ASML

A cloud cost saving journey: Strategies to balance CPU for containerized JAVA workloads in K8s

Laurentiu Marinescu
Ajith Ganesan

Cloud costs: The risking risk no one can ignore

Cloud Budgets Are Skyrocketing

Global spend from **$595.7B to $723.4B in 2025**

#1 Challenge: Controlling cloud spend

“Controlling cloud costs” outranks security and compliance as the #1 cloud concern

17% over budget

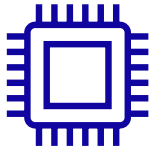
Forecasts often fail, causing financial risk

Source 1: Flexera: State of the cloud report 2025

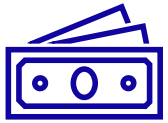
Source 2: Gartner cloud spending 2024 report

Cloud resources: [illegible] idle, $43 Billion Lost

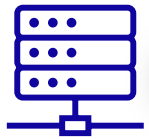
More CPUs, more cloud.. still more waste



[illegible] **CPU Utilization** in Kubernetes clusters



$43.3B wasted

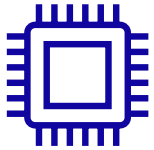


[illegible] **CPU Utilization** even in 1000+ CPU clusters

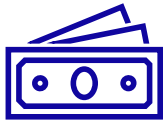
ASML

Cloud resources: 90% idle, $43 Billion Lost

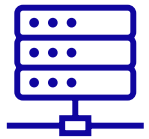
More CPUs, more cloud.. still more waste



10% CPU Utilization in Kubernetes clusters



$43.3B wasted by enterprises



17% CPU Utilization even in 1000+ CPU clusters

ASML

Imagine the possibilities!

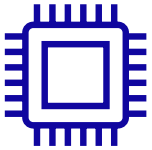
Big challenge, bigger Opportunity.

What if we could reclaim **even a fraction of that $43B?**

What if 90% idle **became 50% or even 30%?**

Let’s look ahead and take control.

Our mission today



Goal 1: CPU-optimization design strategies

Equip you to think about **balanced design to promote utilization** based on carefully analyzed performance and availability requirements

Goal 2: Development best practices

Equip you to measure, diagnose and right size CPU requests and limits, and spot-node tactics to maximize utilization and slash idle spend in Kubernetes clusters.

Development best practices for Java/Spring-on-K8s.

Who are we?

Laurentiu Marinescu

- Full stack sw engineer, problem solver
- Passionate about software craftsmanship, new tech
- Advocate of pair/team programming.
- Bouldering/Climbing lover

Ajith Ganesan

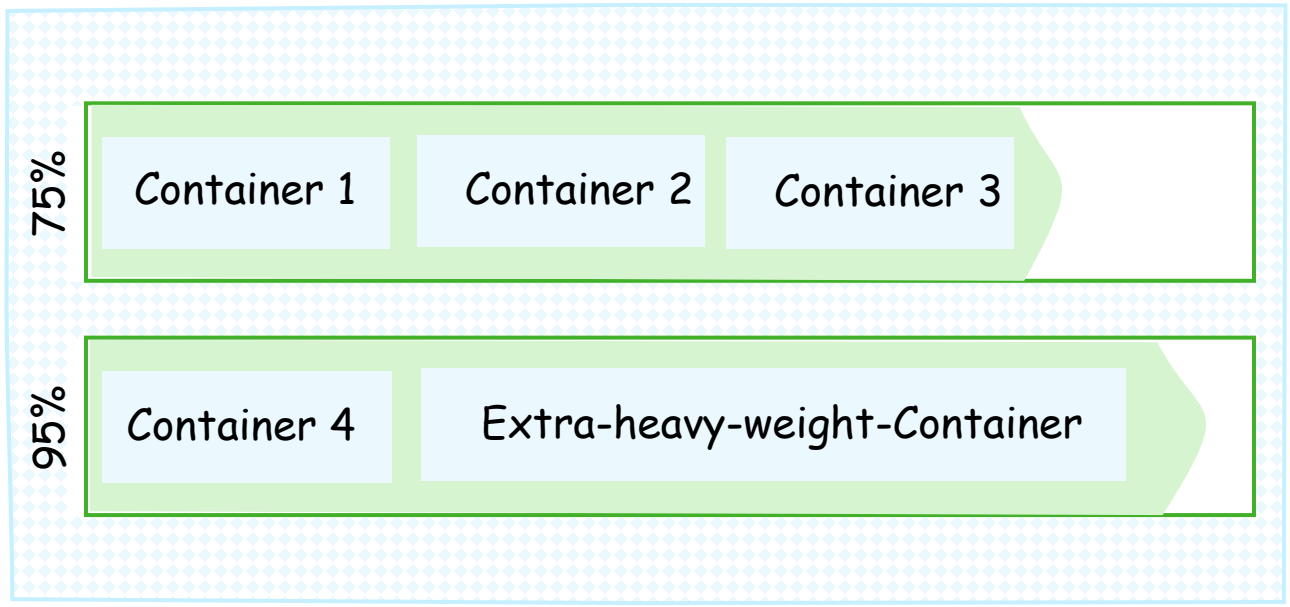
- Systems engineer, Data platform strategy
- Passionate about tech, Micro SaaS, AI exploration
- Cricket/Cooking/Movies

Revisit the storyline so far…

- Companies spend a lot on cloud budget
- #1 priority to control cloud budget
- 17% over-budget
- But still utilization is only ~10% on average, why?

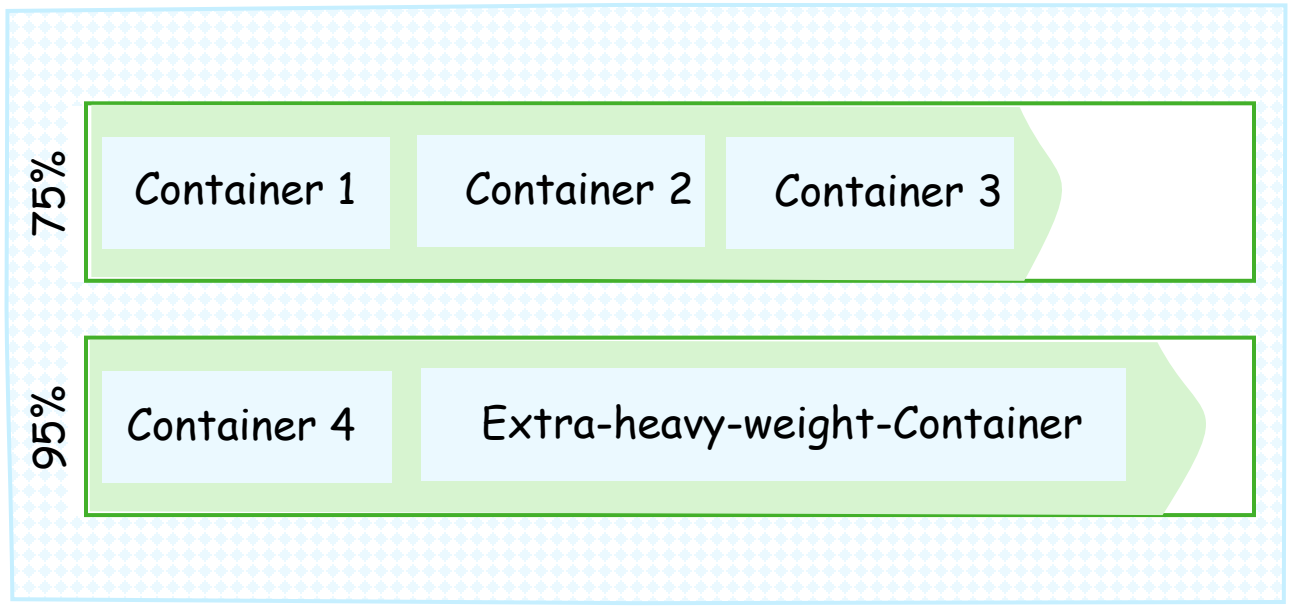
Simulated example of a cluster

2 nodes with 5 containers with very high utilization

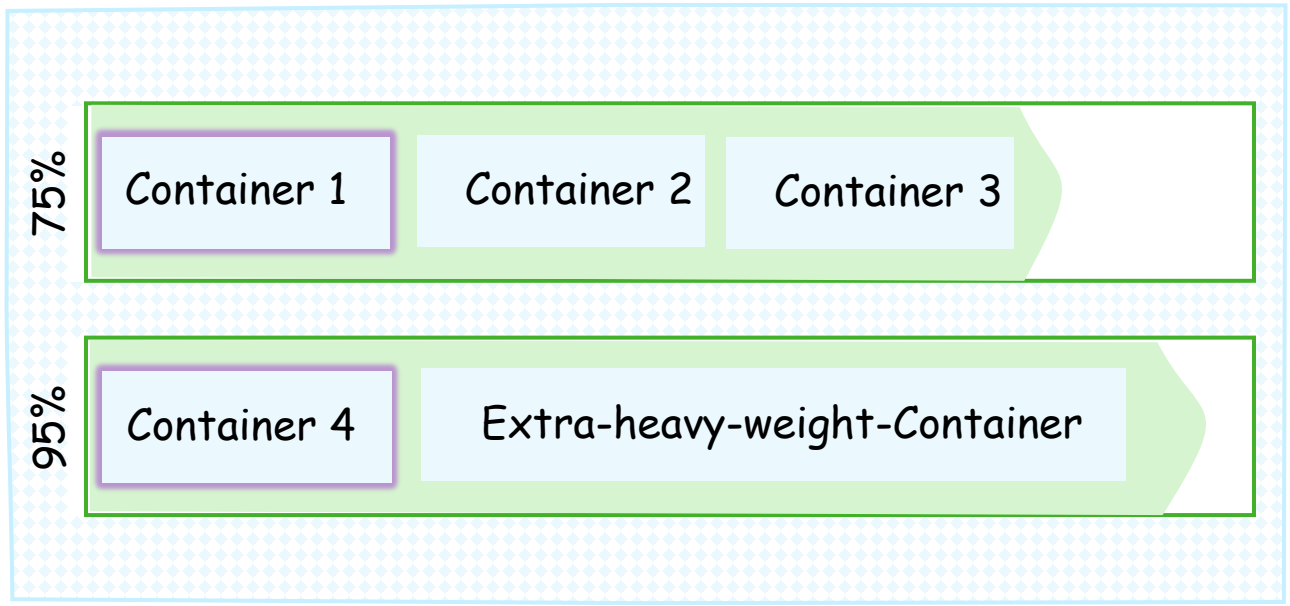


Simulated example of a cluster

2 nodes with 5 containers with very high utilization but at what cost?



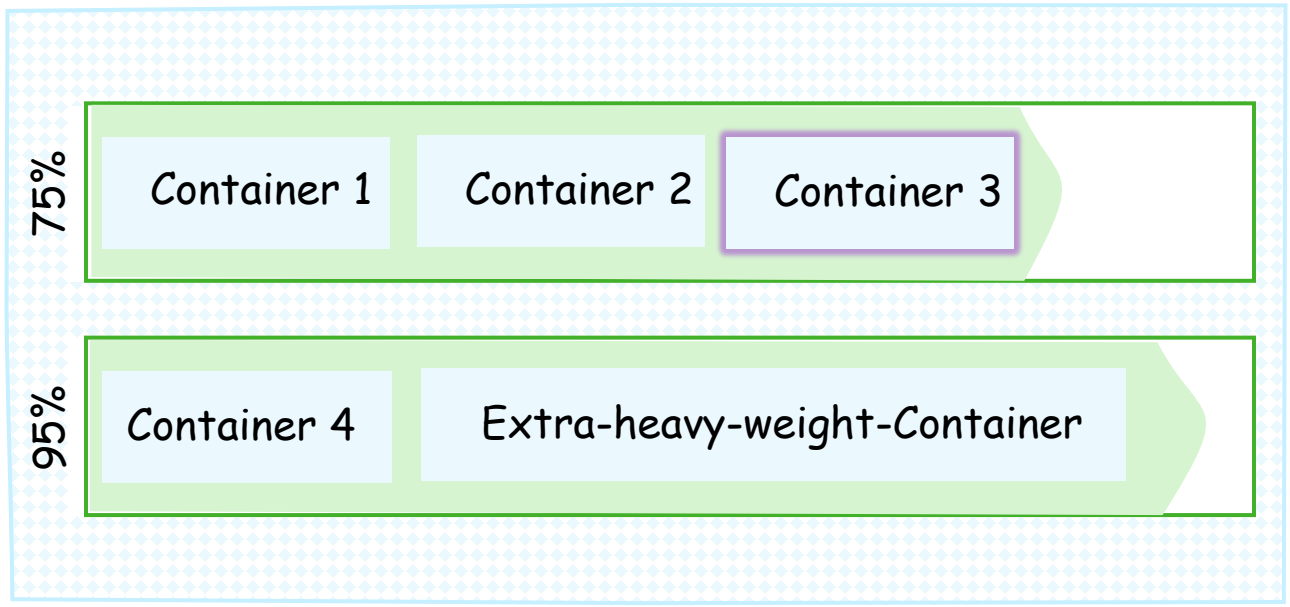
Scenario 1: Upgrades



Scenario 1: Upgrades

- **Containers 1 and 4 require sequential upgrades**
- **Zero-downtime upgrade is not possible** for the **Extra-heavy-weight-container**

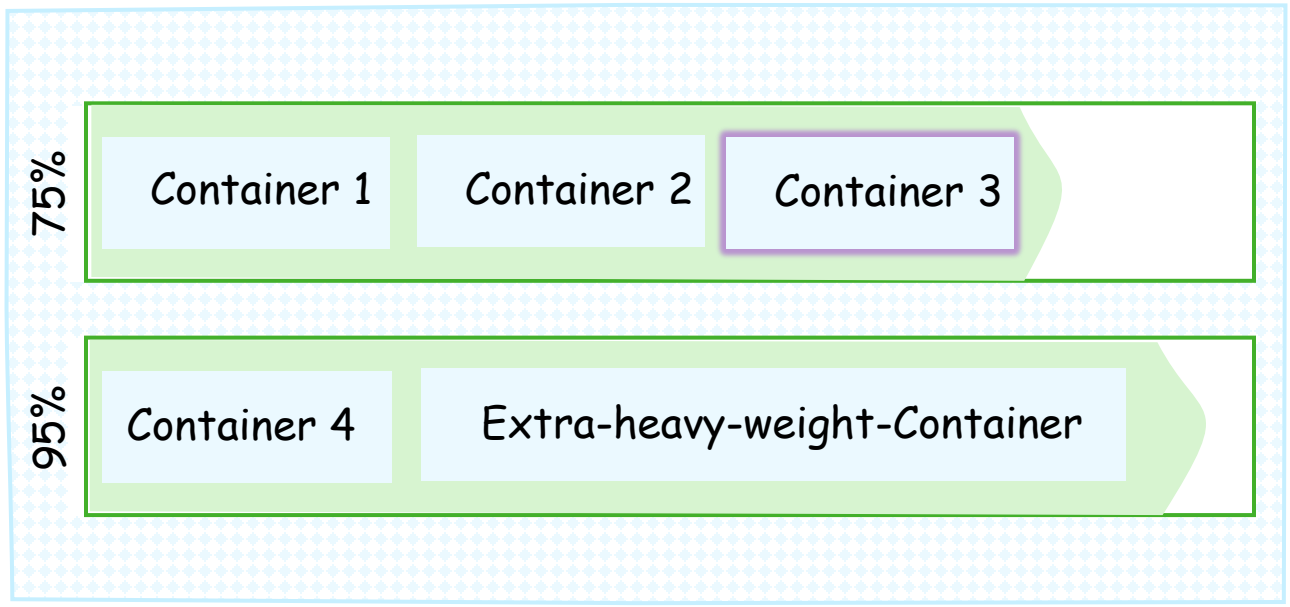
Scenario 2: Scaling



Scenario 2: Scaling

On-demand scaling of container 3 will fail due to resource exhaustion

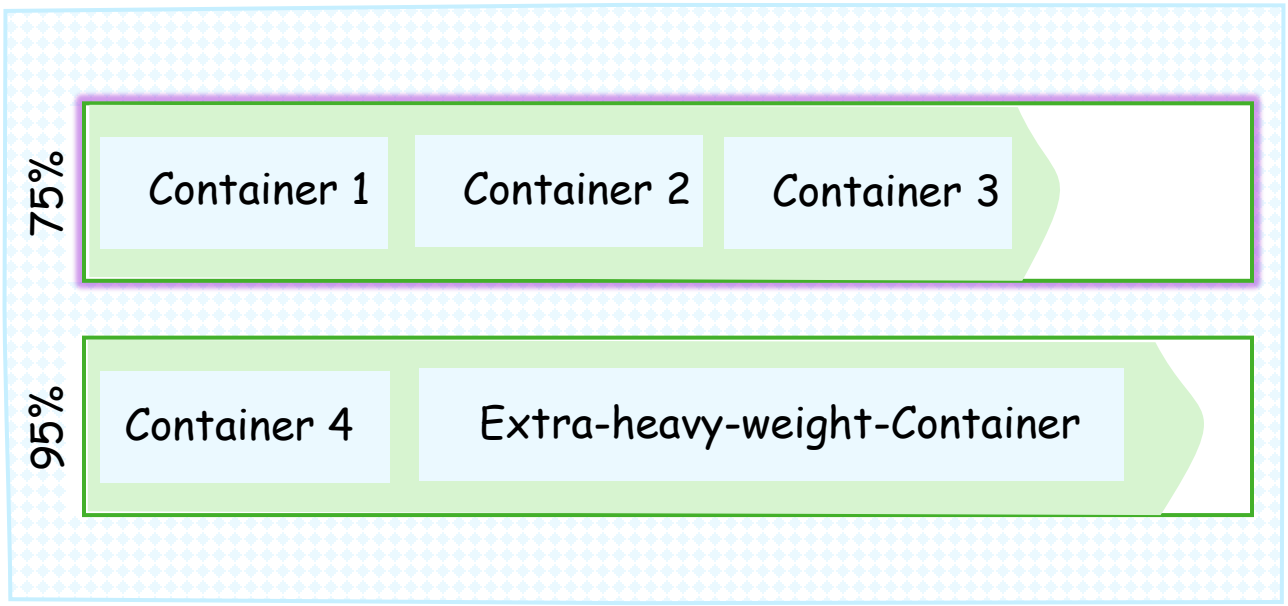
Scenario 3: Accommodating app crash



Scenario 3: Accommodating app crash

When scaled-out containers crash, insufficient capacity can prevent/slow recovery, making remediation efforts slow or infeasible

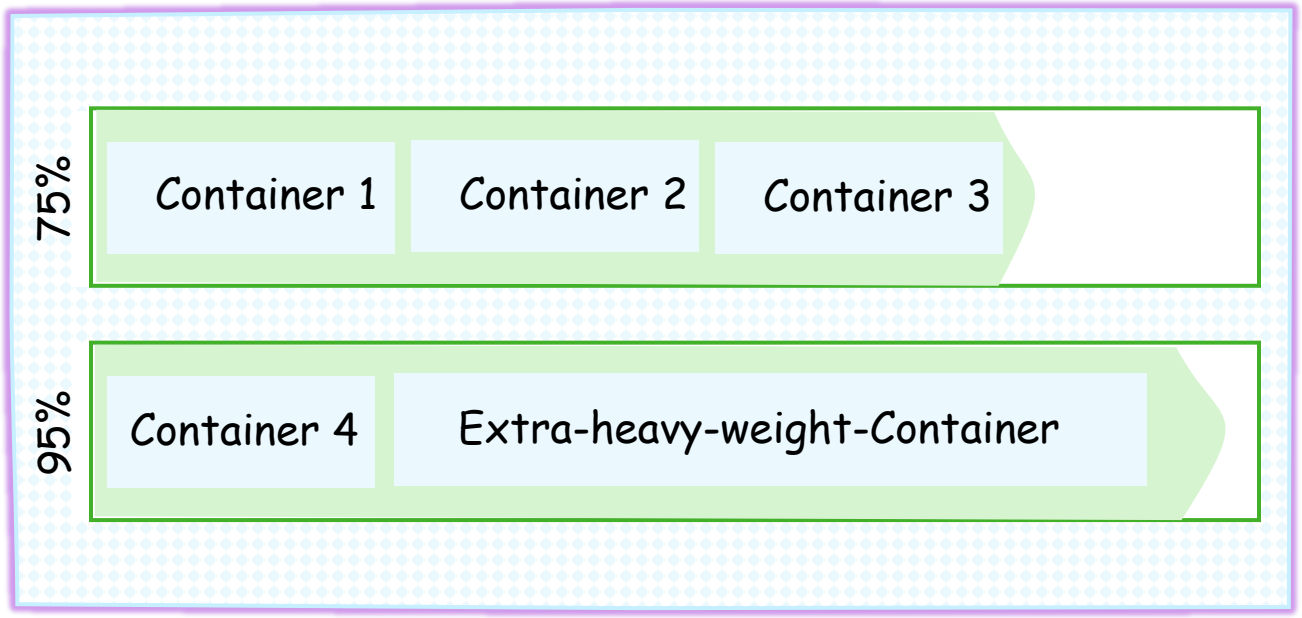
Scenario 4: Node failures



Scenario 4: Node failures

Failure of Node 1 will result in non-deployment of Container 1, 2 and 3 assuming equal priorities without eviction policies

Scenario 5: Zone failure

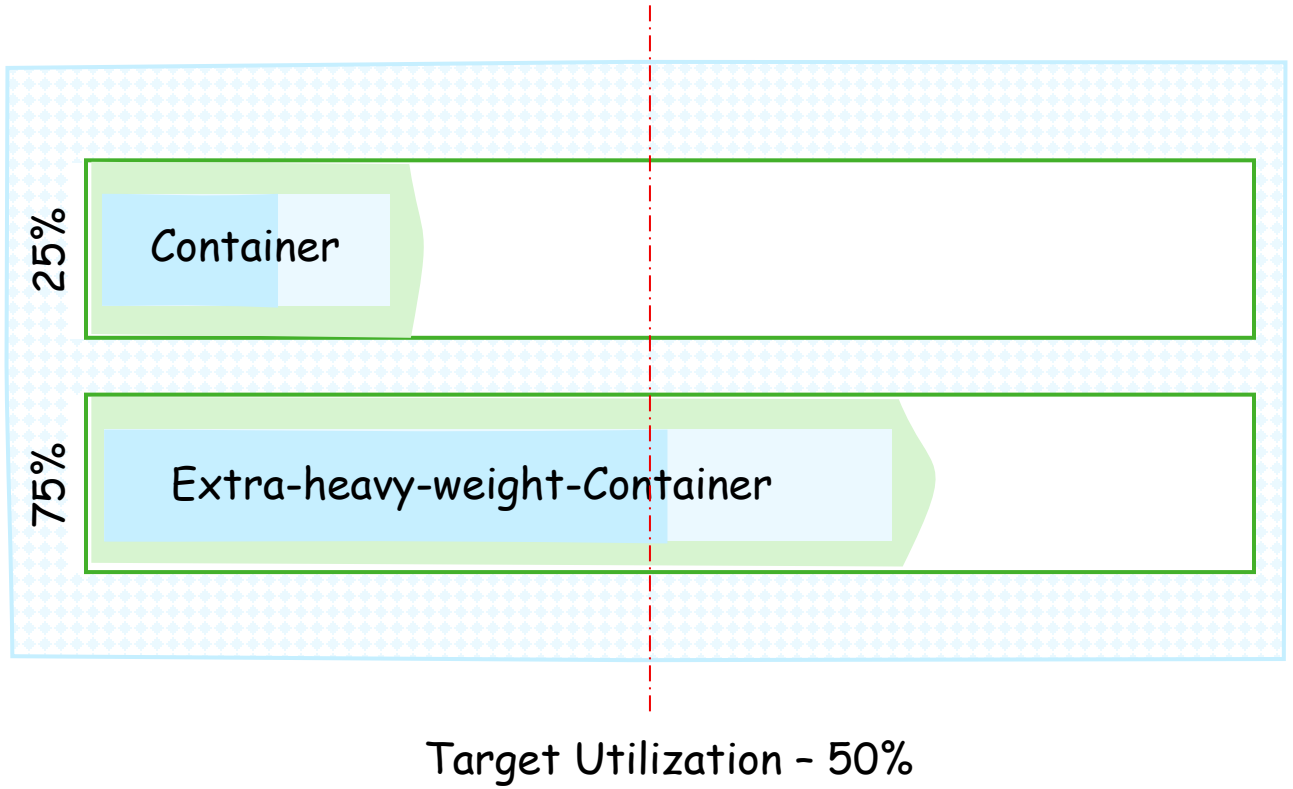


Scenario 5: Zone failure

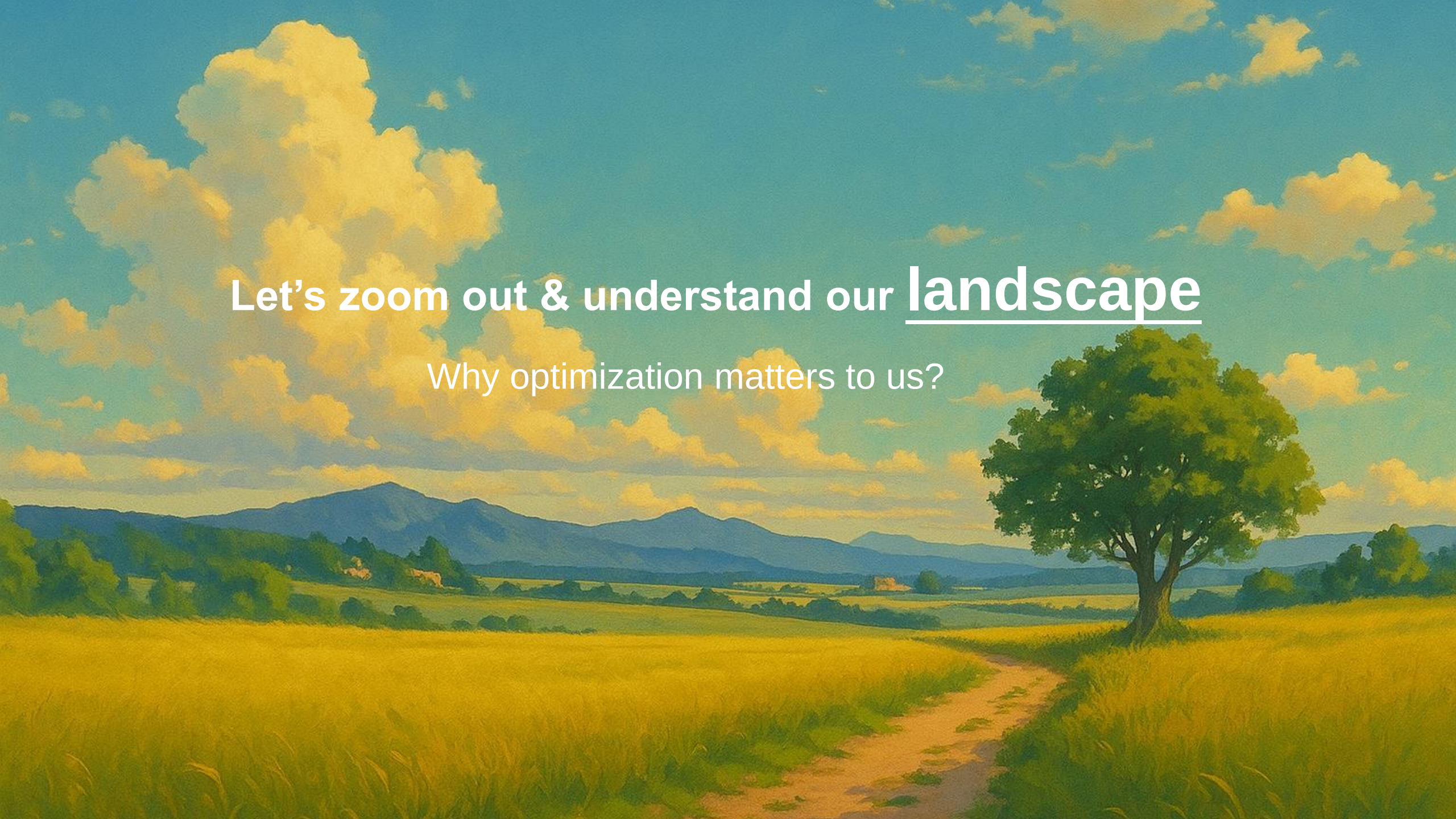
Zone failure will result in total availability loss if node 1 and 2 are connected to same zones

Maximizing ~~utilization~~ efficiency is a balancing act

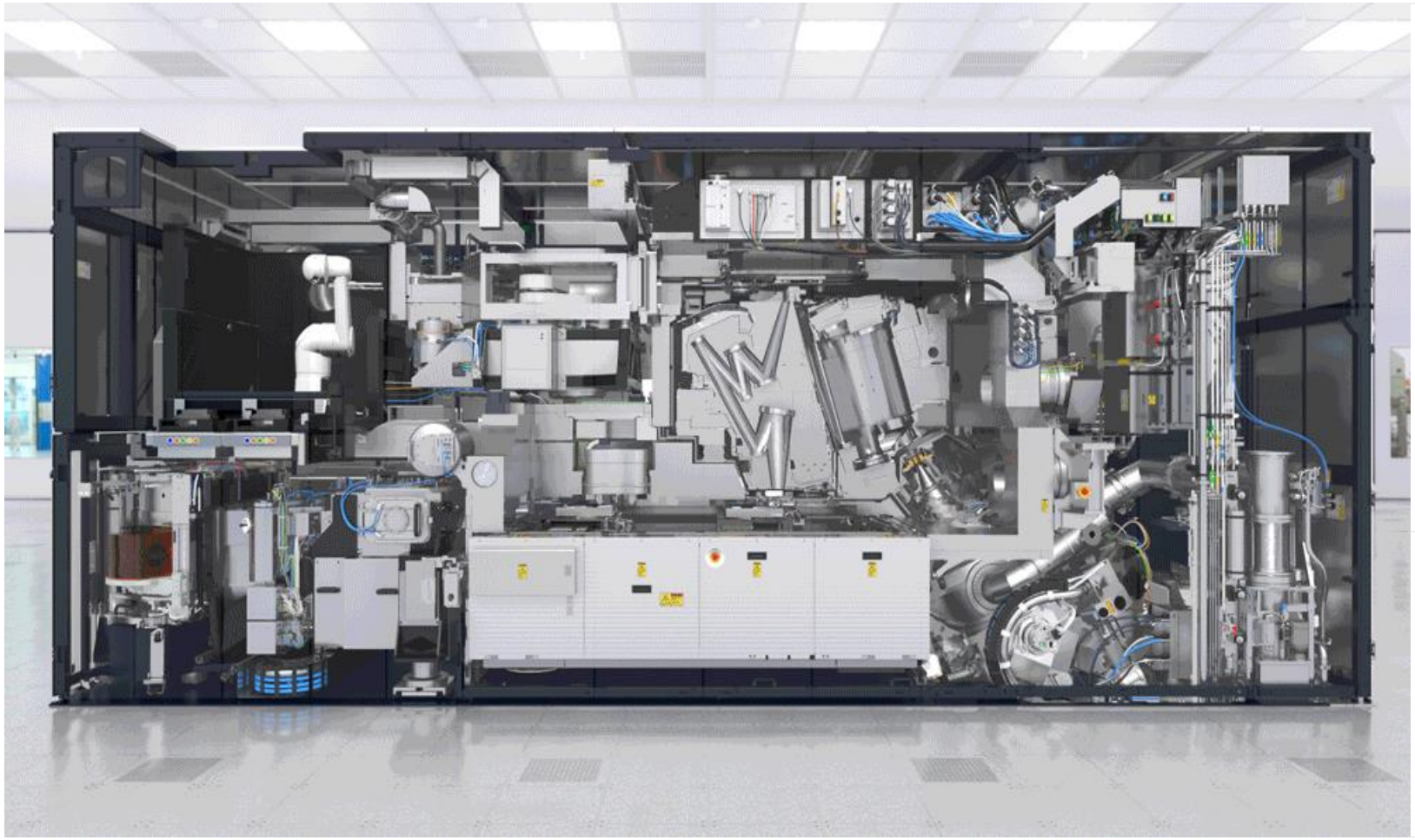
Even targeting 50% of utilization will not guarantee upgrade activities during a node down



Efficiency: capacity of the system to perform its designated functions in an optimized way with the given resources (under stated conditions).


Let’s zoom out & understand our landscape
Why optimization matters to us?

Our Data Platform: Enabling Misson Critical Lithography Applications



Complex process with nanometer precision

Process variations corrected by applications integrated into control loops

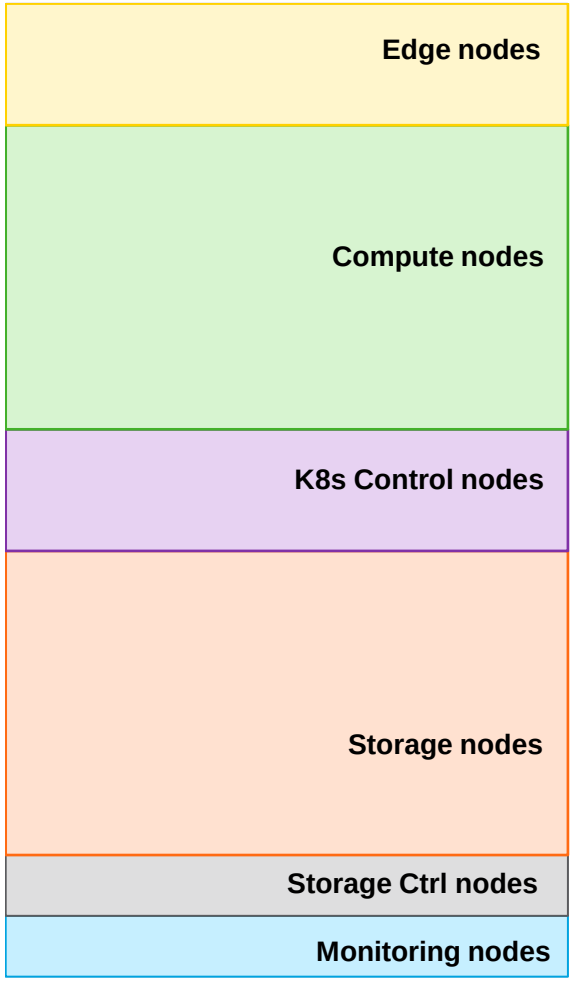
Data platform to host the mission-critical applications

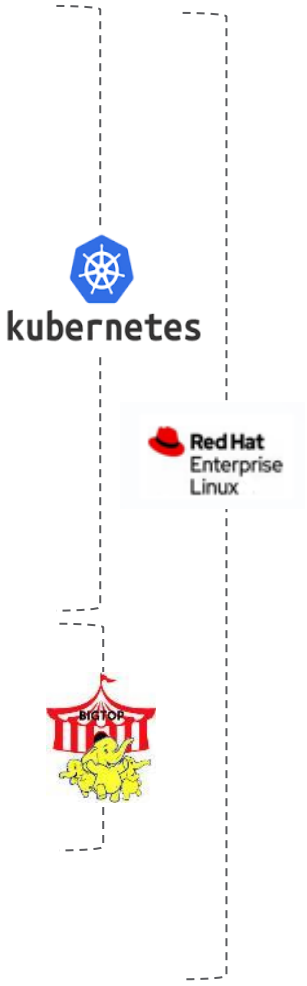
Managing a diverse technology stack to power mission-critical applications

Our data platform cluster topology

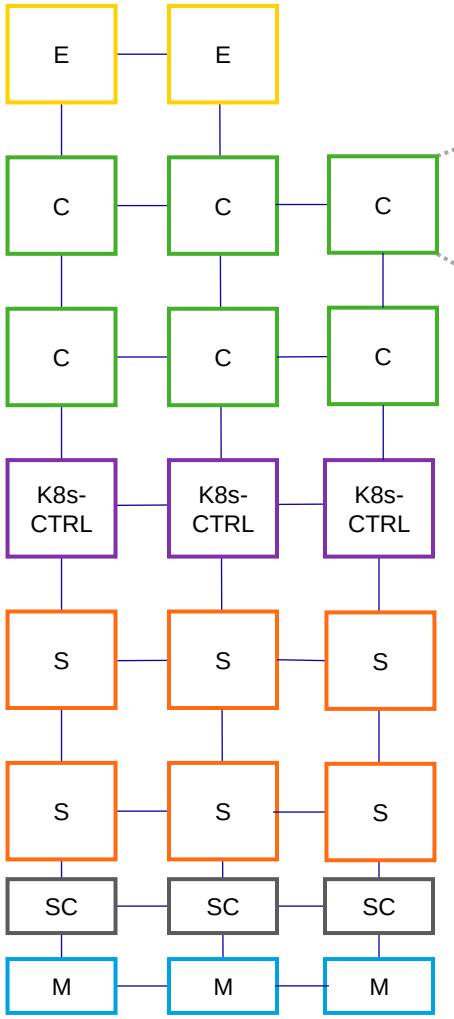
How we organize our platform

Example distribution of applications to node groups

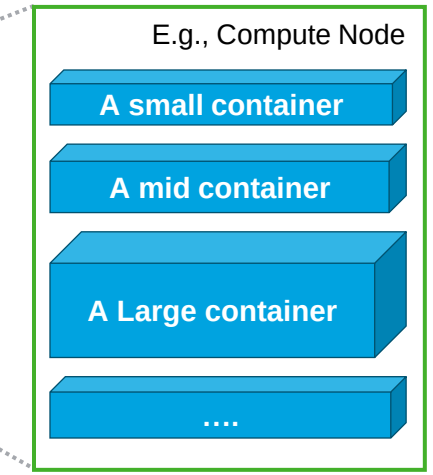




Example distribution of node groups



Example distribution of containers in a compute node



ASML

Key challenges and observations in our data platform

1. Diverse workloads & SLOs

Applications with varying service level objectives increase complexity

2. Critical & Non-critical applications co-exist

Critical & Non-critical applications share the same infrastructure

3. Misson-Critical demands

Downtime of even 10 minutes can lead to substantial losses

4. On-prem Hardware Constraints

Customer premise Hardware leads to long lead times for scaling (> 6 months)

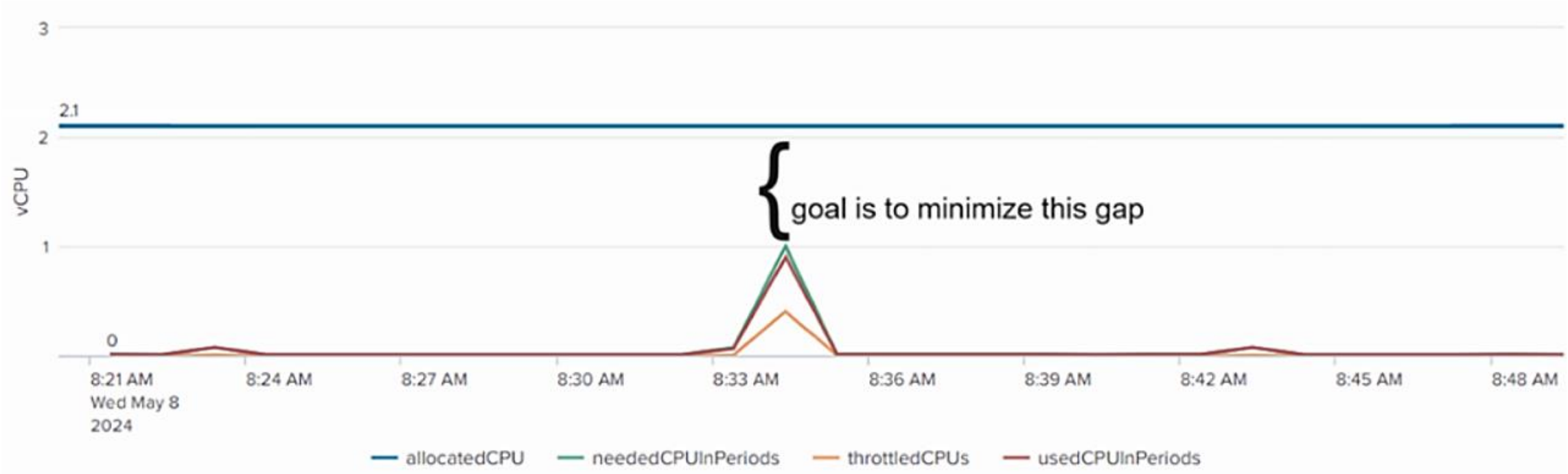
5. Underutilization of resources

Many environments remain underutilized

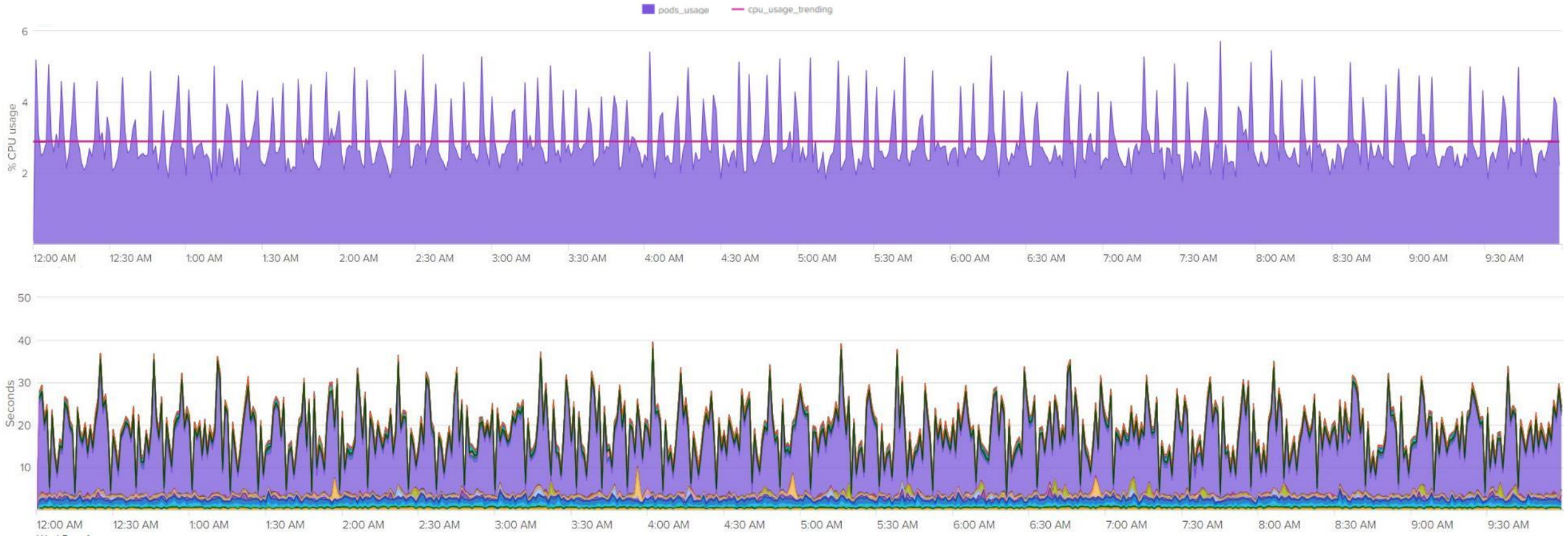
Different types of workloads

- Applications are **not constantly executed, and containers have different execution patterns**.
- Applications are assigned to nodes, and **not all containers peak at the same time.**
- Applications exhibit **non-deterministic behavior**, with some being CPU-intensive, others memory-intensive, and some I/O-intensive, making resource management complex.

Observation 1: Underutilization

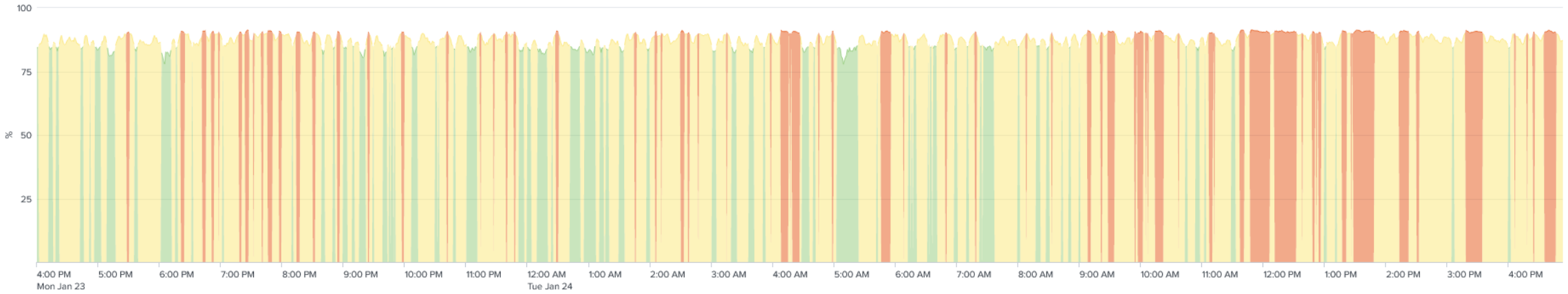


Observation 2: Low performance due to throttling



ASML

Observation 3: App saturation



ASML

Main bottlenecks on our Java Spring Boot apps

Blocking I/O Operations

Slow database queries

Frequent or long GC pauses

Thread contention (locks)

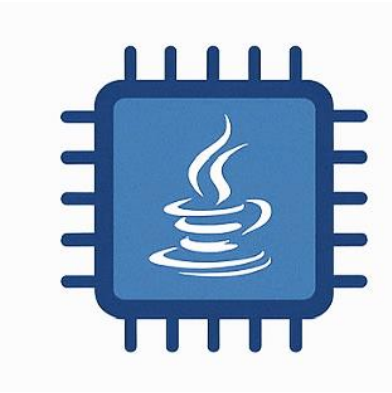
ASML

Java applications are CPU-hungry

Multithreaded

CPU starvation: When a container doesn't get enough CPU, leading to slow performance and timeouts.

CPU overcommit: When a container uses more CPU than allocated, causing throttling and potential node crashes.



ASML

CPU structure in Linux Systems

How Linux counts CPUs

$$\mathbf{CPU}(s) = \mathbf{Thread}(\mathbf{s})\ \mathbf{per\ core} * \mathbf{Core}(\mathbf{s})\ \mathbf{per\ socket} * \mathbf{Socket}\ (\mathbf{s})$$

```
cpu $ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
CPU(s):                  16
On-line CPU(s) list:     0-15
Thread(s) per core:      1
Core(s) per socket:      1
Socket(s):               16
```

Figure A CPU configuration overview sample.

```
cpu $ lscpu -e
CPU NODE SOCKET CORE L1d:L1i:L2:L3 ONLINE
0   0    0      0    0:0:0:0       yes
1   0    1      1    1:1:1:1       yes
2   0    2      2    2:2:2:2       yes
3   0    3      3    3:3:3:3       yes
4   0    4      4    4:4:4:4       yes
5   0    5      5    5:5:5:5       yes
6   0    6      6    6:6:6:6       yes
7   0    7      7    7:7:7:7       yes
8   0    8      8    8:8:8:8       yes
9   0    9      9    9:9:9:9       yes
10  0    10     10   10:10:10:10   yes
11  0    11     11   11:11:11:11   yes
12  0    12     12   12:12:12:12   yes
13  0    13     13   13:13:13:13   yes
14  0    14     14   14:14:14:14   yes
15  0    15     15   15:15:15:15   yes
```

Figure B. CPU layout mapping sample

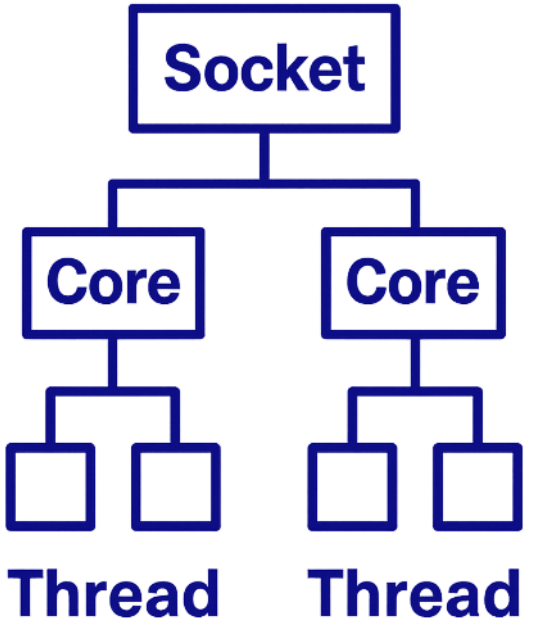


Figure C. CPU structure and relation

Think in time: CPU usage as time slices

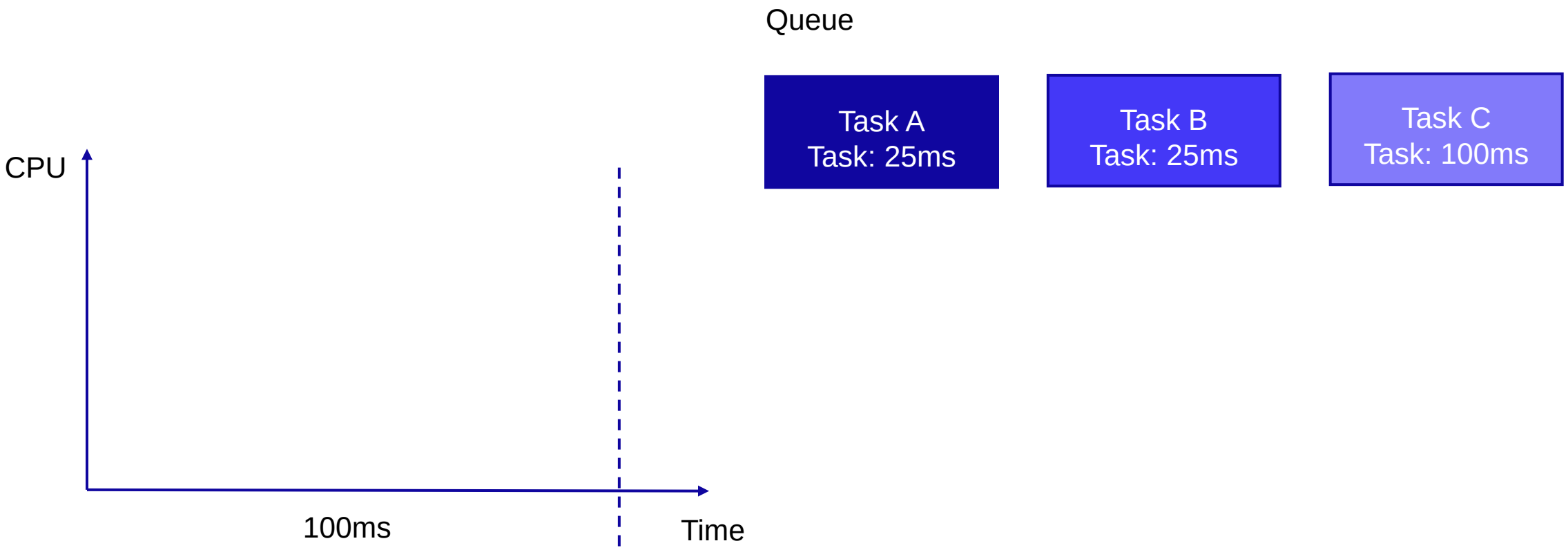
Your Java App does not get CPU, it gets a time slot

Scheduling CPU Time: Traffic Control for Processes

Schedulers decide which process gets CPU time and when.

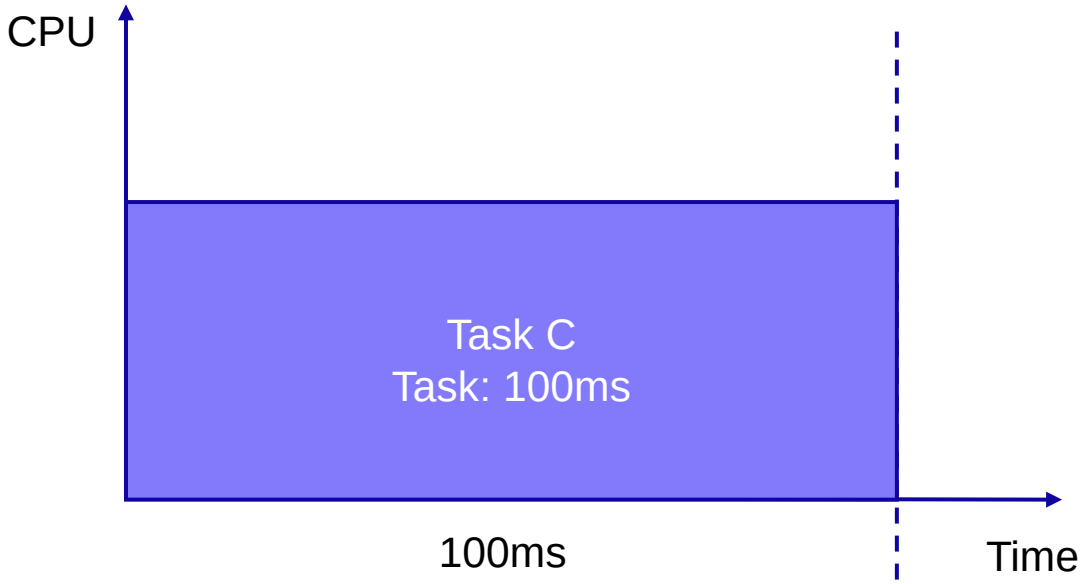
Dividing CPU time with Completely Fair Scheduler

Simulated example (1/3)



Dividing CPU time with Completely Fair Scheduler

Simulated example (2/3)



Queue

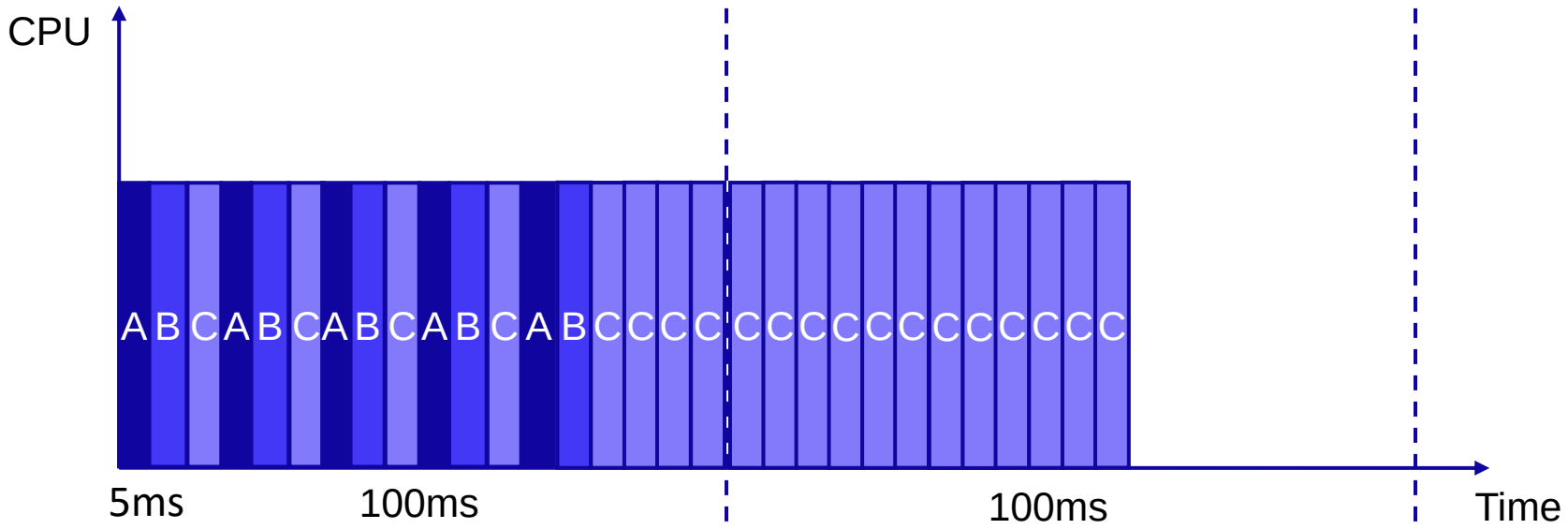
Task A
Task: 25ms

Task B
Task: 25ms

Scheduling Task C will lapse all the available CPUs,
In this case, Task A and B do not get CPU time

Dividing CPU time with Completely Fair Scheduler

Simulated example (3/3)



Smallest task scheduled first

Each task gets fair share of the time

Controlling CPU time with Kubernetes

CPU requests

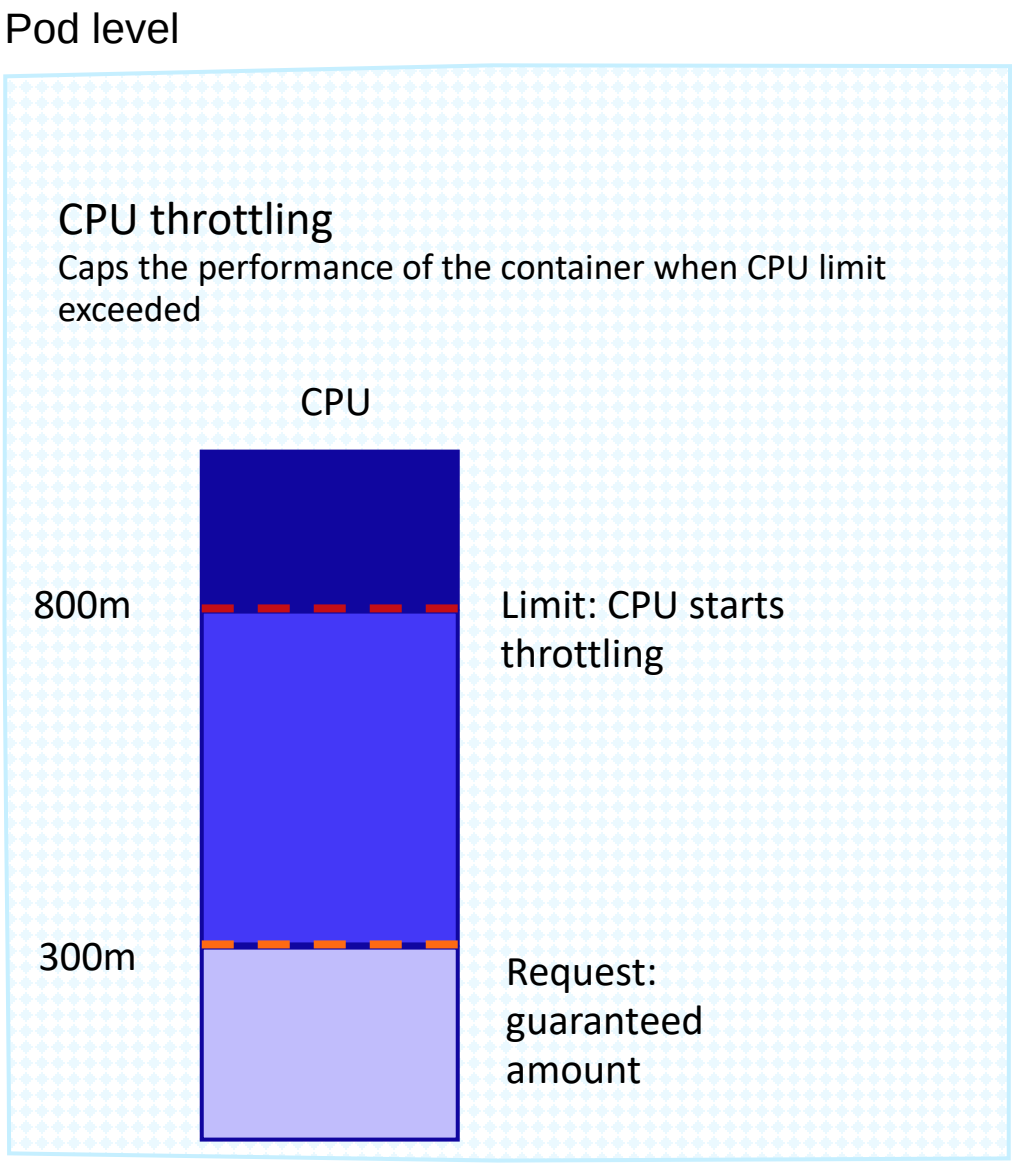
A container is guaranteed to be allocated the CPU requested.

- K8s uses this value to place the container in a node that fulfills this resource claim and be guaranteed.
- **Host CPU relative weight.**

CPU limits

A container cannot use more than configured limit.

- After this value, CPU throttling.
- **If no limit set, the application can consume all CPU in a node.**



ASML

How Kubernetes controls CPU time using Linux cgroups

		cgroup v1	cgroup v2
CPU request	→	cpu.shares (milliCPU * 1024) / 1000	cpu.weight
CPU limit	→	cpu.cfs_quota_us cpu.cfs_period_us (milliCPU * period) / 1000	cpu.max

cpu.shares CPU is allocated in shares (1 core = 1024 shares), default is 1024 shares.

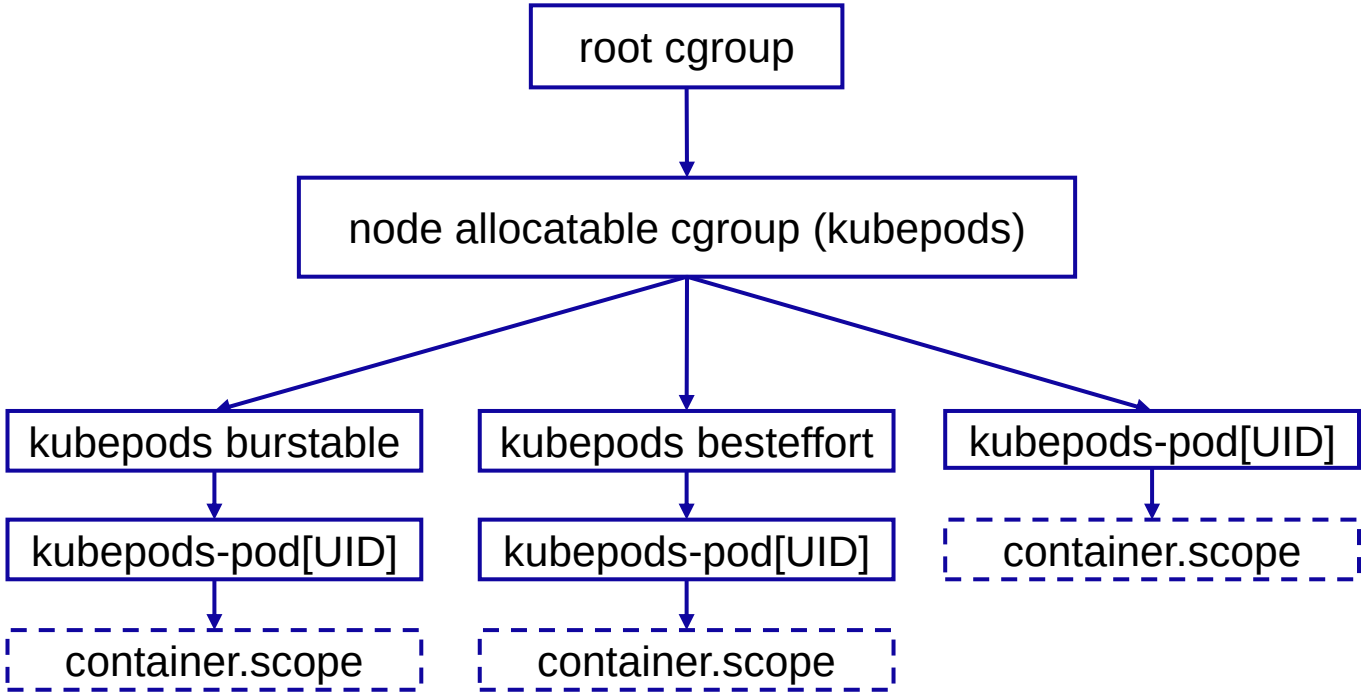
cpu.cfs_quota_us amount of CPU time that a process can consume over a specific time period.

cpu.cfs_period_us time window where CPU quota is enforced, measured in microseconds. (default 100,000us)

How Kubernetes maps Pods into Linux cgroup trees

- Dedicated cgroups for **burstable** QoS pods and **best effort** pods
- Guaranteed QoS pods compete, a **burstable** parent and **besteffort** parent

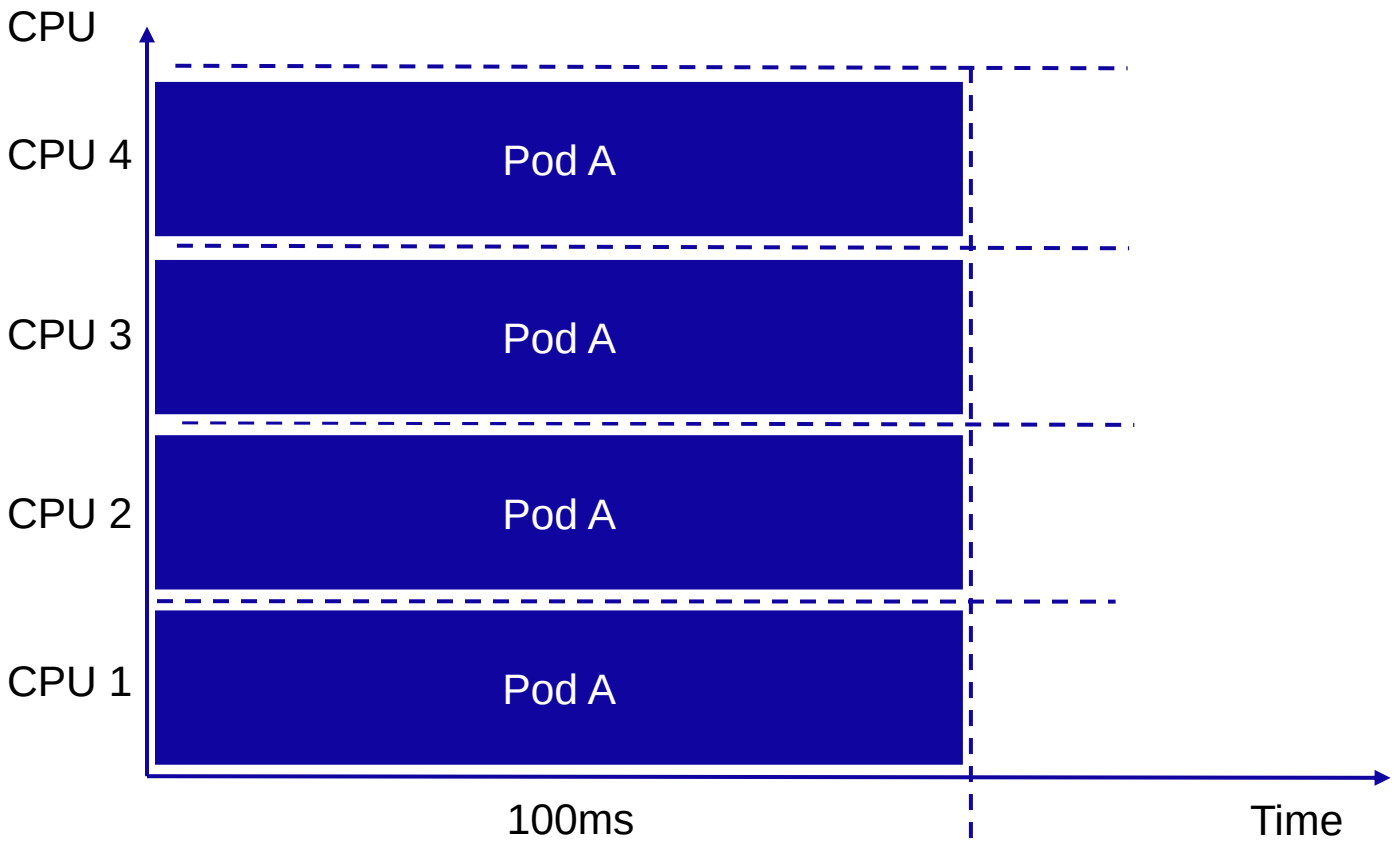
```
cgroup $ tree
.
├── kubepods.slice
│   ├── kubepods-besteffort.slice
│   │   └── kubepods-besteffort-poddd3c9702_8394_40cf_aaa9_beeba673b00f.slice
│   │       ├── cri-containerd-18bfe05e6f4ae7d02733cc8173a64888d944a401f214de0a5cb5af5877e12429.scope
│   │       └── cri-containerd-ad7a91ac58bacfeb44aa4fc98daf34929711845596f6062e30f9d1137269e21e.scope
│   ├── kubepods-burstable.slice
│   │   └── kubepods-burstable-podf8e35f44_5eeb_4212_a45f_85701c370c4f.slice
│   │       ├── cri-containerd-0876addf2f1f41d7ab06e6cd108ea54cc467dc67010bd706b564bfb6fd00ccd2.scope
│   │       └── cri-containerd-94475c8d752434a947916e84be64f44835e62c07ed8e6516361307b122953dca.scope
│   └── kubepods-podf8bf3c94_aed0_403b_b687_508c853cdca3.slice
├── system.slice
│   ├── kubelet.service
│   └── sshd.service
└── user-slice
```



Scenario 1: One pod requires 400ms of CPU time

4 x 100 ms usage time => 100 ms response time

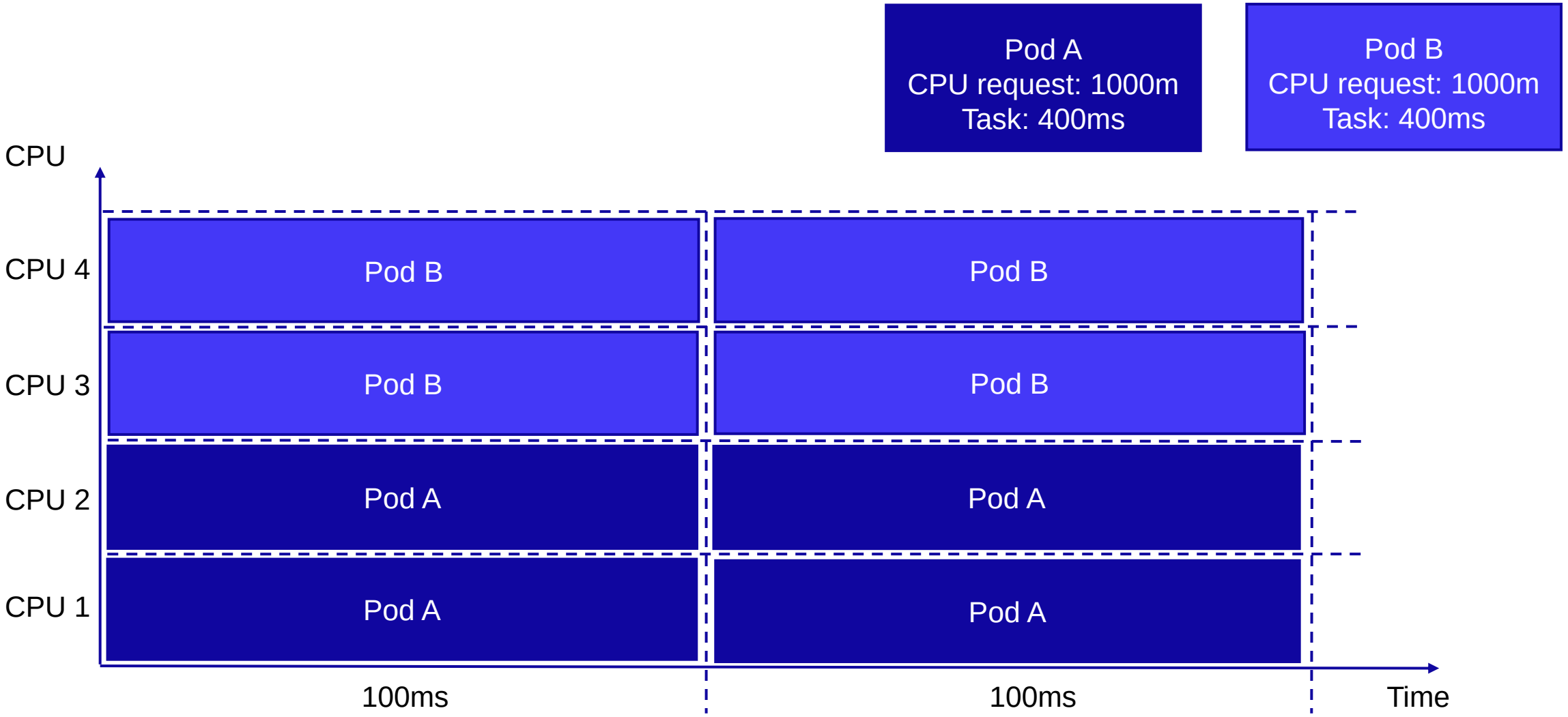
Pod A
CPU request: 1000m
Task: 400ms



ASML

Scenario 2: Two pods require 400ms of CPU time

Pod A and Pod B => 200 ms response time each



ASML

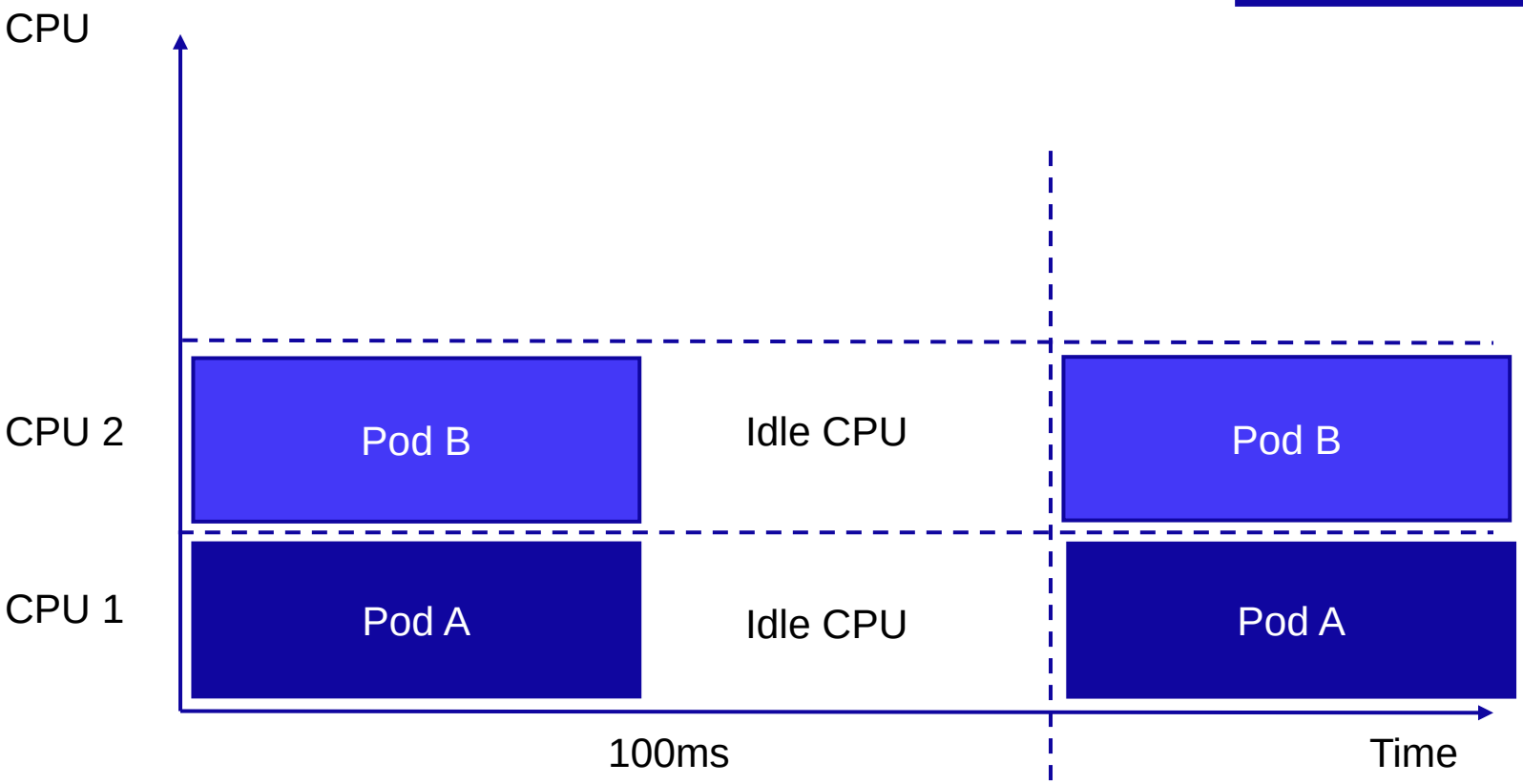
Scenario 3: Two pods with CPU limits

Pods have task execution higher than the limit

Pod A
CPU Limit: 500m
Task: 100ms

Pod B
CPU Limit: 500m
Task: 100ms

CPU limit 500m => 50ms => hard limit



ASML

HOW to solve it?

1. Understand the app specification and behavior

- ✓ Rely on App SLO’s, specific SLO’s
- # Number of threads used
- # Number of requests
- Response times
- Task duration

2. Set a baseline

- Ensure X% load on the nodes where container under change runs
- Agree on realistic worst-case use-cases
- At the beginning, ensure single instance of container is deployed (it eases the test execution and analysis)
- Warm-up java container

```
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource                    Requests      Limits
  --------                    --------      ------
  cpu                         9760m (61%)   27800m (173%)
  memory                      24686Mi (38%) 35338Mi (54%)
Name:               k8s-compute-node-4
```

3. Trigger a series of execution for one use-case (1/3)

Prometheus

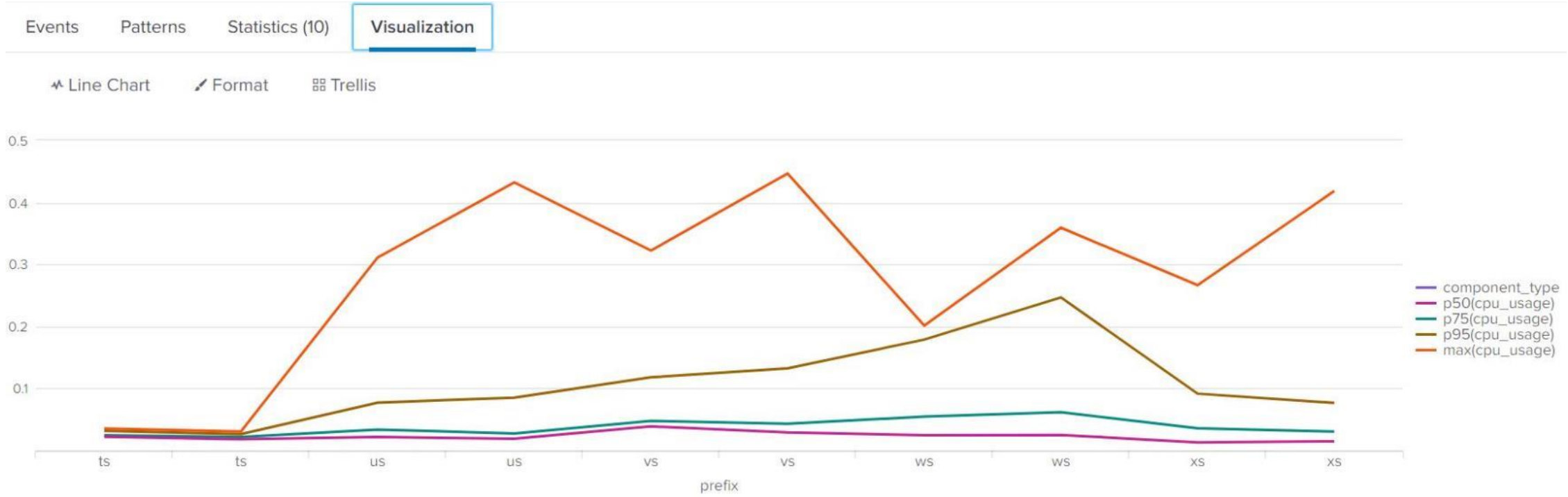
```
max(rate(container_cpu_usage_seconds_total{namespace="app-namespace",container=~"app-.+"}
[1m]))
```

1. **container_cpu_usage_seconds_total**: the total CPU time used across all cores of your container.
   It comes from the usage_usec field in cpu.stat.
2. **container_cpu_user_seconds_total** and **container_cpu_system_seconds_total** track time spent in user mode and kernel mode, pulled from user_usec and system_usec.
3. **container_cpu_cfs_periods_total** tells you how many 100ms CPU periods have passed.
   This comes from nr_periods.
4. **container_cpu_cfs_throttled_periods_total** counts how many of those periods had the container throttled. If your container got throttled during 30 out of 50 windows, this would be 30. It maps to nr_throttled.
5. **container_cpu_cfs_throttled_seconds_total** shows how much total time the container was throttled. If it got paused for 30ms in each of 10 periods, this would show 300000 microseconds (300ms). That's coming from throttled_usec.

3. Trigger a series of execution for one use-case (2/3)

```
| mstats rate_sum(container_cpu_usage_seconds_total) as cpu_usage where index=k8s_metrics container="app-*"
span=1min by pod,container
| rex field=container "app-(?<component_type>[\w]+)-(?<prefix>[a-z]+).*"
| stats p50(cpu_usage), p75(cpu_usage), p95(cpu_usage), max(cpu_usage) by prefix, component_type
```

Profile	*CPU-Request*	*CPU-Limit*
TS, WS	0.2	0.2
XS	0.2	1
US, VS (very high limit)	0.2	2



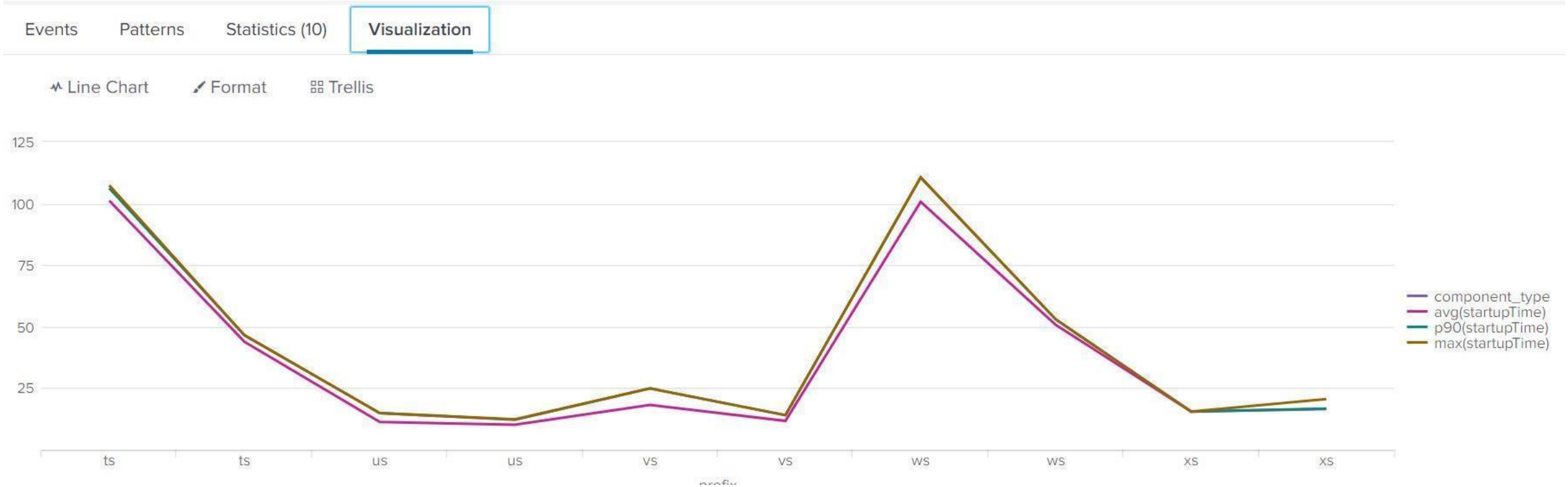
ASML

3. Trigger a series of execution for one use-case (3/3)

High impact on startup time for java apps!

From 100+s => 15s

Profile	*CPU-Request*	*CPU-Limit*
TS, WS	0.2	0.2
XS	0.2	1
US, VS (very high limit)	0.2	2



4. Allocate the needed CPU

1. Restart the container with adjusted values
2. Verify if impact on KPI's and SLA's
3. If there is impact rerun by using binary search
4. If there is no impact then previous execution is the considered value to be claimed by CPU, therefore the container is considered balanced

•Allocated CPU	Process Duration	Comments
2.0	~ 6 – 7 seconds	The KPI used is Task Duration and the starting reference point is **~ 6 – 7 seconds**. The *neededCPUInPeriods* results to less than 2**.**
1.0	~ 8 – 9 seconds	Container is restarted. Test execution series is triggered. Impact is noticed in Task Duration. CPU must be increased to **1.5.**.
1.5	~ 6 – 7 seconds	Container is restarted. Test execution series is triggered. Same KPI for Task Duration is reached. CPU must be decreased.
1.25	~ 6 – 7 seconds	Doesn't change or improve the KPI. Concluding that 1.25 is identified as optimal CPU and resources are balanced within process time boundaries while executing the identified use cases.

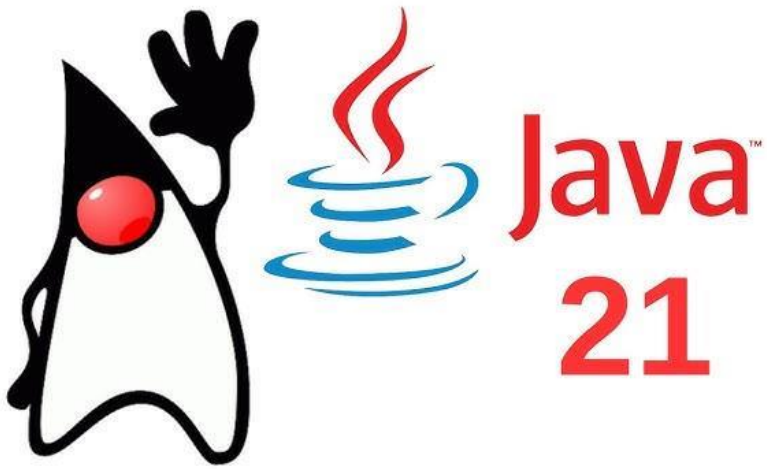
ASML

5. Monitor and adjust (Continuous results analysis in monitoring dashboards)

What helped us

10 steps

1. Optimize application framework, and application



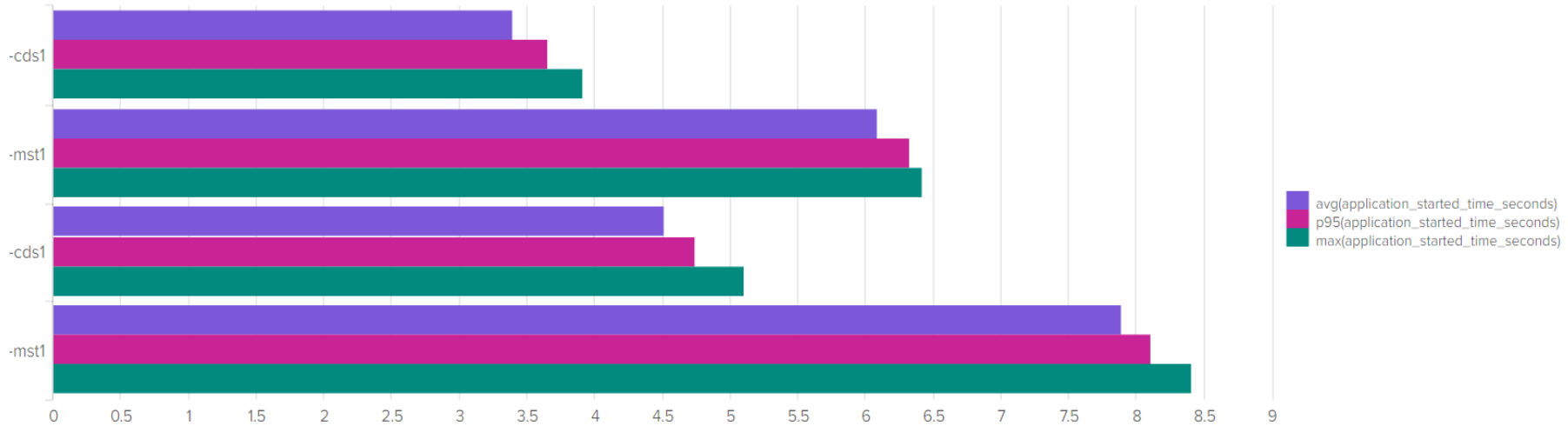
Virtual threads to rescue (JEP 444) – For I/O heavy services.

2. Use AOT processing (and/or native) and App CDS

Reduces start-up time and footprint – Project Leyden

1. AOT processing
2. App CDS

Further Java >24,
AOT Class loading and linking



~ 40% gain on application Start-up time.

~ 33% gain on components CPU resource utilization during Start-up.

Can be integrated with minimum application code changes.

ASML

3. Spring boot apps with Undertow as servlet container

```
server.undertow.threads.io=2
server.undertow.threads.worker=8
server.shutdown=graceful
spring.lifecycle.timeout-per-shutdown-phase=${LIFECYCLE_TIMEOUT_PER_SHUTDOWN_PHASE:30s}
server.undertow.await-graceful-shutdown.timeout=${UNDERTOW_GRACEFUL_SHUTDOWN_AWAIT_DELAY:20000}
```

CPU usage:	Undertow < Jetty < Tomcat
Memory:	Jetty < Undertow < Tomcat
Performance:	Tomcat < Jetty < Undertow

4. Fine tune JVM parameters and set right GC

-XX:ActiveProcessorCount

Specifies the number of CPUs reported by the operating system

Runtime.availableProcessors()

-XX:UseSerialGC

-XX:UseParallelGC

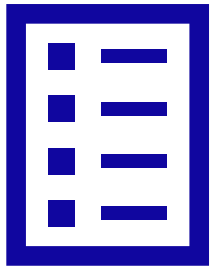
-XX:UseG1GC

-XX:UseZGC

-XX:UseShenandoahGC

Avoid relying on JVM defaults, especially in containerized environment.

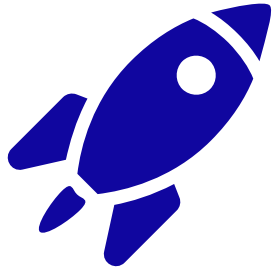
5. Async tasks and define thread pools



@Async, @CompletableFuture, @ScheduledTask

Ideal for long running or non-blocking tasks

Prevents main thread blockage => improving application throughput



Thread pools

For CPU usage, the pool size is best set to the number of CPU cores available.
For I/O-bound tasks, can be 2x time than the number of CPU cores available.

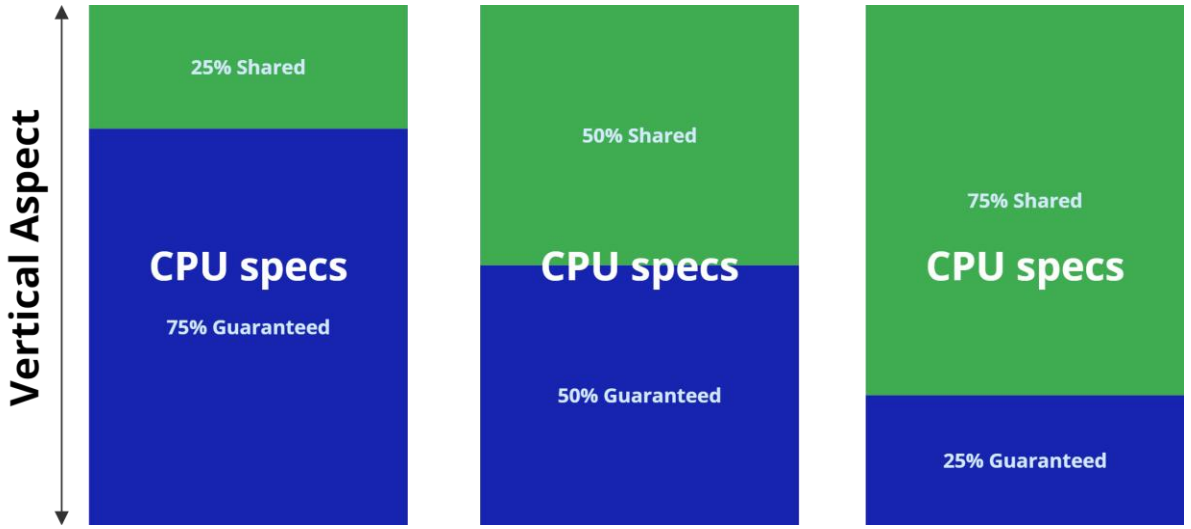
6. Set request for normal usage and high limits (or no limits ☺) (1/3)

To set or not set limits?

Don’t Set Limits Too Low

Idle CPU cycles can be a significant source of waste in a Kubernetes environment. To minimize them, we can employ strategies like:

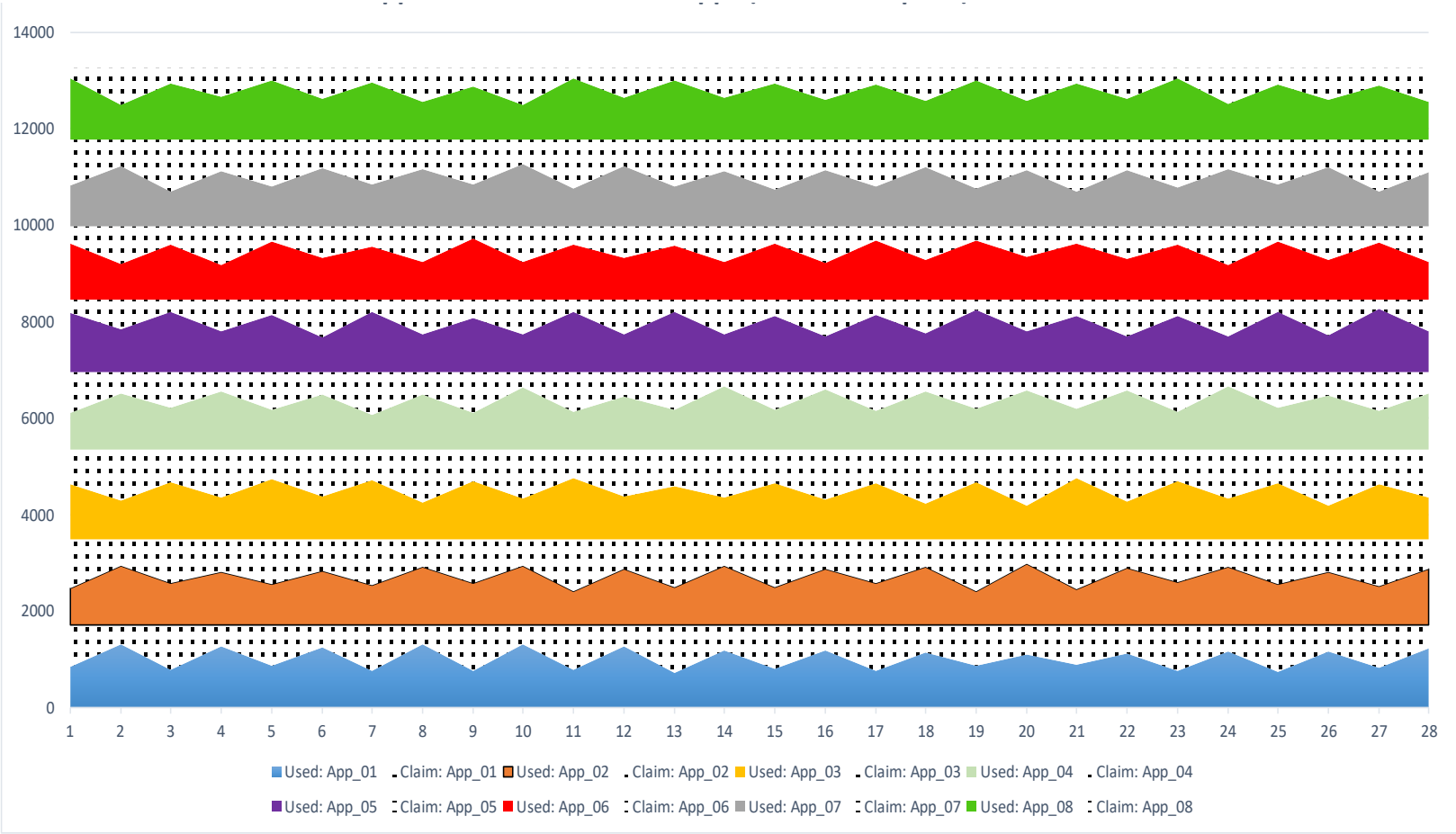
1. CPU bursting
2. Dynamic resource allocation
3. Idle resource reclamation



6. 100% CPU usage does not mean bad usage (2/3)

Check if there is starvation

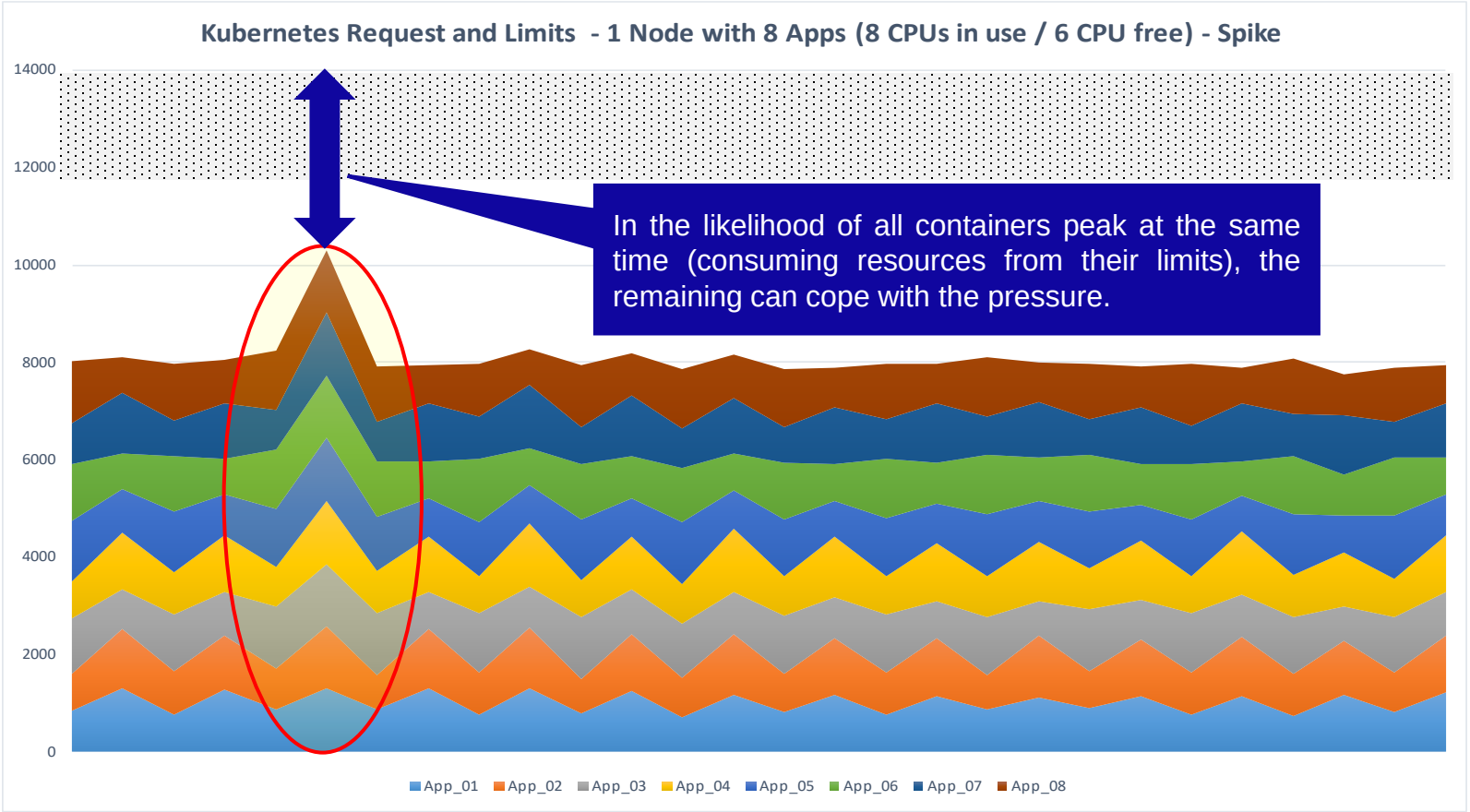
Control the number of threads per instance



6. Aim for 80% resource utilization (3/3)

Resource utilization =

used resource / claimed resource X 100



7. Kubernetes cluster autoscaler

K8s cluster autoscaling: Scaling the number of nodes in a cluster based on changing workloads and conditions.

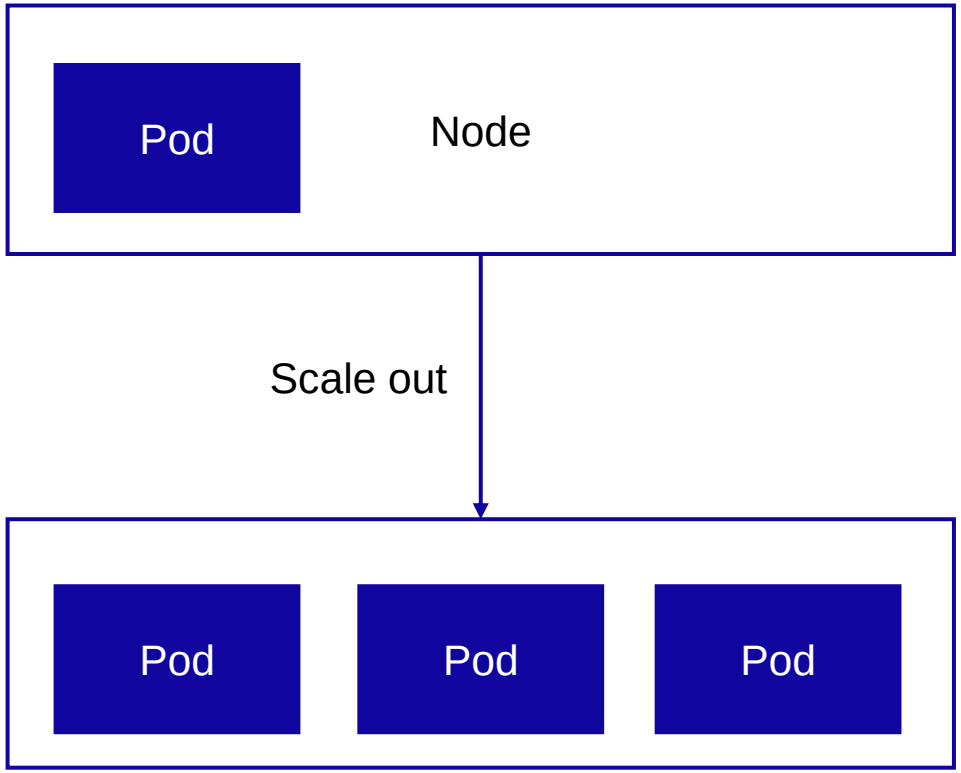
Not an option for us.

On prem cluster.

ASML

8. Horizontal pod autoscaler (1/2)

Horizontal pod autoscaling (HPA): Scaling the number of replicas based on CPU utilization or other metrics.



8. Horizontal pod autoscaler – KEDA (2/2)

KEDA defines autoscaling as a process of two phases:

1. The activation phase (zero-to-one), done by KEDA itself
2. Scaling phase (one-to-many), done by HPA instead

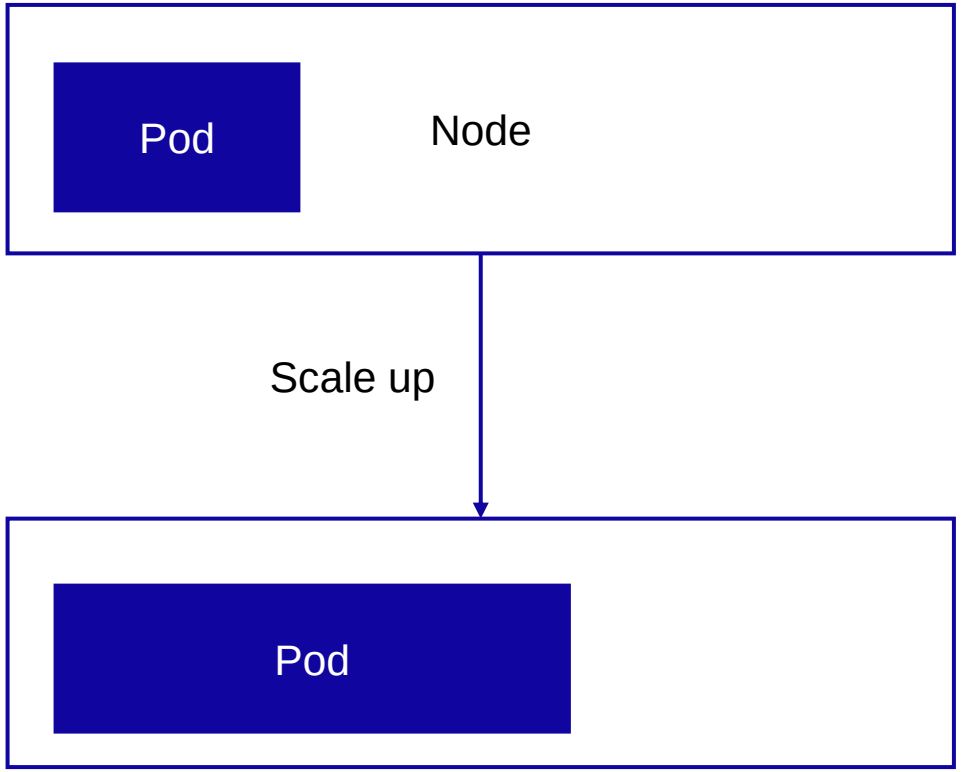
$$desiredReplicas = ceil\left[currentReplicas \times \frac{currentMetricValue}{desiredMetricValue}\right]$$

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: store-scaleobject
  namespace: store
spec:
  scaleTargetRef:
    name: store
  pollingInterval: 30
  cooldownPeriod: 120
  minReplicaCount: 2
  maxReplicaCount: 4
  triggers:
  - authenticationRef:
      name: splunk-auth
    metadata:
      activationValue: "1"
      host: http://splunk_url:splunk_port/search_api
      savedSearchName: store-cpu-usage
      targetValue: "1500"
      valueField: usage
    name: store-cpu-usage
    type: splunk
  - authenticationRef:
      name: splunk-auth
    metadata:
      activationValue: "1"
      host: http://splunk_url:splunk_port/search_api
      savedSearchName: store-connection-count
      targetValue: "12"
      valueField: connection-count
    name: store-connection-count
    type: splunk
```

ASML

9. Vertical pod autoscaler

Vertical pod autoscaling (VPA): Scaling the resources allocated to a pod based on changing workloads and conditions.



10. In-place vertical pod scaling (default enabled, beta) – K8s 1.33

Resizing pods without restart

Patch pod with `resource.requests` and `resource.limits` introduced as part of KEP-1287

Kubelet Check: (Node's allocatable capacity - Sum of all existing container allocations) >= (New request)
If yes, proceed, if no `PodResizePending`

CRI Handshake: Adjust cgroups accordingly without restart (via containerd or CRI-O)

Status update:
`PodResizePending` - Node is busy. Try again later

`PodResizeInProgress` - Kubelet resize accepted (allocated resources), but changes are still applied.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: component
spec:
  containers:
  - name: pause
    image: image-registry/image-name:image-version
    resizePolicy:
    - resourceName: cpu
      restartPolicy: NotRequired # Default, but explicit here
    - resourceName: memory
      restartPolicy: RestartContainer
    resources:
      limits:
        memory: "800Mi"
        cpu: "0.5m"
      requests:
        memory: "800Mi"
        cpu: "2m"
```

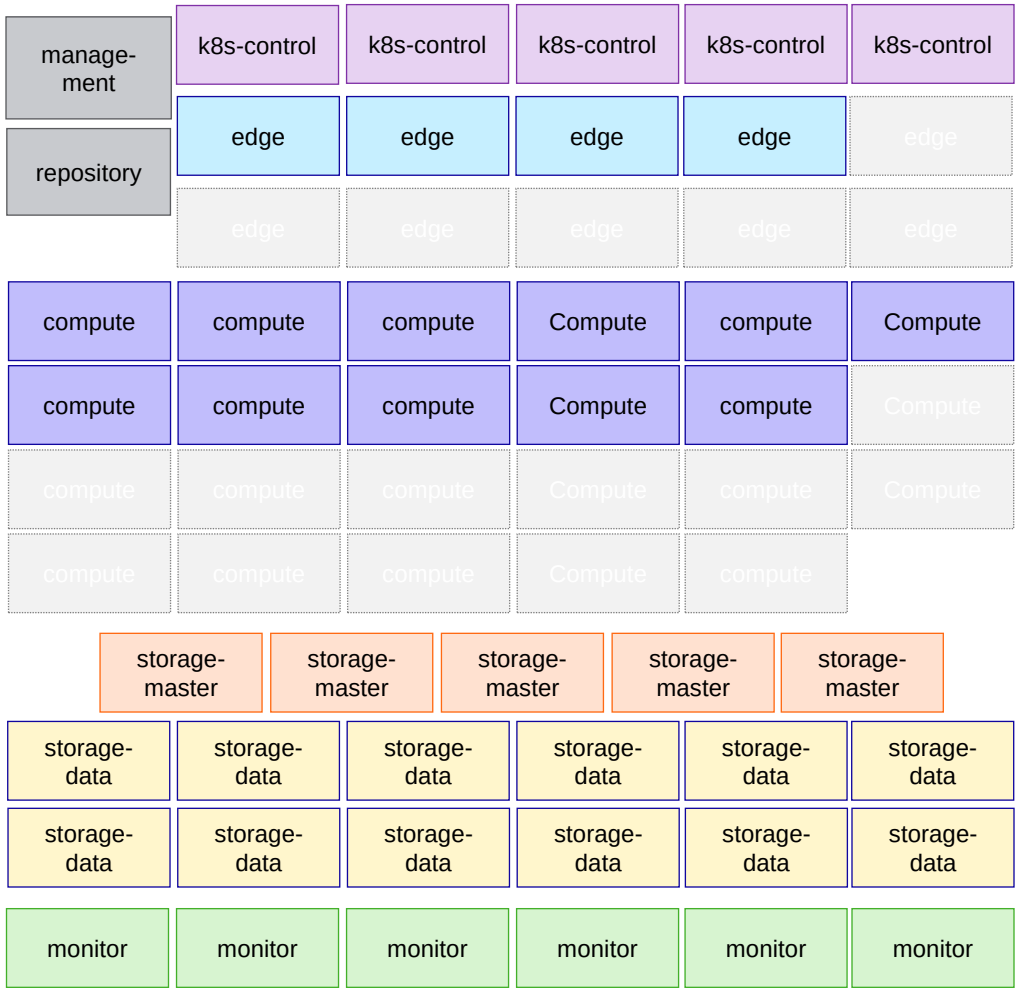
Success story: Optimizing CPU sharing

Real gains

50%+ reduction in infrastructure footprint

10 to 20% faster SLO adherence across key workloads

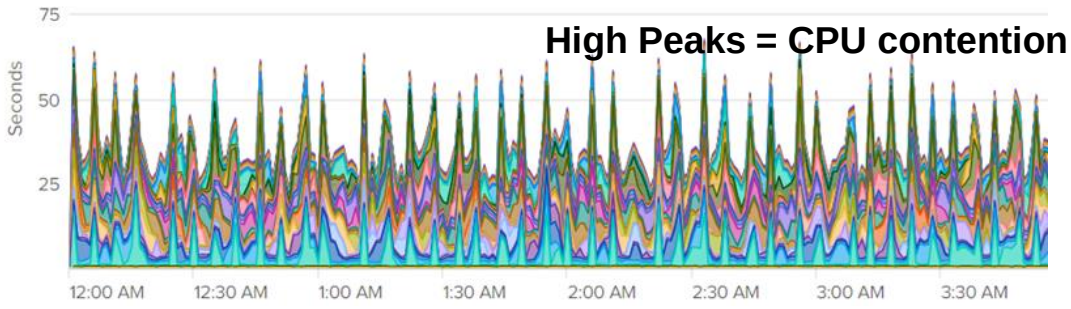
Fewer idle cores and better burst handling



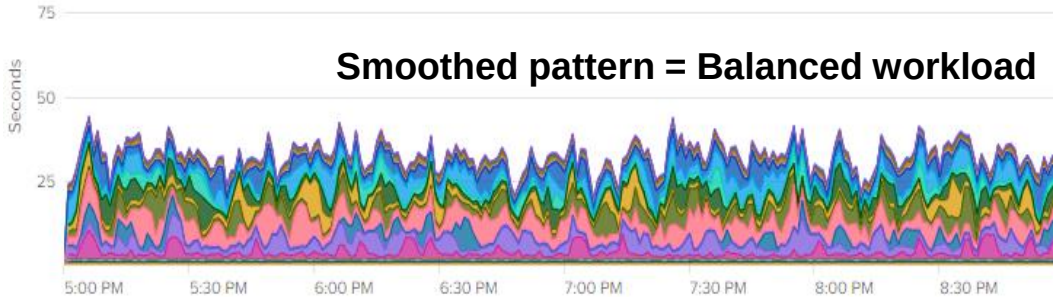
Results: CPU sharing drives latency gains

- **1500+ jobs/hour** completed consistently – high throughput in shared environments
- **Reduced average process latency** observed – clear performance gain after CPU sharing
- **Lower latency variance across processes**

Before Optimization



After Sharing idle CPU

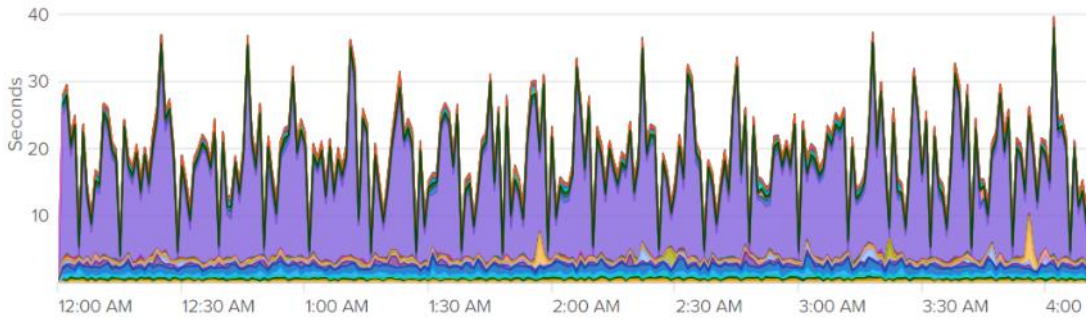


CPU time (Seconds) across various workloads

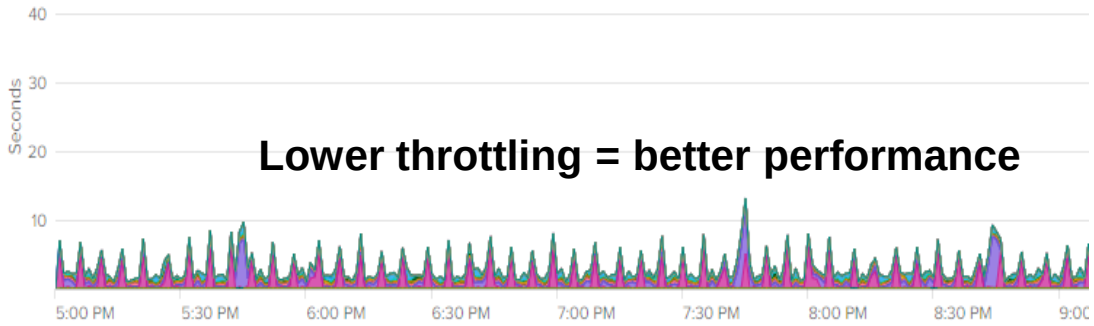
CPU sharing reduces throttling – with 50% less hardware

- **50% hardware footprint reduction**
- **Significantly lower CPU throttling** observed after sharing idle resources

Before Optimization

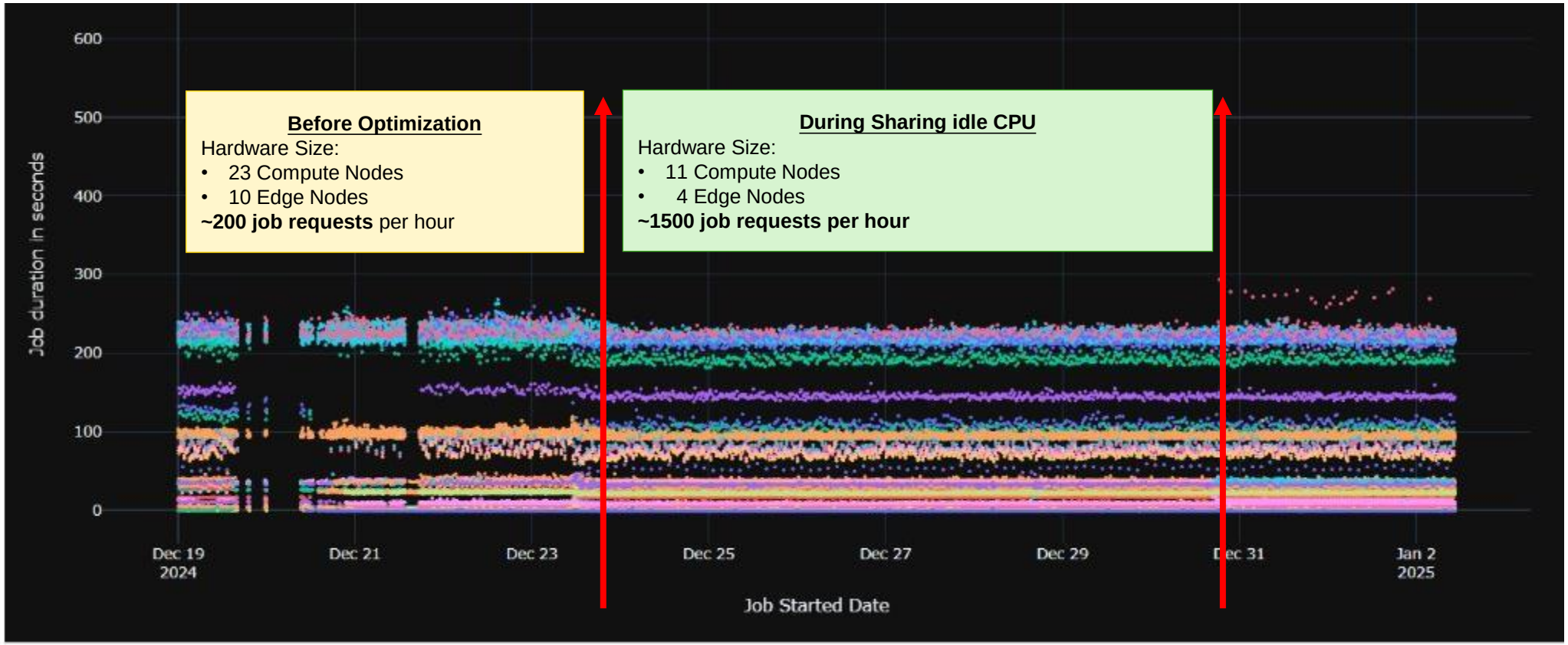


After sharing idle resources



Process throttling time (Seconds) across various workloads

Response times stabilized after sharing idle resources



ASML

Key takeaways

Understand your application’s behavior and load profile

Don’t rely on default JVM settings – fine-tune parameters
Limit thread count to avoid contention

Reduce requested CPU to maximize packing, set appropriate limits to avoid throttling
We reduced it by as much as 75% for critical workloads and 99% for non-critical workloads

Continuously monitor, adapt and tune

Aim for efficiency, not a fixed target (Utilization can be 50-100% based on app requirements)

Scale using app-specific KPIs (not just CPU/memory)

Happy optimizing – may your pods never throttle

Q & A