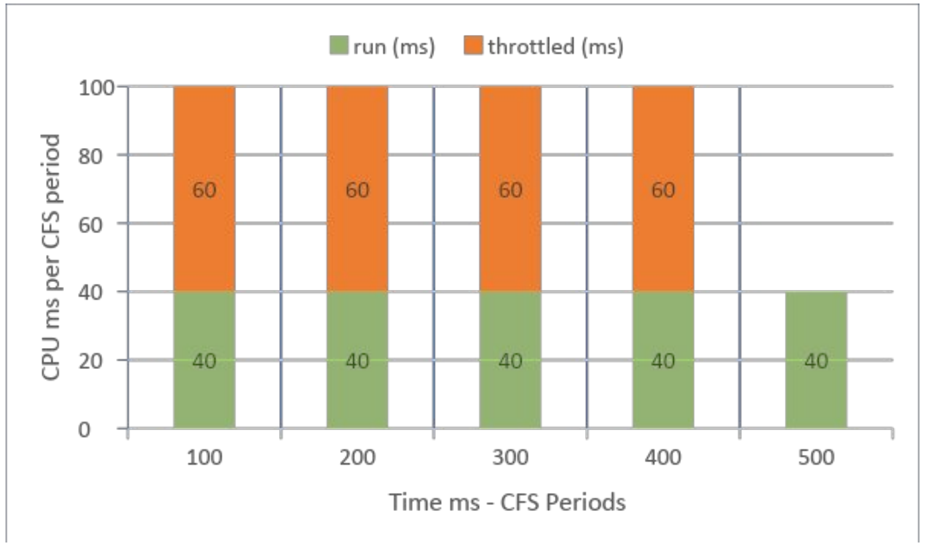
- Try various calculus, using mentioned above formulas and heuristics,
  - somewhere convert “usec” (microseconds) to “ms” (millis).
- Put in the same page simulated:
  - Timetable vs CPU(s) Usage per period and data-chart.
  - Simulated cpu.stat

Throttling Explained. Same Example#1. Calculus

Time	run (ms)	throttled (ms)
100	40	60
200	40	60
300	40	60
400	40	60
500	40	

```
# cpu.max
40_000 100_000

# cpu.stat
usage_usec 200_000
nr_periods 5
nr_throttled 4
throttled_usec 240_000
```



- $Throttled\ \%\ (Periods) = 100\% * \frac{\Delta nr\_throttled}{\Delta nr\_periods} = 100\% * \frac{4}{5} = 80\%$

- $Throttled\ \%\ (Time) = 100\% * \frac{\Delta throttled\_ms}{\Delta nr\_periods * quota\_period} = 100\% * \frac{240}{5 * 100} = 48\%$

- $CPU\ Efficiency\ \%\ (Time) = 100\% * \frac{\Delta usage\_ms}{\Delta usage\_ms + \Delta throttled\_ms} = 100\% * \frac{200}{200+240} = 45\%$

- $Desired\ CPUs = \frac{\Delta usage\_ms + \Delta throttled\_ms}{\Delta nr\_periods * quota\_period} = \frac{200+240}{5*100} = \frac{440}{500} = 0.88$

Throttling Explained. Example#1. Conclusions

- CPU Limit caused high throttling frequency, high-latency and inefficient CPU Time usage.
- CPU-bound request finished in 440ms instead of 200ms.
  - Even if overall CPU Usage still it 200ms but 240ms were lost due throttling, even if there were available spare CPU Time.
- Recommendations: Increase CPU Limit until Throttled KPIs tend to zero, or… maybe remove limits at all?

Throttling Explained. Example#2

- One CPU-Bound Task continuously requiring full CPU (long or infinite computation loop).
    - Single process, single thread, single CPU, no noisy neighbors.
- Monitored interval = 1 sec, sampling each 100ms CFS Period.

Use-cases variations:

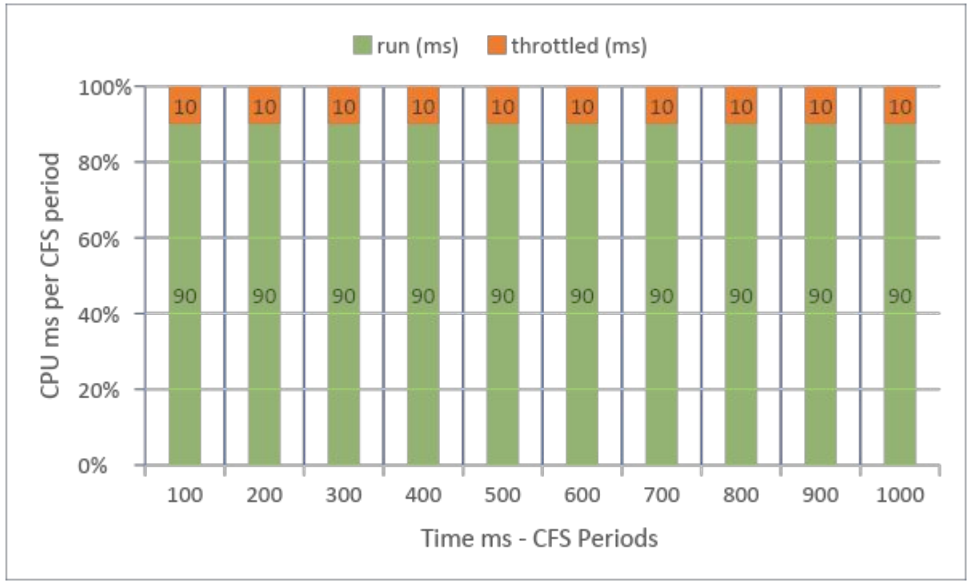
- Sub Example 2.1: CPU Limit = 0.9
- Sub Example 2.2: CPU Limit = 0.1

Throttling Explained. Example#2.1: Calculus

Time	run (ms)	throttled (ms)
100	90	10
200	90	10
300	90	10
400	90	10
500	90	10
600	90	10
700	90	10
800	90	10
900	90	10
1000	90	10

```
#cpu.max
90_000 100_000

# cpu.stat
usage_usec 900_000
nr_periods 10
nr_throttled 10
throttled_usec 100_000
```



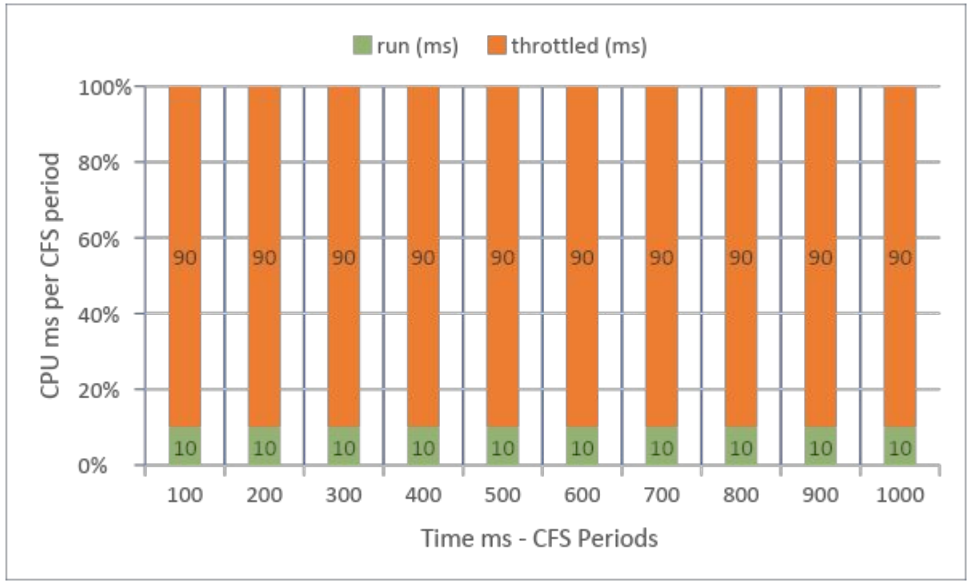
- $Throttled\ \%\ (Periods) = 100\% * \frac{\Delta nr\_throttled}{\Delta nr\_periods} = 100\% * \frac{10}{10} = 100\%$
- $Throttled\ \%\ (Time) = 100\% * \frac{\Delta throttled\_ms}{\Delta nr\_periods * quota\_period} = 100\% * \frac{100}{10 * 100} = 10\%$
- $CPU\ Efficiency\ \%\ (Time) = 100\% * \frac{\Delta usage\_ms}{\Delta usage\_ms + \Delta throttled\_ms} = 100\% * \frac{900}{900+100} = 90\%$
- $Desired\ CPUs = \frac{\Delta usage\_ms + \Delta throttled\_ms}{\Delta nr\_periods * quota\_period} = \frac{900+100}{10*100} = \frac{1000}{1000} = 1$

Throttling Explained. Example#2.2: Calculus

Time	run (ms)	throttled (ms)
100	10	90
200	10	90
300	10	90
400	10	90
500	10	90
600	10	90
700	10	90
800	10	90
900	10	90
1000	10	90

```
#cpu.max
10_000 100_000

# cpu.stat
usage_usec 100_000
nr_periods 10
nr_throttled 10
throttled_usec 900_000
```



- $Throttled\ \%\ (Periods) = 100\% * \frac{\Delta nr\_throttled}{\Delta nr\_periods} = 100\% * \frac{10}{10} = 100\%$

- $Throttled\ \%\ (Time) = 100\% * \frac{\Delta throttled\_ms}{\Delta nr\_periods * quota\_period} = 100\% * \frac{900}{10 * 100} = 90\%$

- $CPU\ Efficiency\ \%\ (Time) = 100\% * \frac{\Delta usage\_ms}{\Delta usage\_ms + \Delta throttled\_ms} = 100\% * \frac{100}{100+900} = 10\%$

- $Desired\ CPUs = \frac{\Delta usage\_ms + \Delta throttled\_ms}{\Delta nr\_periods * quota\_period} = \frac{100+900}{10*100} = \frac{1000}{1000} = 1$

Throttling Explained. Example#2. Conclusions

“These two types of throttling are the same—but oh, they're two big differences!”

- In both use-cases Grafana will show 100% in the standard Throttling (%) report, because it’s period-based, despite that CPU Limits and usage shapes are very different.
    - Maybe there are needed additional Throttling metrics in Grafana?

Throttling Explained. Example#3: Multi-Tasks

- Suppose Multi-Tasks workload(*1), Multiple CPUs, no other noisy neighbors to compete with.
    - 1) It may be single application process with multiple threads (Java, .NET, etc.), or
    - a container/cgroup with multiple single-threaded processes (e.g., PostgreSQL), or
    - mix of above.

- ***$N*** parallel CPU-bound Tasks, each needing ***$M*** ms of CPU Time to complete a request.
    - Each may be an http/grpc request to a server and maybe important one like liveness probe or real-time request, or Garbage Collector thread in JVM, or data serving request between main DB and replica, etc.

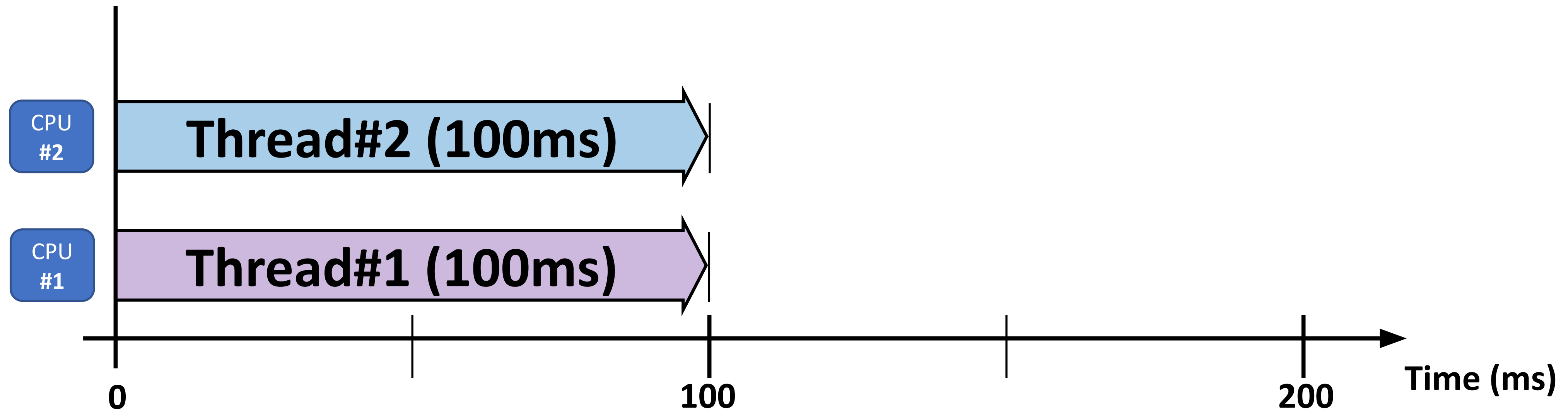
Throttling Explained. Example#3: Multi-Tasks

- Use-case#**3.1**: 2 parallel tasks started at the same time and each requiring 100ms CPU Time. They may (and will, in these examples) run parallelly on different CPUs.
    - 3.1.1: No CPU Limits
    - 3.1.2: CPU Limit = 2

- Use case#**3.2**: 10 parallel Tasks started simultaneously, each needing 50 ms of CPU Time to complete. Being executed on different CPUs.
    - 3.2.1: No CPU Limits
    - 3.2.2: CPU Limit = 2

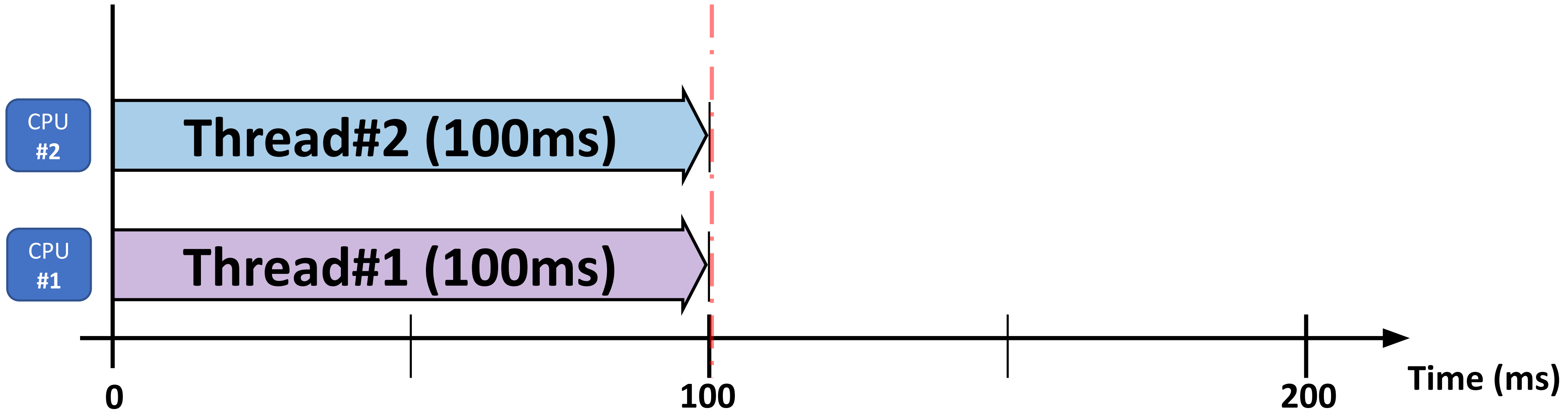
Throttling Explained. Example#3.1.1: Multi-Tasks

- Use-case#3.1.1: 2 parallel tasks requiring 100ms CPU Time, each one running on different CPU.
- No CPU Limits.
- Everything is OK.

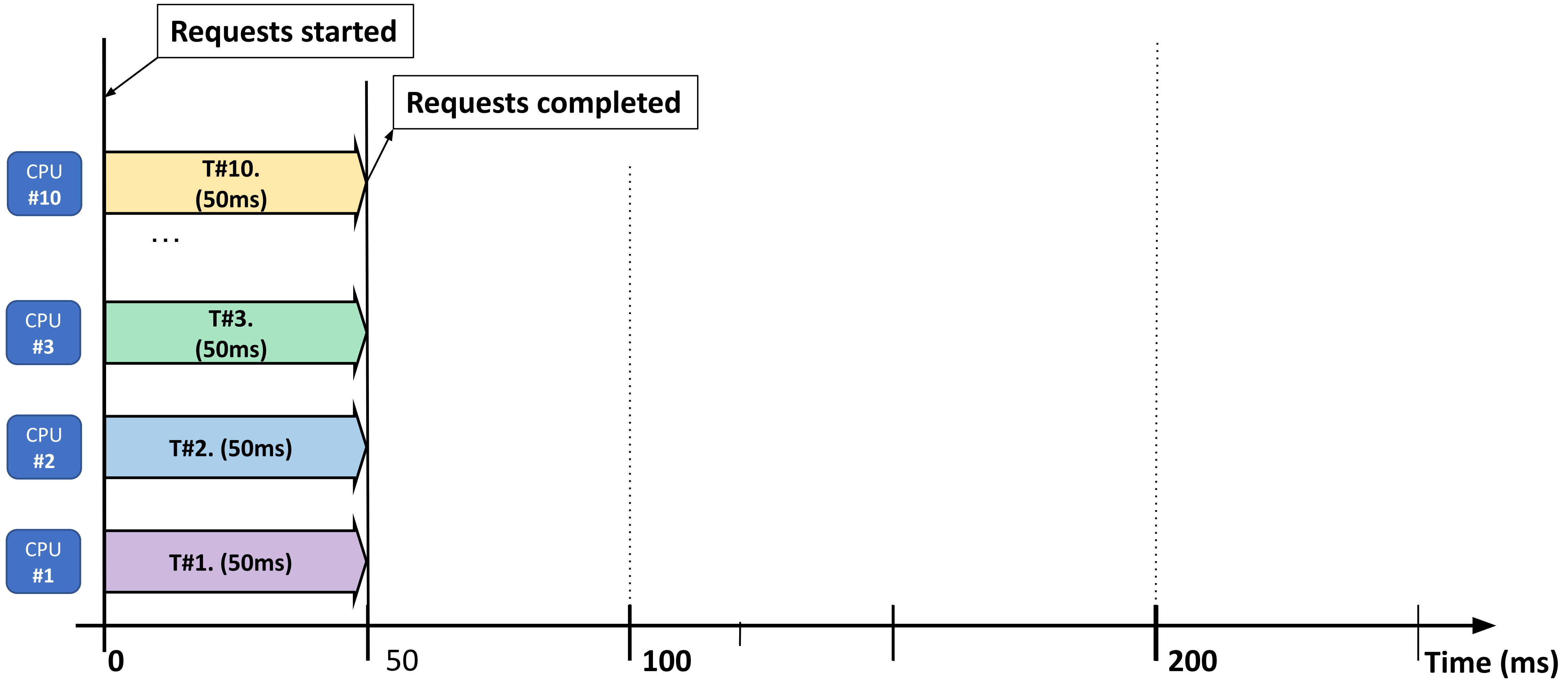


Throttling Explained. Example#3.1.2: Multi-Tasks

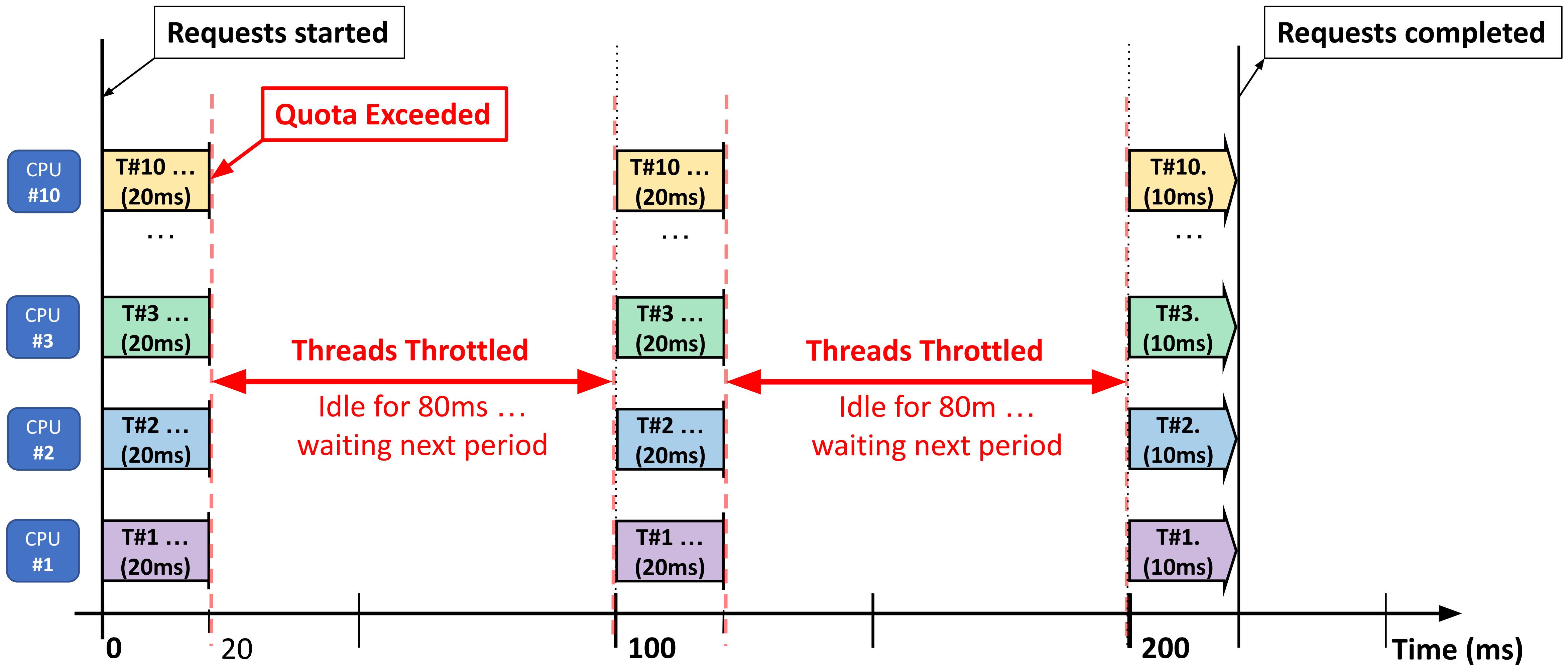
- Use-case#3.1.2: 2 parallel tasks requiring 100ms CPU Time, each one running on different CPU.
- CPU Limits = 2. Quota (cpu.max) = “200ms within a 100ms”
- Everything’s OK. Full Quota used, but NOT exceeded (near the limit).



Example#3.2.1: Multi-Tasks – Visual.
10 parallel tasks, each requiring 50ms CPU Time, each one running on different CPU. No CPU Limits.

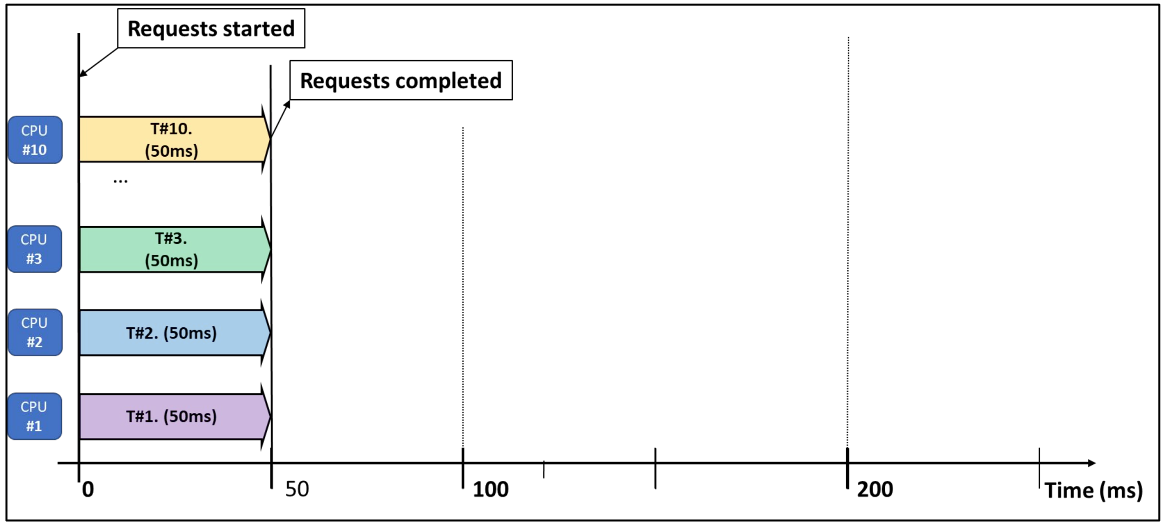


Example#3.2.2: Multi-Tasks – Visual.
10 parallel tasks, each requiring 50ms CPU Time, each one running on different CPU. **<u>CPU Limits = 2</u>**.



Ex.3.2.1: 10 parallel Tasks, No CPU Limits. Texts

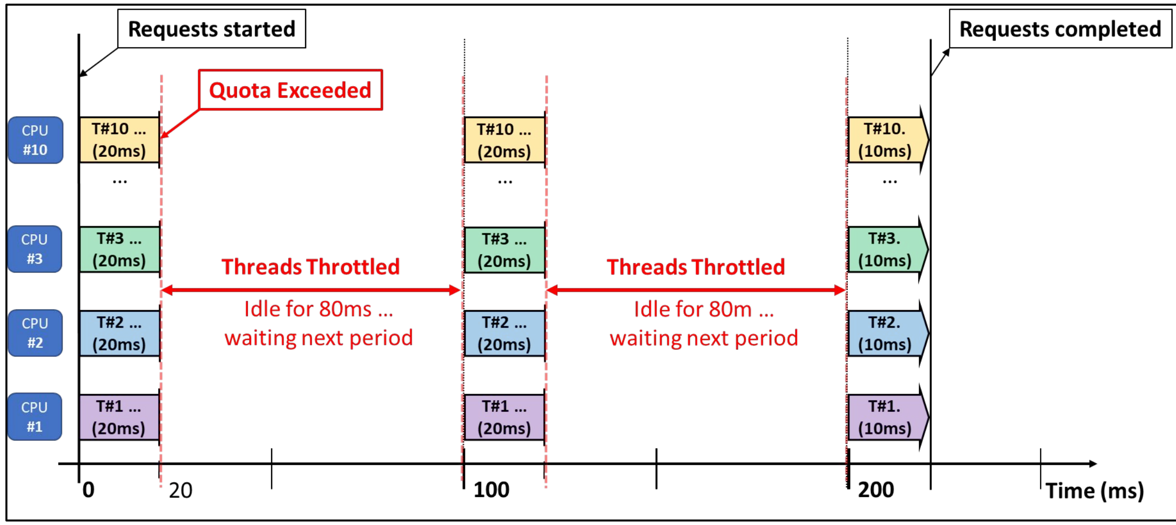
- All 10 tasks started simultaneously, true parallelly due to multiple CPUs available.
- Each task demands 50ms to complete and will get that time.



- Unlimited CPU Quota so no blockages (throttling).
- Wall clock time for all tasks to complete = **50ms**.
- cgroup Total CPU Time = 10 threads * 50ms = **500ms**,
- CPU Usage = 100% * (500ms/50ms) = **1000%**.
  - CPU Usage = CPU Time / (monitored wall clock interval). Above is equivalent to 10 CPU cores loaded at 100% during monitored interval of 50ms.

Ex.3.2.2: 10 parallel Tasks, **CPU Quota = 2**. Texts

- All 10 tasks started simultaneously, true parallelly due to multiple CPUs available.
- Each task demands 50ms to complete and … will get that time?
- CPU Quota = 2 CPUs = 200ms per 100ms of CFS Period.



- 1st CFS Period: Hard CPU Throttling occurred after 20 ms of wall clock time. The cgroup tasks consumed their quota of 200ms (10 threads * 20ms). cgroup is throttled for remaining 80ms and need to wait until next period starts.
- 2nd Period – the same as in 1st.
- 3rd Period – tasks will complete execution, running for remained 10ms.
- Wall clock time for all tasks to complete = **210ms**.
- cgroup Total CPU Time still = 10 threads * 50ms = **500ms**,
- CPU Usage = (500 / 210) = 238%, underutilized compared to previous use-case.

Ex.3.2.2: 10 parallel Tasks, **CPU Quota = 2**. Calculus

● Standard metrics reflected in Grafana:

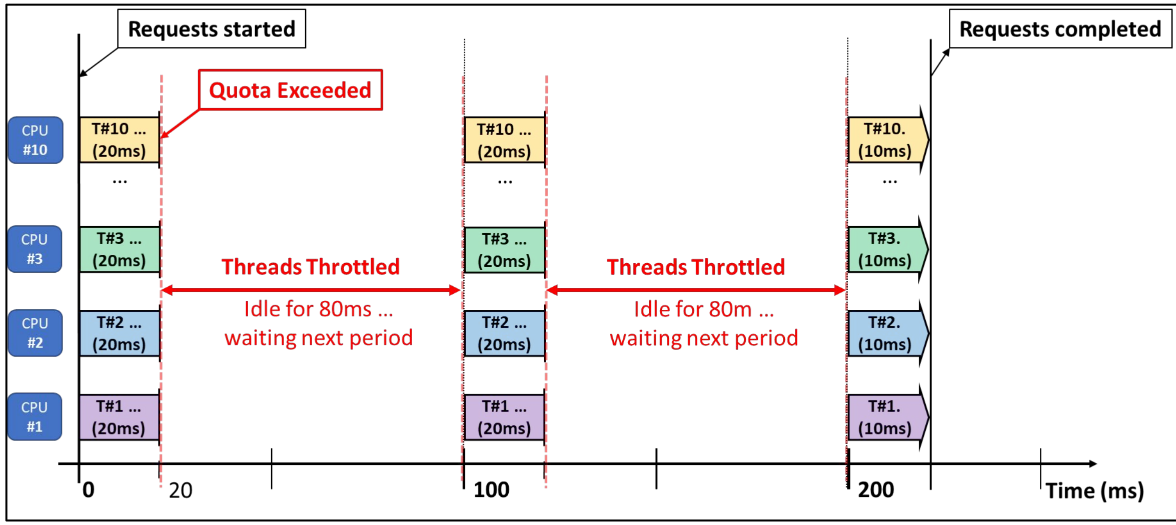
$$Throtlled\ \%\ (Periods) = 100\% * \frac{\Delta nr\_throttled}{\Delta nr\_periods} =$$

$$= 100\% * \frac{2}{3} = 67\%;$$

Reflects severity: more than half of CPU available time is lost:

$$Throtlled\ \%\ (Time) = 100\% * \frac{\Delta throttled\_ms}{\Delta \text{nr\_periods} * quota\_period} =$$

$$= 100\% * \frac{160}{3 * 100} = 53\%;$$



$$CPU\ Efficiency\ \%\ (Time) = 100\% * \frac{\Delta usage\_ms}{\Delta usage\_ms + \Delta throtled\_ms} =$$

$$= 100\% * \frac{500}{500+160} = 76\%;$$ -- CPUs underutilization.

$$Desired\ CPUs = \frac{\Delta usage\_ms + \Delta throttled\_ms}{\Delta nr\_periods * quota\_period} = \frac{500+160}{3*100} = \frac{660}{300} = 2.2;$$ -- Very inaccurate due to parallelism

Throttling Explained. Example#4

- Cgroup with 2 CPU-Bound Tasks continuously requiring full CPU (long or infinite computation loop).
  - 2 single-threaded processes or 2 threads, 2 available CPUs, no noisy neighbors.
- Monitored interval = 1 sec, sampling each 100ms CFS Period.

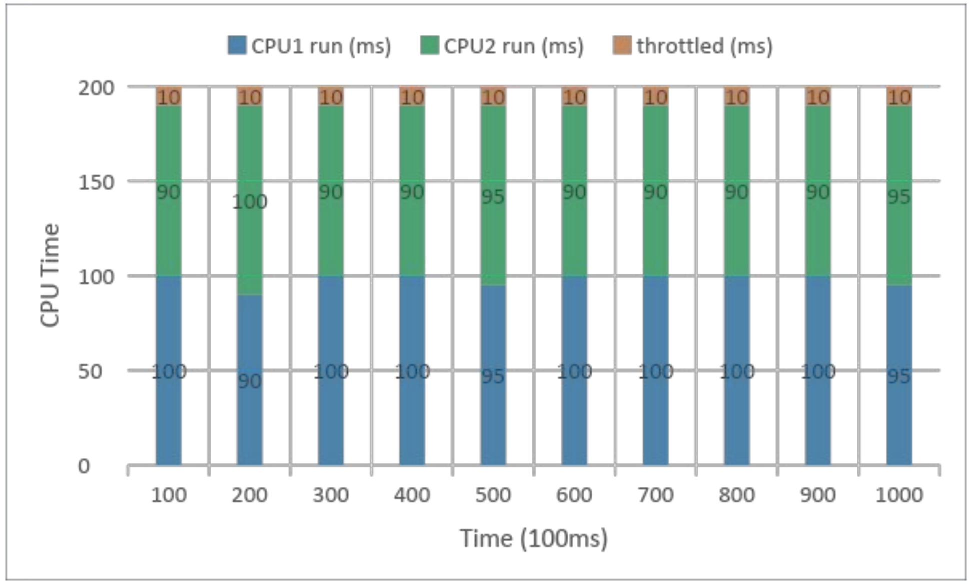
- CPU Limit = 1.9

Throttling Explained. Example#4: Calculus

Time	CPU1 run (ms)	CPU2 run (ms)	throttled (ms)
100	100	90	10
200	90	100	10
300	100	90	10
400	100	90	10
500	95	95	10
600	100	90	10
700	100	90	10
800	100	90	10
900	100	90	10
1000	95	95	10

```
#cpu.max
190_000 100_000

# cpu.stat
usage_usec 1_900_000
nr_periods 10
nr_throttled 10
throttled_usec 100_000
```



- $Throttled\ \%\ (Periods) = 100\% * \frac{\Delta nr\_throttled}{\Delta nr\_periods} = 100\% * \frac{10}{10} = 100\%$

- $Throttled\ \%\ (Time) = 100\% * \frac{\Delta throttled\_ms}{\Delta nr\_periods * quota\_period} = 100\% * \frac{100}{10 * 100} = 10\%$

- $CPU\ Efficiency\ \%\ (Time) = 100\% * \frac{\Delta usage\_ms}{\Delta usage\_ms + \Delta throttled\_ms} = 100\% * \frac{1900}{1900+100} = 95\%$

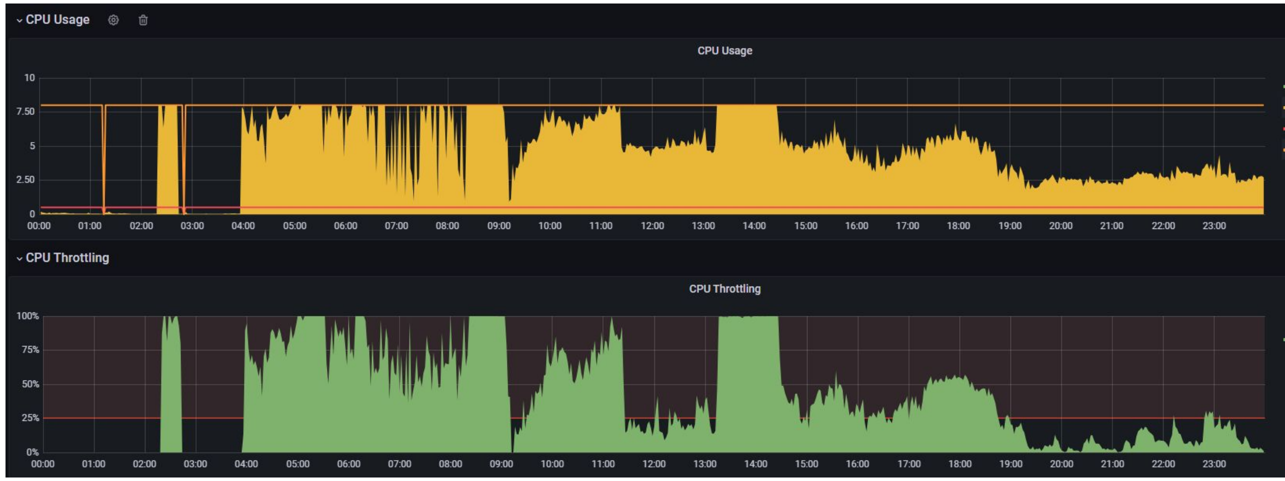
- $Desired\ CPUs = \frac{\Delta usage\_ms + \Delta throttled\_ms}{\Delta nr\_periods * quota\_period} = \frac{1900+100}{10*100} = \frac{2000}{1000} = 2$

CPU Throttling

More explanations

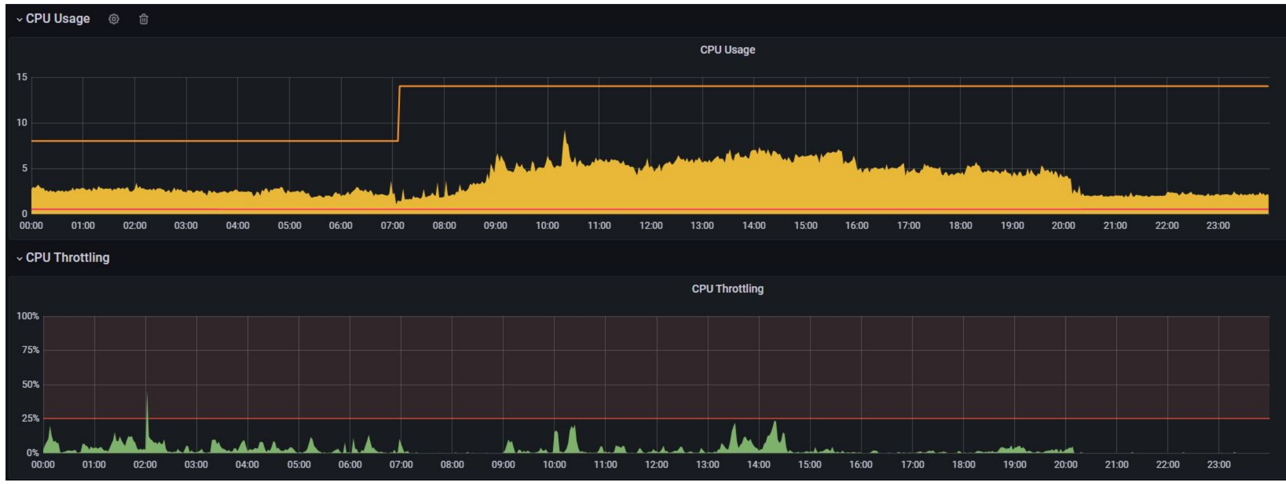
Visual: Throttling use-cases

- This is one of critical containerized workloads, with CPU Limits = 8 of 16 Allocatable vCPUs on the node and ~100 tasks inside (mix of active, idle and bursting tasks).
- CPU Usage and Throttling Grafana Reports display almost same shape, which mean that containers' tasks needed in bursting periods much more than allocated limits.
- Due to enforced CPU Quota workload tasks were intensively throttled, causing various issues (heartbeats lost, high latency, timed out requests, etc.)

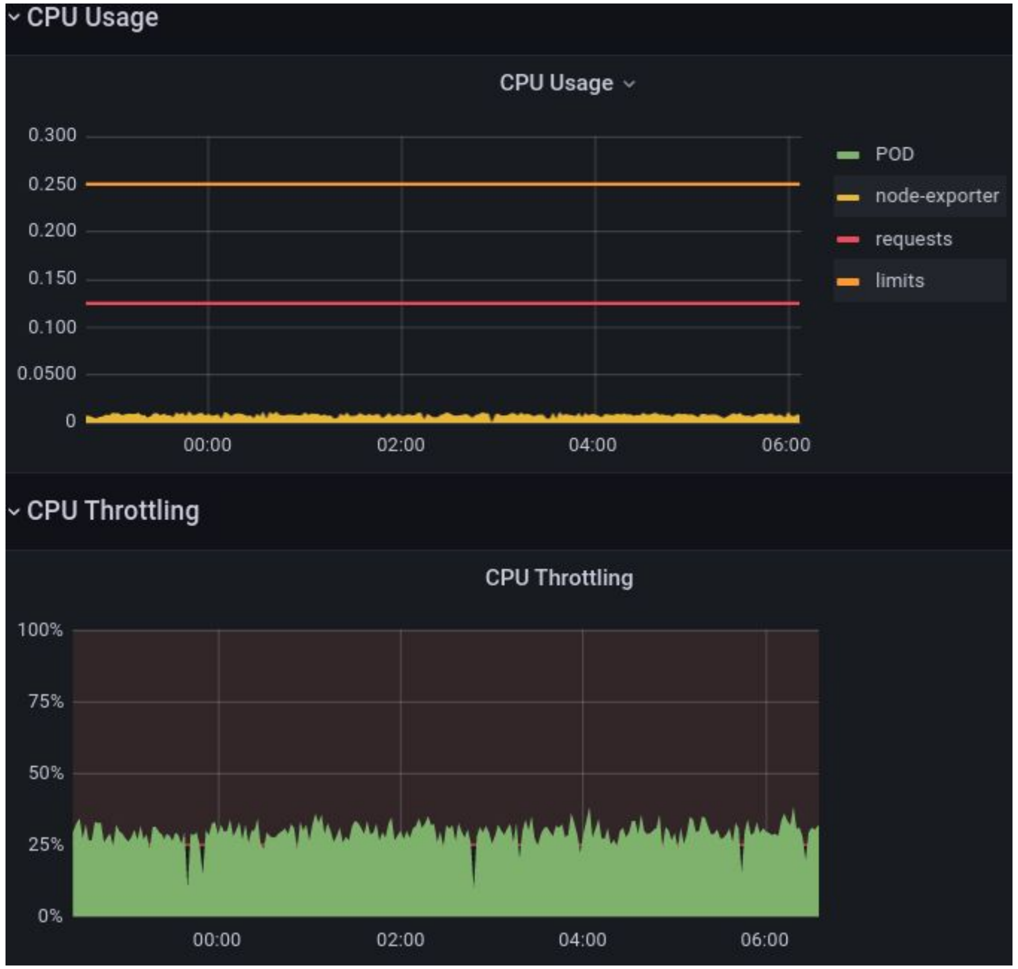


- After changed CPU Limits from 8 to 14, throttling started disappear and workload stability improved significantly.

- **Conclusion:** CPU Limits are bad for critical workloads!



Visual: Throttling, other use-case: CPU Low usage, High Throttling.



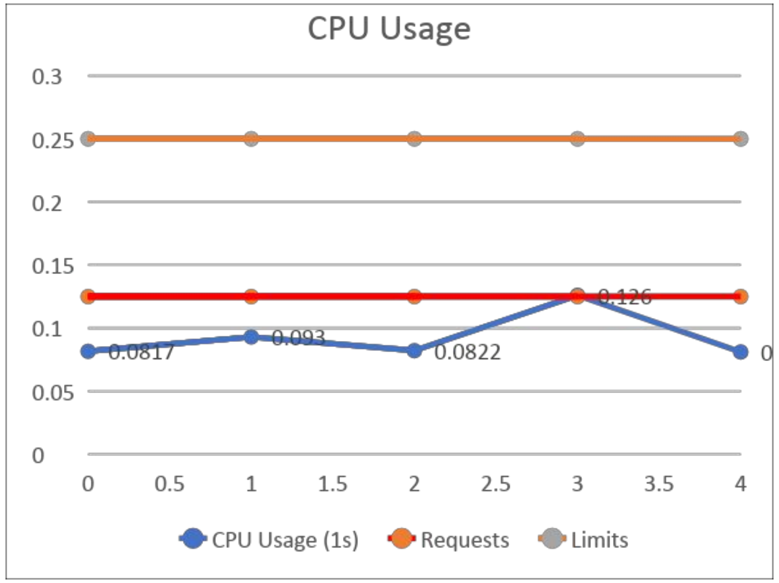
- But how to explain such use-case, when CPU Usage is below than request, but hard Throttling is reported?
    - Requests = 125m
    - Limits = 250m
    - AVG Usage ~=0.01
    - Throttling: ~25% .. 33%
- CPU Usage looks low here because of data are averaged through long period of time, where workload is idle or has low CPU demands most of the time, but bursts periodically.
- Throttling is high because more than 25% of used CPU CFS Periods were throttled.
- Let try zoom in for interval of 4-5 seconds – we may observe something like (see next page):

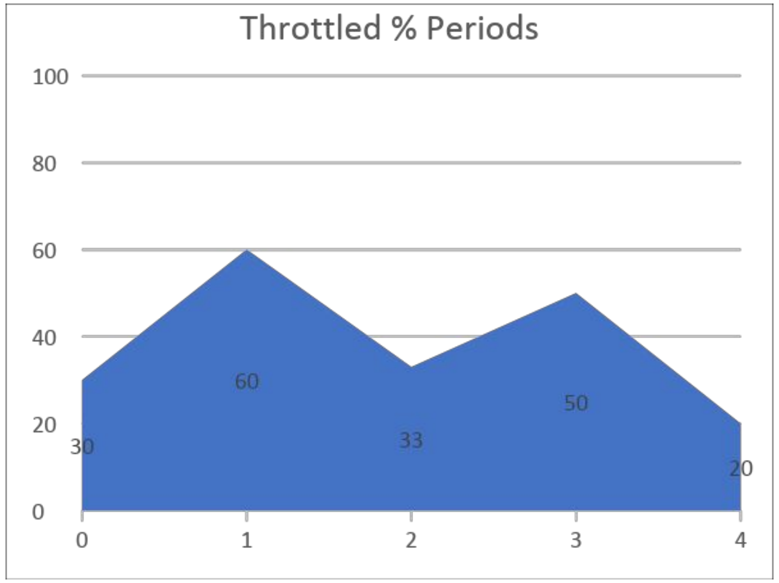
CPU Low usage, High Throttling:

Below, in the table, are simulated samples for CPU usage (time) and Throttling at 1 second resolution:

Time (s)	CPU Usage (AVG 1s)	Throttled % Period
0	0.0817	30
1	0.093	60
2	0.0822	33
3	0.126	50
4	0.081	20

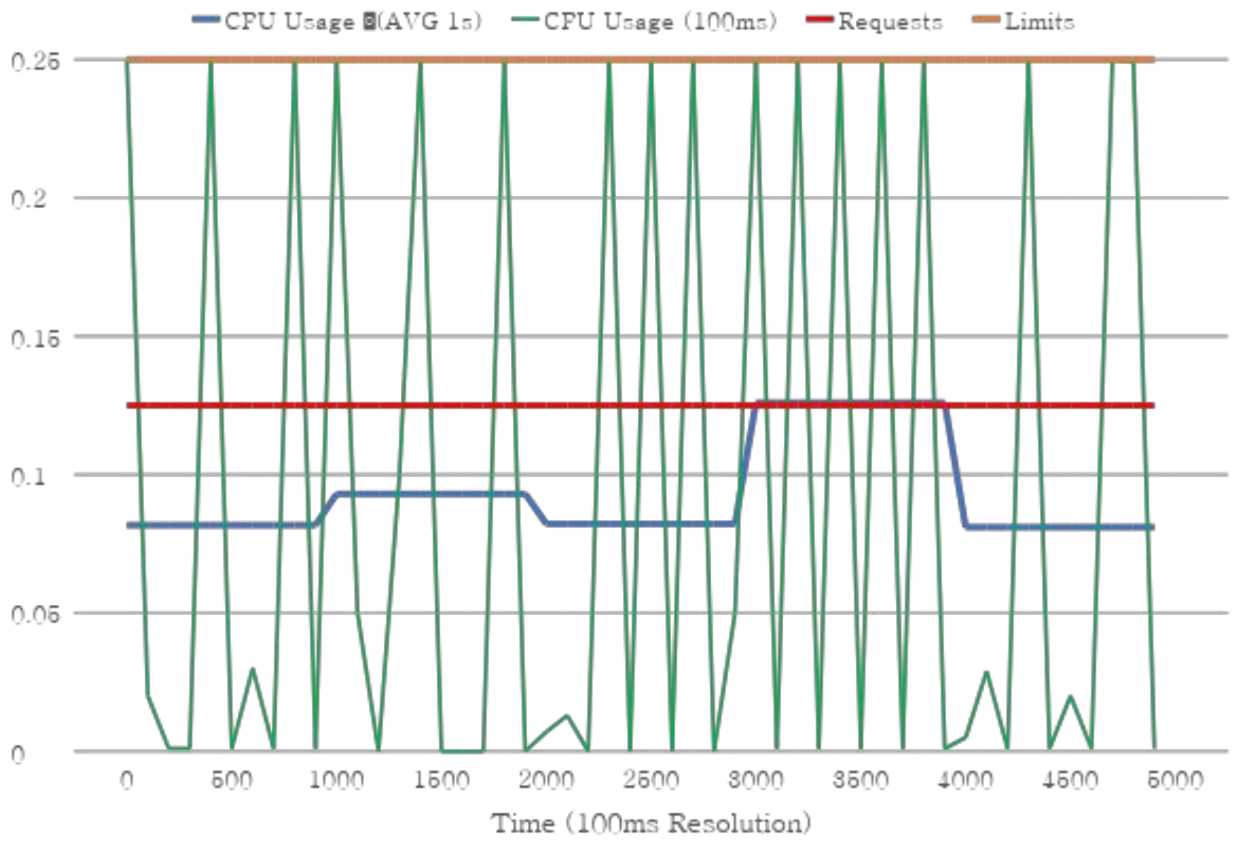
- Zoom in for period of 5 seconds – we may observe higher CPU Usage, because of more narrow monitoring interval where it's supposed that workload had some activity spike, but also higher Throttling – How is it possible?
- Within 1 sec there are 10 CFS Periods and some of them may be throttled.
- Let zoom in it at resolution of 100 ms (see next slide)





CPU Low usage, High Throttling:

CPU Usage and Throttled periods per every 100ms CFS Quota Period.



Time (ms)	CPU Usage (AVG 1s)	CPU Usage (100ms)	Period (1-used/0-not)	Throttled Period (1-Yes/0-No)	Throttled % Period
0		0.251	1	1	
100		0.02	1	0	
200		0.001	1	0	
300		0.001	1	0	
400	0.0817	0.251	1	1	30%
500		0.001	1	0	
600		0.03	1	0	
700		0.001	1	0	
800		0.26	1	0	
900		0.001	1	1	
1000		0.26	1	1	
1100		0.05	1		
1200		0	0		
1300		0.1	1		
1400	0.093	0.26	1	1	60%
1500		0	0		
1600		0	0		
1700		0	0		
1800		0.26	1	1	
1900		0	0		
2000		0.007	1		
2100		0.013	1		
2200		0			
2300		0.252	1	1	
2400	0.0822	0			33%
2500		0.249	1		
2600		0			
2700		0.251	1	1	
2800		0			
2900		0.05	1		
3000		0.251	1	1	
3100		0.001	1		
3200		0.251	1	1	
3300		0.001	1		
3400	0.126	0.251	1	1	50%
3500		0.001	1		
3600		0.251	1	1	
3700		0.001	1		
3800		0.251	1	1	
3900		0.001	1		
4000		0.005	1		
4100		0.029	1		
4200		0.001	1		
4300		0.252	1	1	
4400	0.081	0.001	1		20%
4500		0.02	1		
4600		0.001	1		
4700		0.251	1	1	
4800		0.249	1		
4900		0.001	1		

CPU Request, Limits and Throttling

Further thoughts…

CPU Throttling: Impacts on Applications

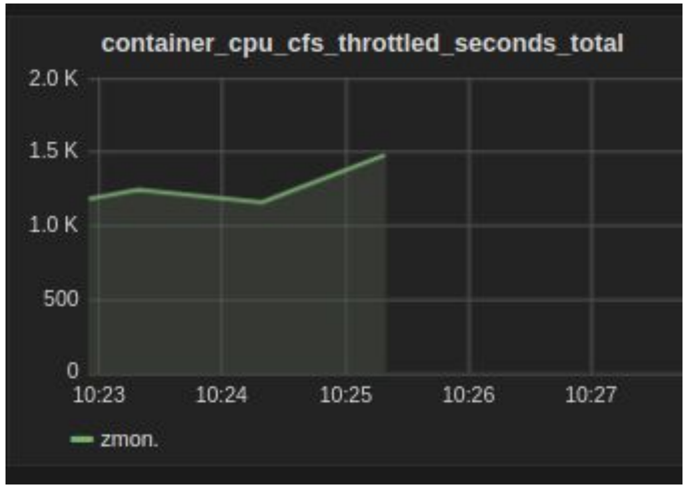
- Throttling JVM/CLR Garbage Collector (GC) threads may lead to:
  - killing container by Kernel with “OOMKilled” reason because of exceeding memory limits, or
  - Runtime itself may fail with “Out of Heap Space” unrecoverable exception.
  - In both case because GC Threads being throttled may not succeed reclaiming memory in time.
- Readiness/Liveness probe failures.
- Latency in critical operations.
- … and a lot of others…

CPU Usage vs Throttling. Grafana Reports

- Beside well known - CFS Periods based - Throttling report

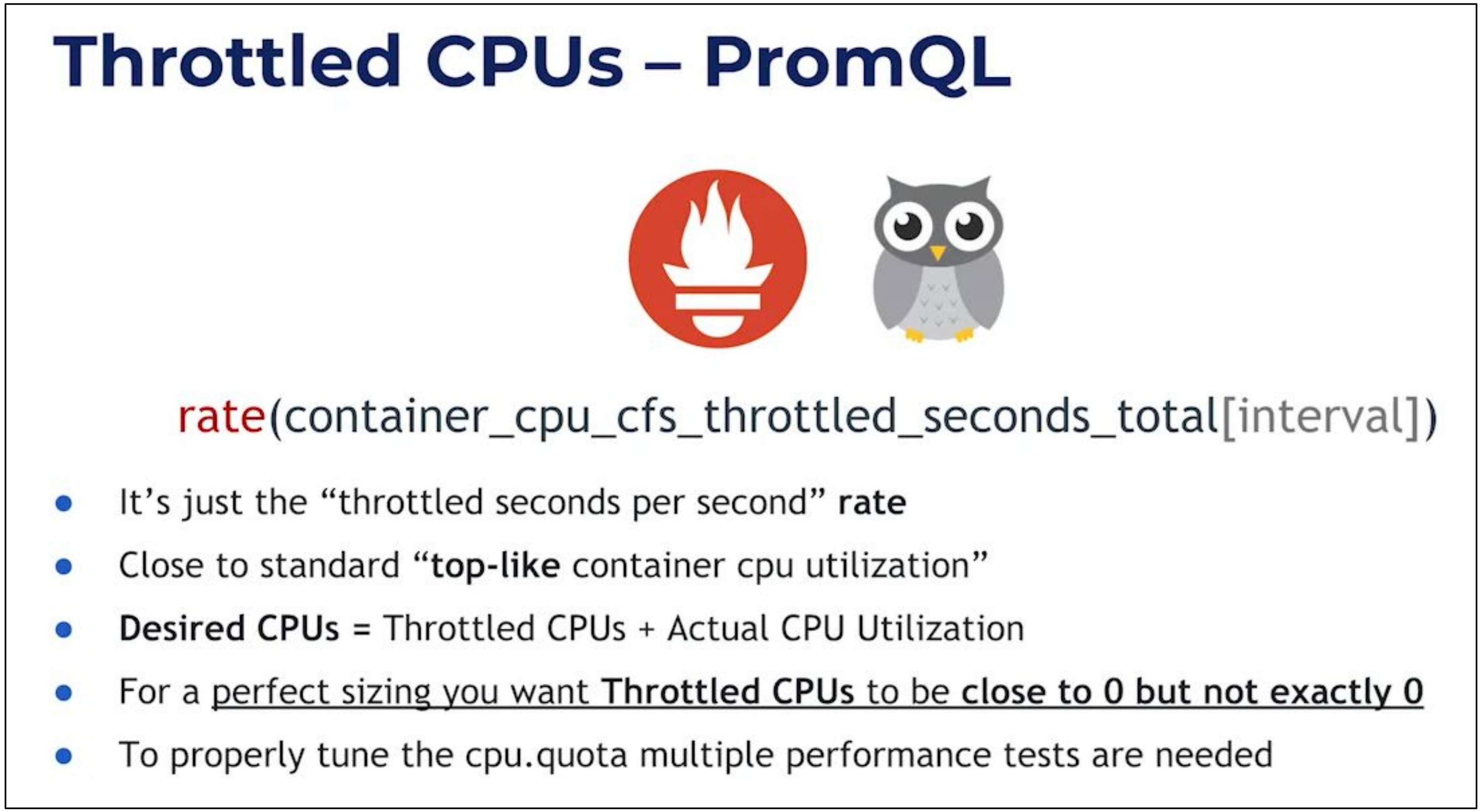
```
100 * (rate(container_cpu_cfs_throttled_periods_total{namespace="$namespace", pod="$pod"}[1m]) /
rate(container_cpu_cfs_periods_total{namespace="$namespace", pod="$pod"}[1m]))
```

- it is recommended to add also some reports based on cadvisor metric "`container_cpu_cfs_throttled_seconds_total`", also available in Prometheus + Grafana monitoring stacks.



CPU Usage vs Throttling. Grafana Reports, cont'd

Valuable usage of time-based throttled metrics were proposed by **Francesco Fabbrizio** at Performance Summit 2021.



Throttled CPUs – PromQL

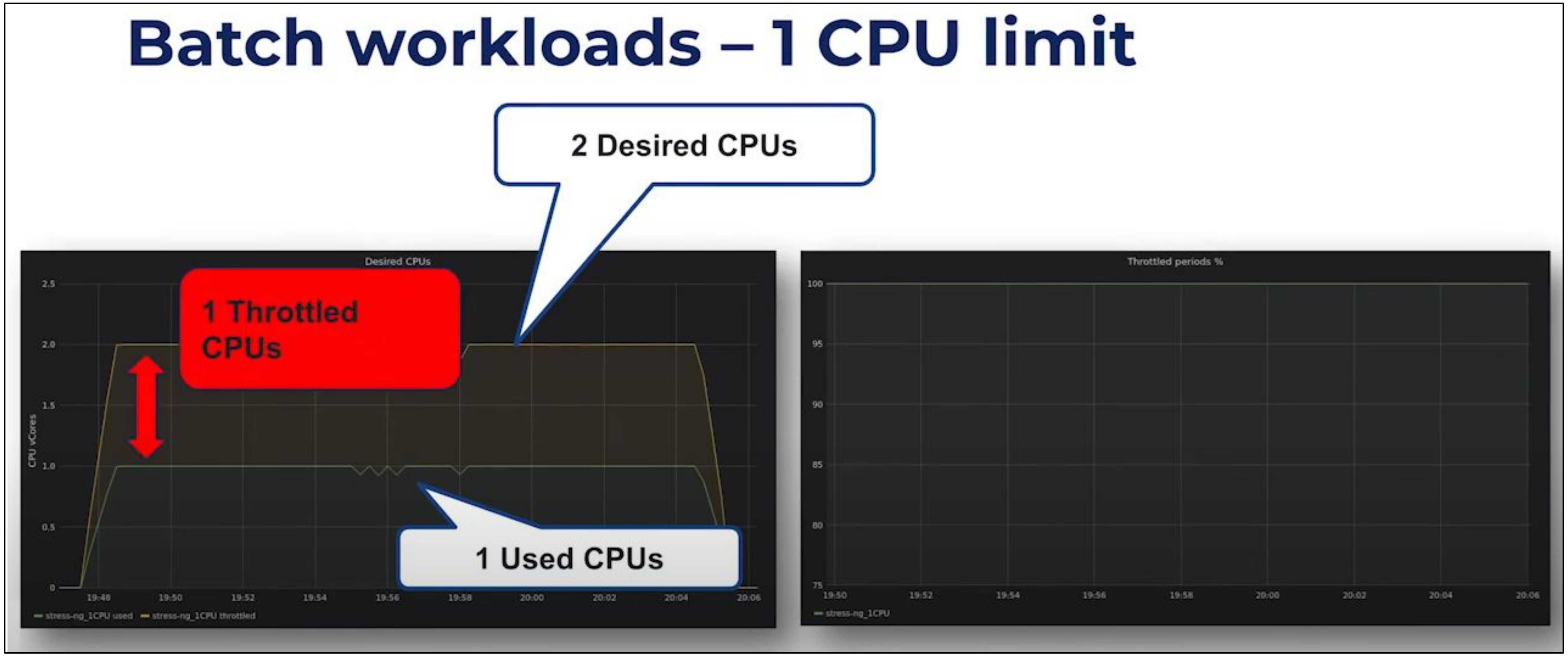
```
rate(container_cpu_cfs_throttled_seconds_total[interval])
```

- It's just the "throttled seconds per second" **rate**
- Close to standard "**top-like** container cpu utilization"
- **Desired CPUs** = Throttled CPUs + Actual CPU Utilization
- For a perfect sizing you want **Throttled CPUs** to be **close to 0 but not exactly 0**
- To properly tune the cpu.quota multiple performance tests are needed

source: https://www.youtube.com/watch?v=beWXWEimTxs&t=671s

CPU Usage vs Throttling. Grafana Reports, cont’d

“**Desired CPU**” estimation, from the same presentation of **Francesco Fabbrizio** at Performance Summit 2021.



source: https://youtu.be/beWXWEimTxs?t=1000

CPU Request, Limits and Throttling

Final Thoughts,
Best and/or Community Practices,
Recommendations.

Community Opinions: High Overview

This is one of the most debatable, complex and contradictory subject in community. There are 2 popular opposite-opinion popular articles on topic:

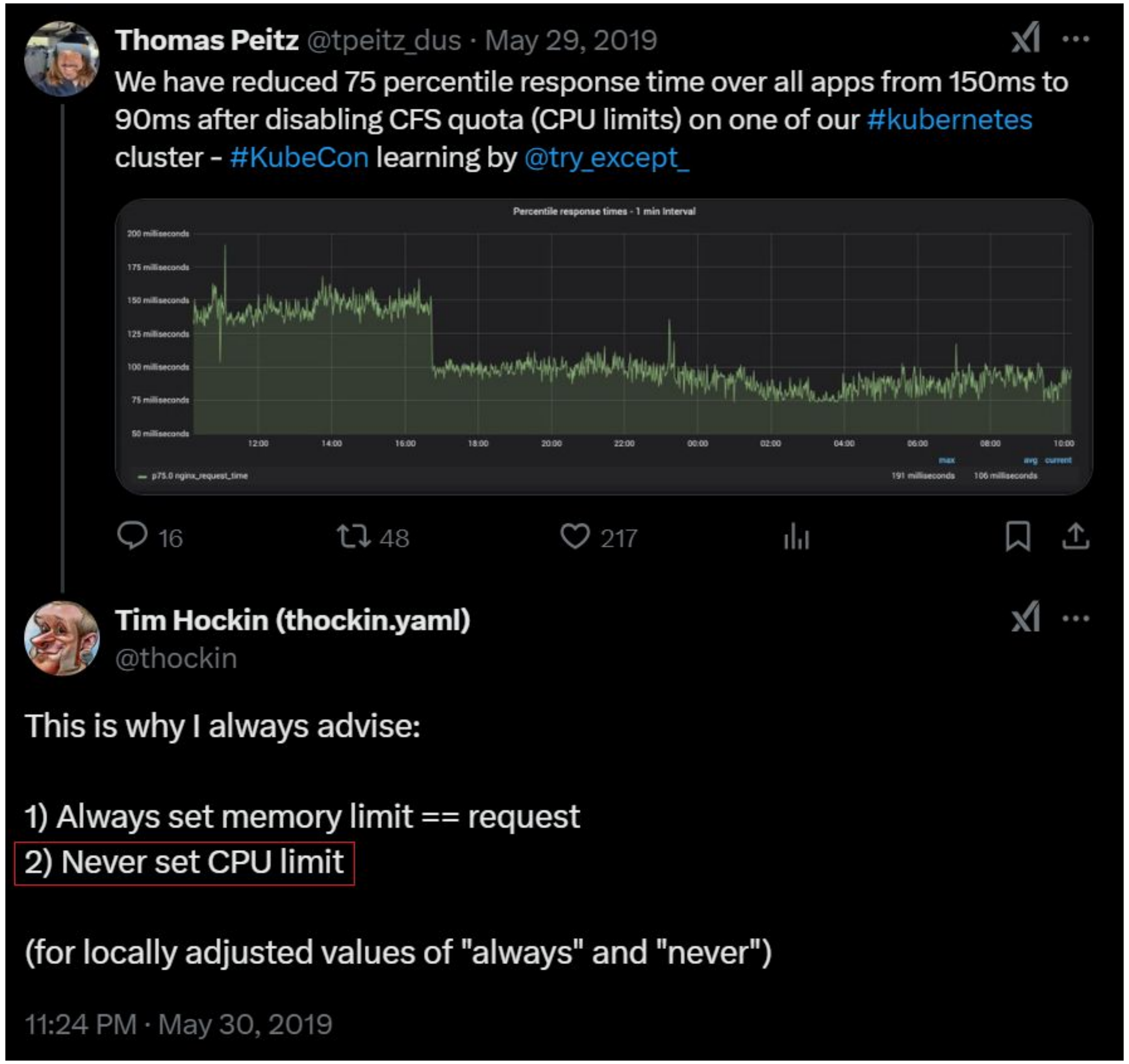
- "**CPU limits on Kubernetes are an antipattern**": https://home.robusta.dev/blog/stop-using-cpu-limits.
  - The main idea is to remove at all CPU Limits and let workloads, with properly defined CPU Requests, consume as much spare CPU as they need, in order to avoid catastrophic failures, high latencies, etc.

vs

- "**Why You Should Keep Using CPU Limits on Kubernetes**" - https://dnastacio.medium.com/why-you-should-keep-using-cpu-limits-on-kubernetes-60c4e50dfc61.
  - The main idea of which is to define right and test proven CPU Requests/Limits and stay within those sizes but not rely too much on available CPU which may be unavailable when deploy environment suffer changes.

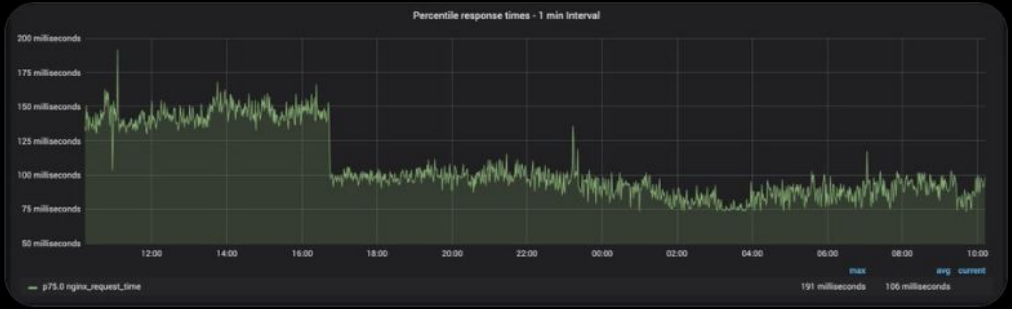
Community Opinions: High Overview, Cont'd

- **Tim Hockin**, one of current and from day#0 K8s maintainer, affirmed several times that in most cases CPU Limits should be removed:
  - https://x.com/thockin/status/1134193838841401345
  - Restating and elaborating on the idea: https://news.ycombinator.com/item?id=24381813



Thomas Peitz @tpeitz_dus · May 29, 2019

We have reduced 75 percentile response time over all apps from 150ms to 90ms after disabling CFS quota (CPU limits) on one of our #kubernetes cluster - #KubeCon learning by @try_except_



16 48 217

Tim Hockin (thockin.yaml)
@thockin

This is why I always advise:

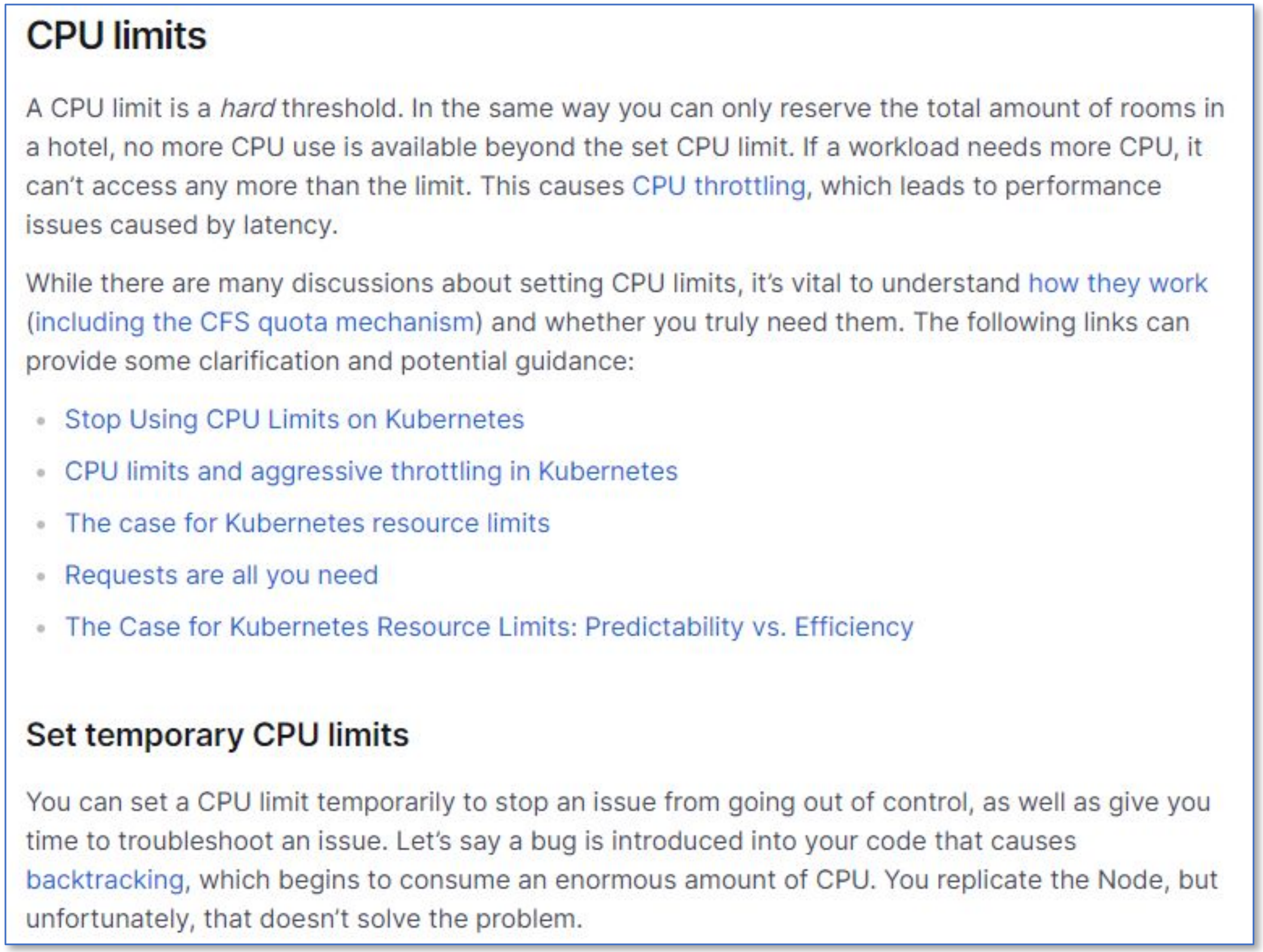
1) Always set memory limit == request
2) Never set CPU limit

(for locally adjusted values of "always" and "never")

11:24 PM · May 30, 2019

Community Opinions: More sources

Grafana Labs highlights the drawbacks of CPU limits on their website, providing several articles that advocate for either removing CPU limits entirely or using them only in specific scenarios.



CPU limits

A CPU limit is a *hard* threshold. In the same way you can only reserve the total amount of rooms in a hotel, no more CPU use is available beyond the set CPU limit. If a workload needs more CPU, it can't access any more than the limit. This causes CPU throttling, which leads to performance issues caused by latency.

While there are many discussions about setting CPU limits, it's vital to understand how they work (including the CFS quota mechanism) and whether you truly need them. The following links can provide some clarification and potential guidance:

- Stop Using CPU Limits on Kubernetes
- CPU limits and aggressive throttling in Kubernetes
- The case for Kubernetes resource limits
- Requests are all you need
- The Case for Kubernetes Resource Limits: Predictability vs. Efficiency

Set temporary CPU limits

You can set a CPU limit temporarily to stop an issue from going out of control, as well as give you time to troubleshoot an issue. Let's say a bug is introduced into your code that causes backtracking, which begins to consume an enormous amount of CPU. You replicate the Node, but unfortunately, that doesn't solve the problem.

src: https://grafana.com/docs/grafana-cloud/monitor-infrastructure/kubernetes-monitoring/optimize-resource-usage/container-requests-limits-cpu/#cpu-limits

Community Opinions: more

“We can do many things to save cost, **but throttling workloads** in a way that hurts their performance and maybe even availability **should be our last resort**.

… (After Autoscalers introducing)… In this situation, **we don’t need Limits** – just accurate Requests for/on everything (which ensures the minimum each one requires via CPU shares to function) and the Cluster Autoscaler. This will ensure that no workload is throttled unnecessarily and that we have the right amount of capacity at any given time.”

src: https://sysdig.com/blog/kubernetes-cpu-requests-limits-autoscaling/

“**Kubernetes CPU Limit Best Practices: Avoid CPU Limits for Performance**: If performance is your goal, avoid setting CPU limits to allow your containers to utilize idle CPU resources. This approach maximizes the use of available resources and improves the overall performance of your applications.”
src: https://www.perfectscale.io/blog/kubernetes-cpu-limit-best-practises

“**Disable CPU limits — unless you have a good use case**”

src: https://learnk8s.io/production-best-practices

Cockroach Labs on CockroachDB: “**Warning:** While setting memory limits is strongly recommended, setting CPU limits can hurt tail latencies as currently implemented by Kubernetes. **We recommend not setting CPU limits at all *unless…***”

src: https://www.cockroachlabs.com/docs/stable/kubernetes-performance.html#resource-requests-and-limits

Community Opinions: more+

- And yet again PerfectScale: Our rule of thumb is: “Remove K8s CPU limits - they are inefficient, wasteful and may negatively impact performance”.
  - src: https://www.perfectscale.io/blog/kubernetes-cpu-limits

- “Optimizing resource requests and limits is hard. Although it’s much easier to break things when setting limits, those breakages might help prevent a catastrophe later”
  - src: https://kubernetes.io/blog/2023/11/16/the-case-for-kubernetes-resource-limits/

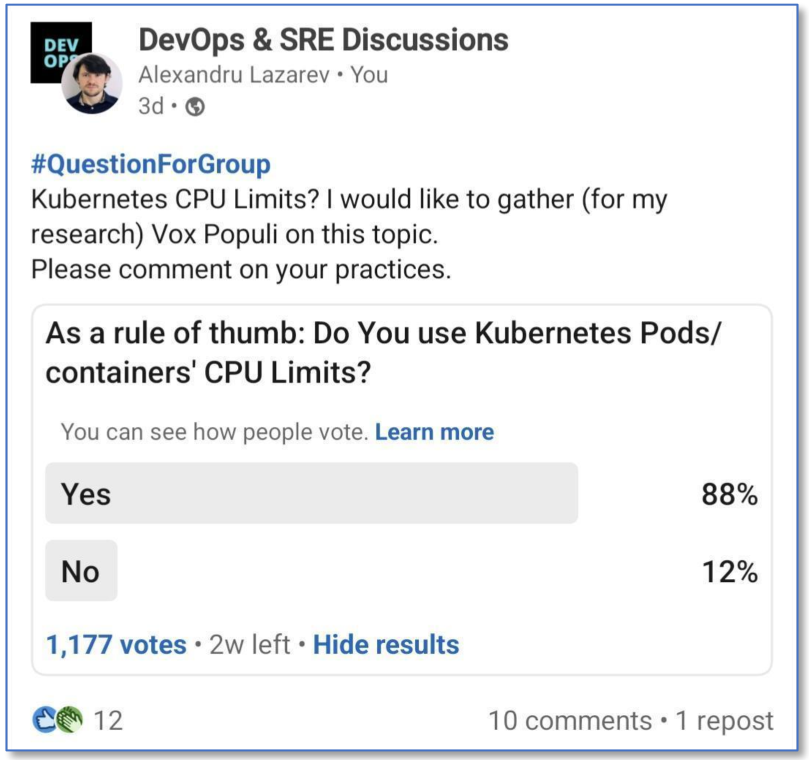
- “Understand the biggest Kubernetes misunderstanding and **why you should remove your CPU limits** and unleash your cluster's full potential” by Shon Lev-Ran, K8s optimization expert, slimstack.io,
  - src: https://medium.com/directeam/kubernetes-resources-under-the-hood-part-3-6ee7d6015965

- NetData demystifying various myths on topic: “Kubernetes Throttling Doesn’t Have To Suck. Let Us Help! A lot of engineers advise the use of CPU limits on every container as Kubernetes best practice. Unfortunately, as we will prove below, they are **wrong: CPU limits should rarely be used, if used at all!**”
  - src: https://www.netdata.cloud/blog/kubernetes-throttling-doesnt-have-to-suck-let-us-help/

“**Kubernetes: Make your services faster by removing CPU limits**” by Eric Khun from Buffer.com.
  - src: https://erickhun.com/posts/kubernetes-faster-services-no-cpu-limits/

Community Opinions: more++

OTOH, my LinkedIn polls…



DevOps & SRE Discussions
Alexandru Lazarev • You
3d •

#QuestionForGroup
Kubernetes CPU Limits? I would like to gather (for my research) Vox Populi on this topic.
Please comment on your practices.

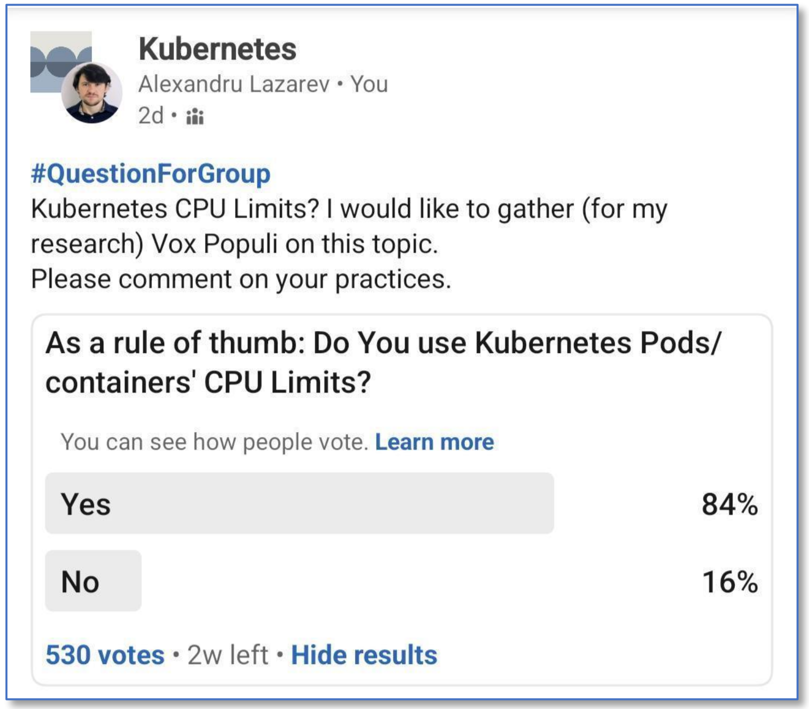
As a rule of thumb: Do You use Kubernetes Pods/ containers' CPU Limits?

You can see how people vote. Learn more

Yes	88%
No	12%

1,177 votes • 2w left • Hide results

12 10 comments • 1 repost



Kubernetes
Alexandru Lazarev • You
2d •

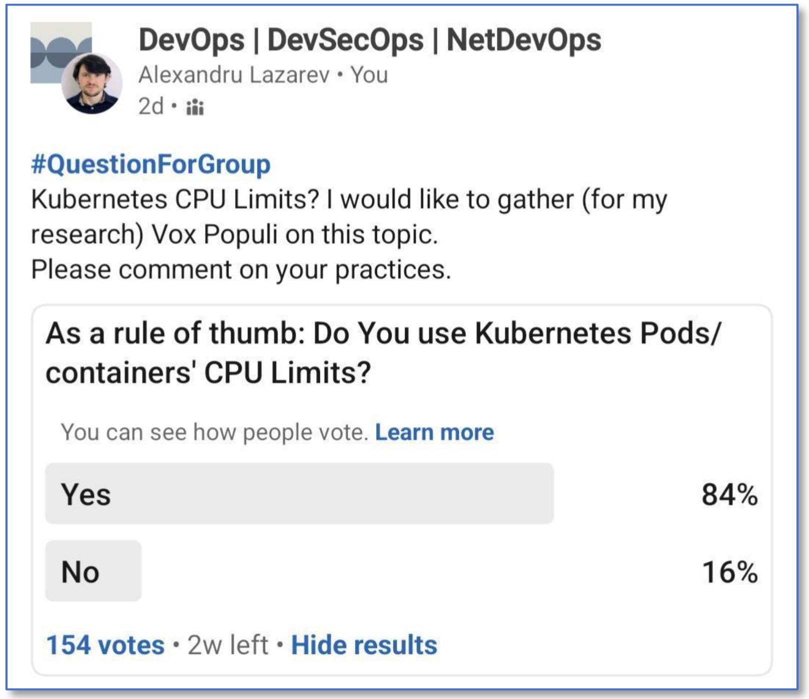
#QuestionForGroup
Kubernetes CPU Limits? I would like to gather (for my research) Vox Populi on this topic.
Please comment on your practices.

As a rule of thumb: Do You use Kubernetes Pods/ containers' CPU Limits?

You can see how people vote. Learn more

Yes	84%
No	16%

530 votes • 2w left • Hide results



DevOps | DevSecOps | NetDevOps
Alexandru Lazarev • You
2d •

#QuestionForGroup
Kubernetes CPU Limits? I would like to gather (for my research) Vox Populi on this topic.
Please comment on your practices.

As a rule of thumb: Do You use Kubernetes Pods/ containers' CPU Limits?

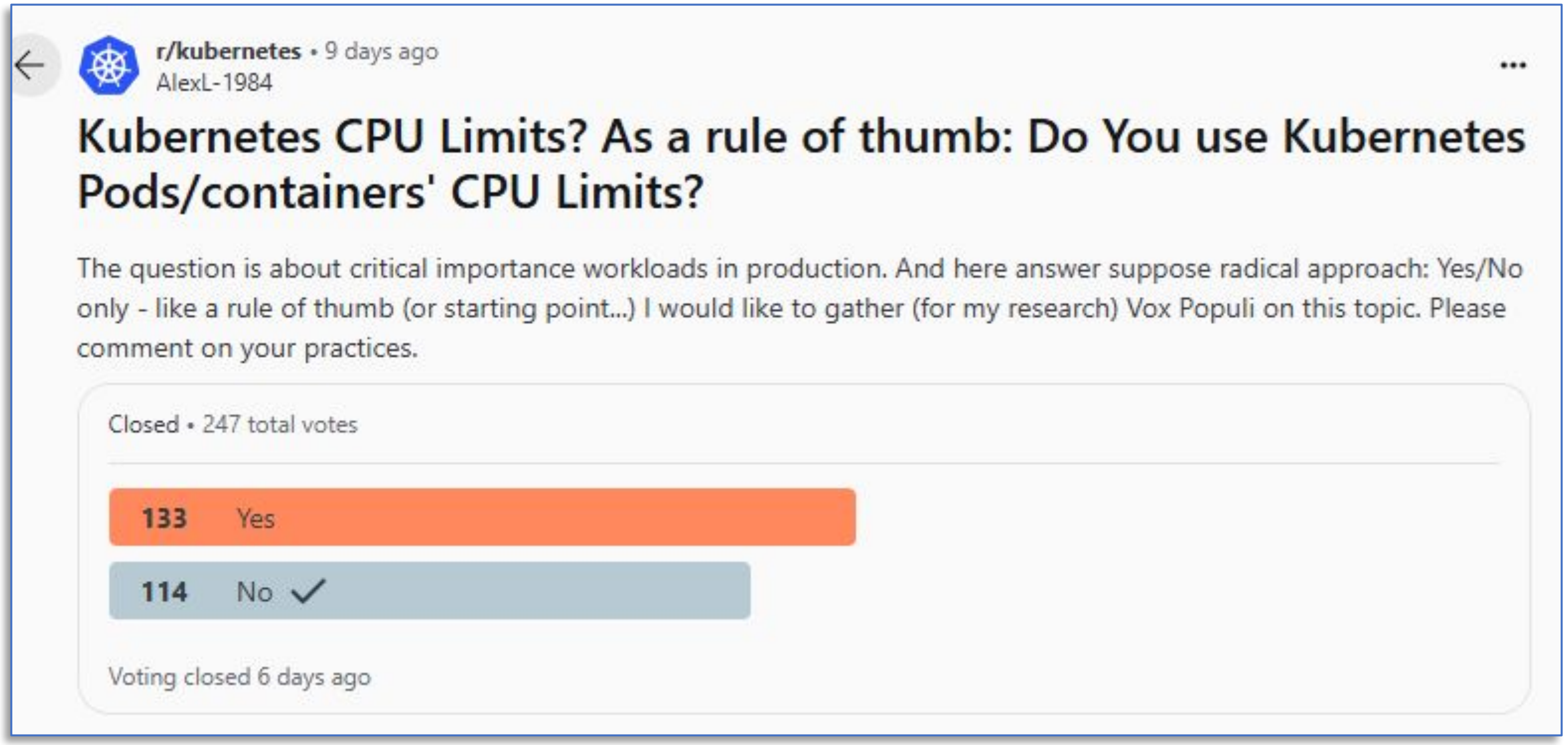
You can see how people vote. Learn more

Yes	84%
No	16%

154 votes • 2w left • Hide results

Community Opinions: more++

And on reddit


r/kubernetes • 9 days ago
AlexL-1984

Kubernetes CPU Limits? As a rule of thumb: Do You use Kubernetes Pods/containers' CPU Limits?

The question is about critical importance workloads in production. And here answer suppose radical approach: Yes/No only - like a rule of thumb (or starting point...) I would like to gather (for my research) Vox Populi on this topic. Please comment on your practices.

Closed • 247 total votes

133	Yes
114	No ✓

Voting closed 6 days ago

CPU Limits. **So what?**

- I *(author)* personally opt for avoiding CPU Limits in production, especially for latency-sensitive, time-critical workloads such as API Gateways, core system components, garbage collector threads, and similar processes.
  - Especially that calculate right-sized Limits is tricky process *(see my heuristics calculus formulas)*
- And yes: **always set proper CPU Requests!**
- However, CPU Limits are invaluable during **regression, performance, and stability testing phases**, where controlled environments are necessary.
- If you plan to remove CPU Limits in production, it’s crucial to conduct thorough testing in **two key scenarios**:
  - **With CPU Limits applied**, and
  - **Without CPU Quotas**, to ensure that your workloads behave predictably and reliably under real-world conditions.

- But ..? If ..?
  - *See on next slides…*

CPU Limits. Community Recommendations:

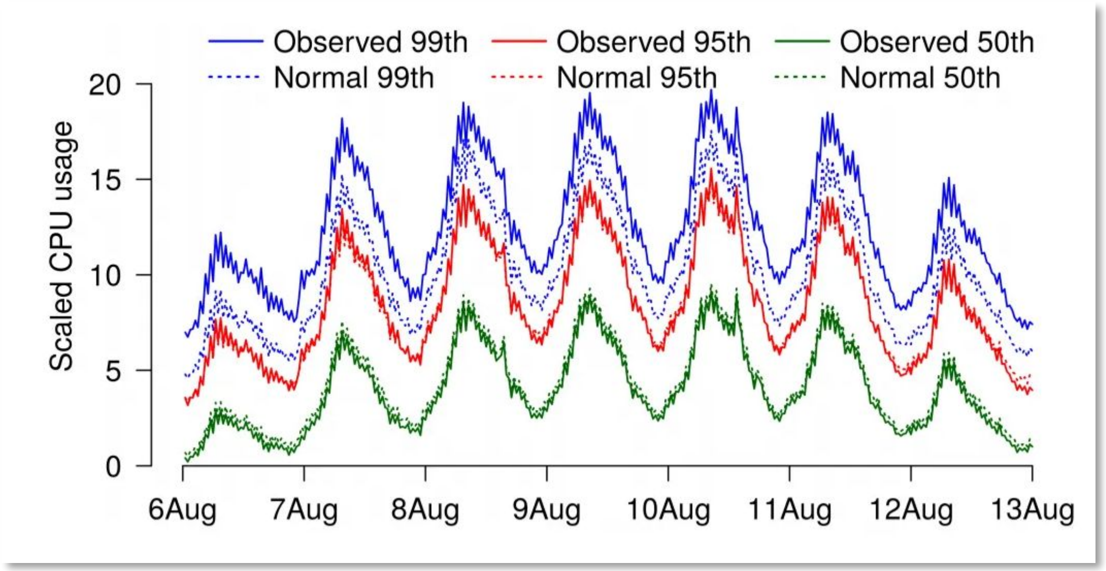
- The following slides outline common best practices for defining CPU Limits, specifically for scenarios where setting them is necessary.

- **Key takeaway**: Workloads should undergo **multiple iterative performance tests** and **production-like evaluations** to determine appropriate CPU Requests. Based on these requests and observed metrics, CPU Limits can then be fine-tuned for optimal performance.

CPU Limits. Community Recommendations: Cont'd

Following article https://itnext.io/cpu-limits-and-requests-in-kubernetes-fa9d55948b7c *(by Daniele Polencic from learnk8s.io)* reflects one of common opinions:

- CPU Limits = 99th percentile + 30-50%.
- You should profile the app (or use the VPA) for a more detailed answer.



CPU Limits. Community Recommendations: Cont’d

Kubecost (an IBM Company): You can classify applications into different availability tiers and apply these rules of thumb for targeting the appropriate level of availability:

Tier	Request	Limit
Critical / Highly Available	99.99th percentile + 100% headroom	2x request or as higher if resources available
Production / Non-critical	99th + 50% headroom	2x request
Dev / Experimental	95th or consider namespace quotas*	1.5x request or consider namespace quotas*

This best practice of analyzing historical usage patterns typically provides both a good representation of the future and is easy to understand/introspect. Applying extra headroom allows for fluctuations that may have been missed by your historical sampling. We recommend measuring usage over 1 week at a minimum and setting thresholds based on the specific availability requirements of your pod.
src: https://blog.kubecost.com/blog/requests-and-limits/

Community Opinions++: **When to Set CPU Limits**

Most common applicable use cases:

- **Benchmarking:** Performance and stability tests in staging environments, to understand workload sizing and fine-tune CPU requirements. Here is recommended to set limits closer to requests then retest, retest and again retest.
- **Multi-Tenant Environments:** Enforce fair CPU distribution across tenants using ResourceQuota.
- **Predictability:** CPU limits ensure consistent performance by preventing unpredictable resource overuse and spikes, enabling fair distribution and reliable application behavior, especially under SLAs.
- **Guaranteed QoS class:** hints kubelet to evict such pods as last resort if node is under resources (but not CPU) pressure, or when need to assign pod on a specific set of CPUs (see below)
    - https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/#static-policy-options
    - https://kubernetes.io/blog/2024/08/22/cpumanager-static-policy-distributed-cpu-across-cores/

Community Opinions++: **When to Set CPU Limits**

Again Tim Hockin:

“If you want to do it scientifically:

Benchmark your app under a load that represents the high end of reality. If you are preparing for BFCM *(**B**lack **F**riday **C**yber **M**onday)*, triple that.

For these benchmarks, set CPU request = limit.

Measure the critical indicators. Vary the CPU request (and limit) up or down until the indicators are where you want them (e.g. p95 latency < 100ms).”

- src and more very interesting details in this shot post: https://news.ycombinator.com/item?id=24381813

Other Best Practices

- Permanent CPU Metrics Monitor with
  - Prometheus/Grafana:
  - Others..?
- Use smart tools for CPU sizing predictability
  - HPA
  - Others (to investigate: KEDA, others)

CPU Request, Limits and Throttling

Inspiration Sources, References and nice or must-read articles

Inspiration Sources: Intro into the CPU-stuff topics

- Official K8s Docs: **Resource Management for Pods and Containers**
  - https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

- “**For the Love of God, Stop Using CPU Limits on Kubernetes. CPU limits on Kubernetes are an antipattern**” by Natan Yellin, Robusta.dev co-founder
  - https://home.robusta.dev/blog/stop-using-cpu-limits

vs

- Denilson Nastacio (IBM, Operations architect, corporate observer, software engineer, inventor): “**Why You Should Keep Using CPU Limits on Kubernetes. Or why staying away from unused CPU may be good for your containers**”
  - https://dnastacio.medium.com/why-you-should-keep-using-cpu-limits-on-kubernetes-60c4e50dfc61

Sources: Delving into… **Must Read**

- **Practical tips for rightsizing your Kubernetes workloads**
  - https://www.datadoghq.com/blog/rightsize-kubernetes-workloads/

  and
- **Kubernetes CPU limits and requests: A deep dive**
  - https://www.datadoghq.com/blog/kubernetes-cpu-requests-limits/

- CPU Limits and Throttling explained: "**Using Prometheus to Avoid Disasters with Kubernetes CPU Limits**"
  - https://aws.amazon.com/blogs/containers/using-prometheus-to-avoid-disasters-with-kubernetes-cpu-limits/

- "**CPU limits and aggressive throttling in Kubernetes**" by Fayiz Musthafa (omio.com), 2020
  - https://medium.com/omio-engineering/cpu-limits-and-aggressive-throttling-in-kubernetes-c5b20bd8a718
  - A detailed exploration, covering multithreading too, primarily focused on cgroups v1. While the implementation details have evolved, the fundamental concepts remain relevant today.

Sources: Delving into… **Must/Strongly Recommended**

A deep diving journey from Dave Chiluk, a Linux Kernel Contributor, exploring CPU Throttling from High Level Overview thought and till Kernel Code, Bugs and Fixes, Recommendations:

- "**Unthrottled:…**" series
  - p1: https://medium.com/indeed-engineering/unthrottled-fixing-cpu-limits-in-the-cloud-a0995ede8e89
  - p2: https://medium.com/indeed-engineering/unthrottled-how-a-valid-fix-becomes-a-regression-f61eabb2fbd9

- + accompanied by excellent, famous because often quoted, VIDEO from KubeCon 2019, on the same topic but which much more deeper insides and explanations: "**Throttling: New Developments in Application Performance with CPU Limits - Dave Chiluk, Indeed**"
  - CNCF video: https://youtu.be/UE7QX98-kO0
  - slides: https://static.sched.com/hosted_files/kccncna19/dd/Kubecon_%20Throttling.pdf

Sources: Delving into… **Recommended conference**

“Understanding and Measuring CPU Throttling in Containerized Environments”.

- This video explores CPU throttling in containerized environments, focusing on Kubernetes and Linux cgroups. Francesco Fabbrizio explains its performance impact, how to measure it with metrics like throttled_usec and additional helper Prometheus metrics, provides practical tips for optimizing resource allocation. It covers multi-threaded workloads, CPU limits, and fine-tuning practices to enhance production stability and avoid bottlenecks.
    - https://youtu.be/beWXWEimTxs

Sources: Delving into… **Must Read**

A series of posts by Daniele Polencic (Kubernetes consultant and teacher, learnk8s.io) exploring CPU management in Kubernetes, throttling, and recommendations. The content spans from foundational concepts to Linux Kernel insights, examining impacts on multithreading workloads, with an evolving perspective on CPU Limits:

- **“CPU Limits and Requests in Kubernetes”**
  - https://itnext.io/cpu-limits-and-requests-in-kubernetes-fa9d55948b7c
  - + complementary X/tweet on the same article but with more explanations, pictures and discussions: https://x.com/danielepolencic/status/1632717409051181056
- **“😵‍💫 Kubernetes CPU limits are not intuitive at all.”**
  - https://www.linkedin.com/posts/danielepolencic_cpu-limits-in-kubernetes-activity-7269343849582387200-367J
- **“Challenge 16: Throttled”**
  - https://medium.com/@danielepolencic/challenge-16-throttled-93133f8fd0ad

Sources: **Further/Recommended Readings**

A “***Kubernetes resources under the hood***” series, from basics to advanced, from Shon Lev-Ran and Shir Monether (both from SlimStack.io):

- **P#1**: “I’m sure we’re all familiar with the ‘resources’ block of containers in a pod. But do we really know what Kubernetes uses them for under the hood?”
  - https://medium.com/directeam/kubernetes-resources-under-the-hood-part-1-4f2400b6bb96
- **P#2**: “Do you think that CPU requests are just used for scheduling? Think again. Introducing CPU Shares, and laying the grounds for removing your limits!”
  - https://www.linkedin.com/posts/danielepolencic_cpu-limits-in-kubernetes-activity-7269343849582387200-367J
- **P#3**: “Kubernetes resources, breaking the limits! Understand the biggest Kubernetes misunderstanding and why you should remove your CPU limits and unleash your cluster's full potential”
  - https://medium.com/directeam/kubernetes-resources-under-the-hood-part-3-6ee7d6015965

Sources: **Kernel** - references and advanced topics

- Linux Kernel **CFS** – “**Completely Fair Scheduler**”
  - https://docs.kernel.org/scheduler/sched-design-CFS.html
- **CFS Bandwidth Control** (BWC)
  - https://docs.kernel.org/scheduler/sched-bwc.html
  - incl. “Burst feature”, allowing transferring part of unused quote for future periods use: https://docs.kernel.org/scheduler/sched-bwc.html#burst-feature
- “**CFS Group Scheduling**” - detailed explanation, from Hi to Lowest levels, from Imran Khan - an Oracle Linux Maintainer.
  - https://blogs.oracle.com/linux/post/cfs-group-scheduling
- **Red Hat Docs** on CPU Sharing Concepts (inclusive “borrowing” cycles from others reserved if not used)
  - https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/resource_management_guide/sec-cpu#sect-cfs
  - Even if it covers RHEL v6 and cgroups v1, concepts are explained in detail and are generally applied as for now.

Sources: **Advanced topics**

- K8s "**Control CPU Management Policies on the Node**", allowing assigning some dedicated vCPU sets to specific pods.
  - Official DOCs: https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/
  - K8s blog posts:
    - https://kubernetes.io/blog/2024/12/16/cpumanager-strict-cpu-reservation/
    - https://kubernetes.io/blog/2024/08/22/cpumanager-static-policy-distributed-cpu-across-cores/
    - https://kubernetes.io/blog/2022/12/27/cpumanager-ga/
    - https://kubernetes.io/blog/2018/07/24/feature-highlight-cpu-manager/

Advanced: Kernel CFS BWC "Burst Feature" in K8s

Below are series of references from Alibaba Cloud which implemented the Kernel CPU CFS BWC Feature in their custom K8s.

- User Guide: "**Enable CPU Burst**"
  - https://www.alibabacloud.com/help/en/ack/ack-managed-and-ack-dedicated/user-guide/cpu-burst
- Blog with analytics and mathematical researches:
  - "Kill the Annoying CPU Throttling and Make Containers Run Faster"
    - https://www.alibabacloud.com/blog/kill-the-annoying-cpu-throttling-and-make-containers-run-faster_598738
  - "Does CPU Burst Have Side Effects? Let the Math Answer!"
    - https://www.alibabacloud.com/blog/598745
  - "Review Probability Theory through CPU Burst"
    - https://www.alibabacloud.com/blog/review-probability-theory-through-cpu-burst_598755

Advanced/Future Topics: Auto-scalers

Autoscaling Tools Overview

- **HPA**: Scales pod replicas based on metrics like CPU/memory utilization.
  - https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
- **VPA**: Adjusts pod resource requests/limits dynamically.
  - https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler
- **KEDA**: Enables event-driven pod scaling based on external event sources.
  - https://keda.sh/
- **Karpenter**: Provisions and manages cluster nodes for optimal resource utilization.
  - https://karpenter.sh/docs/
- **AWS Fargate**: Serverless compute engine for running containers without managing infrastructure.
  - https://aws.amazon.com/fargate/

Advanced/Future Topics: Auto-scalers

Integrating Autoscaling Tools

- **HPA & VPA:** Combine for comprehensive pod scaling; use complementary metrics to avoid conflicts.
- **KEDA & HPA:** Use KEDA to provide event metrics to HPA for event-driven scaling.
- **Karpenter & AWS Fargate:** Leverage Karpenter for dynamic node provisioning alongside Fargate for serverless workloads, optimizing cost and performance.

By understanding and combining these tools, teams can create a flexible, efficient, and responsive Kubernetes environment tailored to their specific workload demands.

Closing Thoughts

- *… no more thoughts …*

Placeholder

- Text1
- This is The End.