KubeCon | CloudNativeCon

Europe 2022

WELCOME TO VALENCIA

Getting the optimal service efficiency that autoscalers won't give you

Mauro Pessina

moviri

Agenda & Speaker

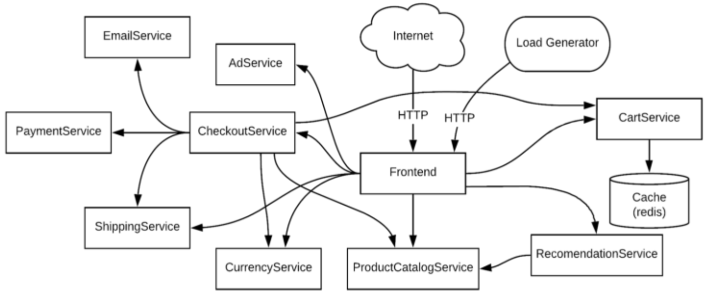
- The problem
- Kubernetes challenges
- AI-powered optimization: a real-world case
- Conclusions and Q&A

Mauro Pessina
Head of Performance Eng
Moviri

The problem

New challenges for modern apps

cloud-native applications



dozens or hundreds of microservices

tunables parameters and options

- Application Runtime (each microservice)
- Resource requests (each microservice)
- Resource limits (each microservice)
- Replicas & Auto-Scaling policies
- Cloud instance options
- ...

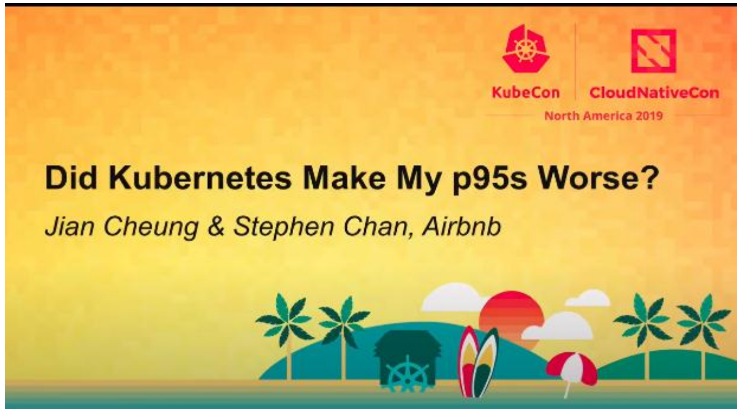
hundreds or thousands configurations

Kubernetes challenges are real

Kubernetes failure stories

(https://k8s.af)

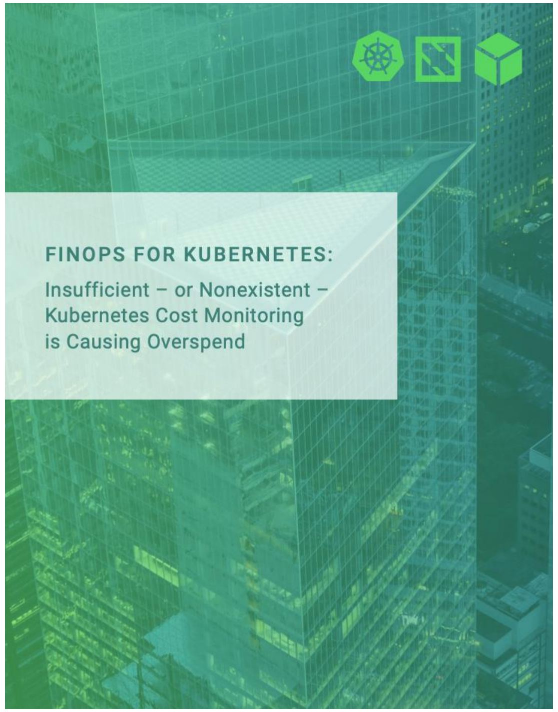
https://www.youtube.com/watch?v=4CT0cI62YHk



https://youtu.be/QXApVwRBeys

growing Kubernetes costs

(https://www.cncf.io/wp-content/uploads/2021/06/FINOPS_Kubernetes_Report.pdf)

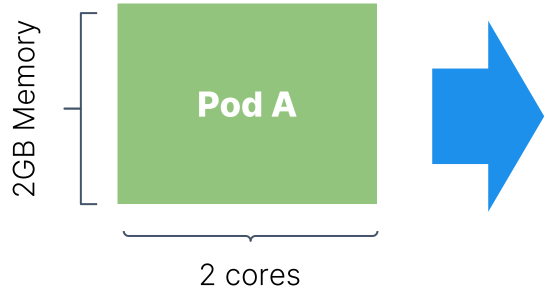


Kubernetes challenges

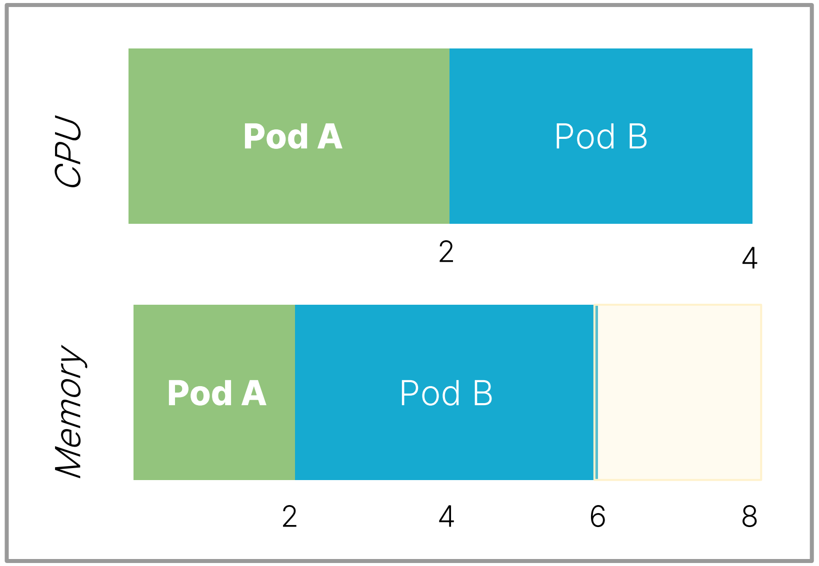
Fact #1: Resource requests make cluster capacity

Resource Requests from Pod Manifest

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: Pod A
spec:
  containers:
  - name: app
    image: nginx:1.1
    resources:
      requests:
        memory: “2Gi”
        cpu: “2”
```



Node (4 CPU, 8 GB Memory)

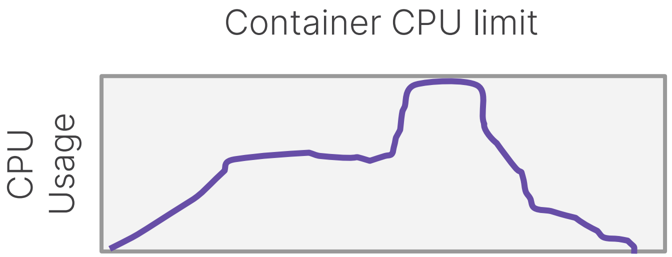


- Requests are resources the container is guaranteed to get
- **Cluster capacity** is based on pod resource requests - **there is no overcommitment!**
- **Resource requests is not equal to utilization**: a cluster can be full even if utilization is 1%

Fact #2: Resource limits may strongly impact application performance and stability

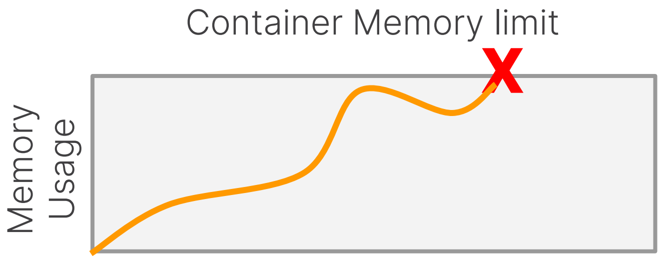
- **A container can consume more resources than it has requested**
- **Resource limits** allow to specify the maximum resources a container can use (e.g. CPU = 2)
- **When a container hits its resource limits bad things can happen**

When hitting CPU Limits



K8s throttles container CPU
-> Application performance slowdown

When hitting Memory Limits

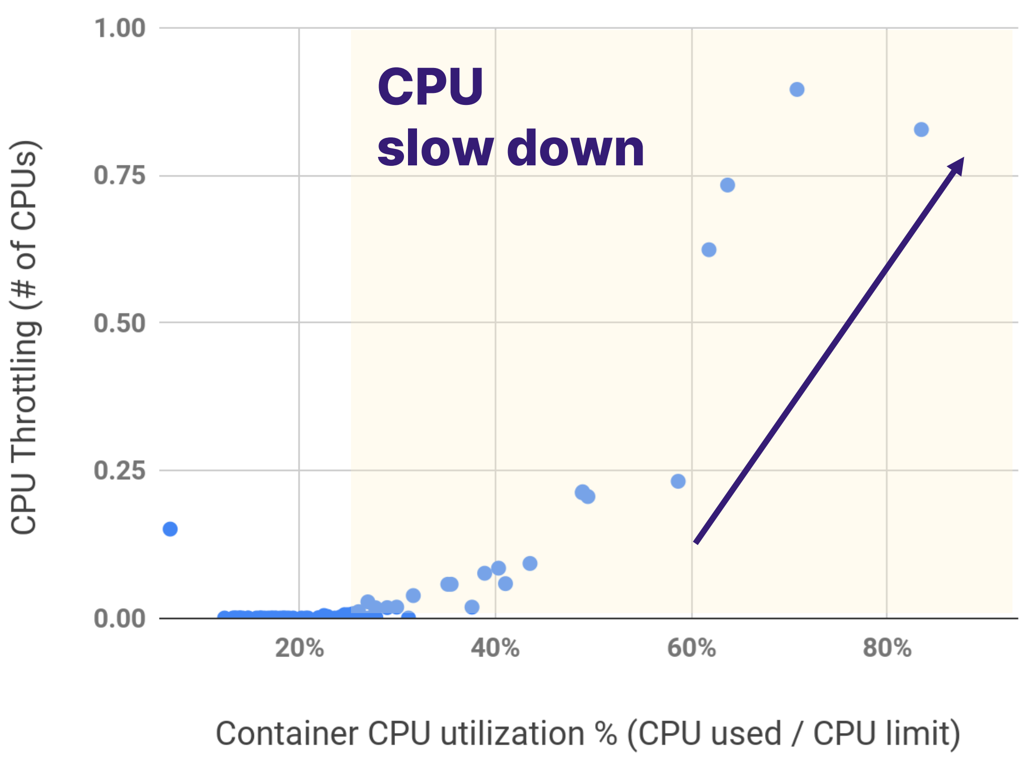


K8s kills the container
-> Application stability issues

Fact #3: CPU limits may disrupt service performance, even when CPU used << limit

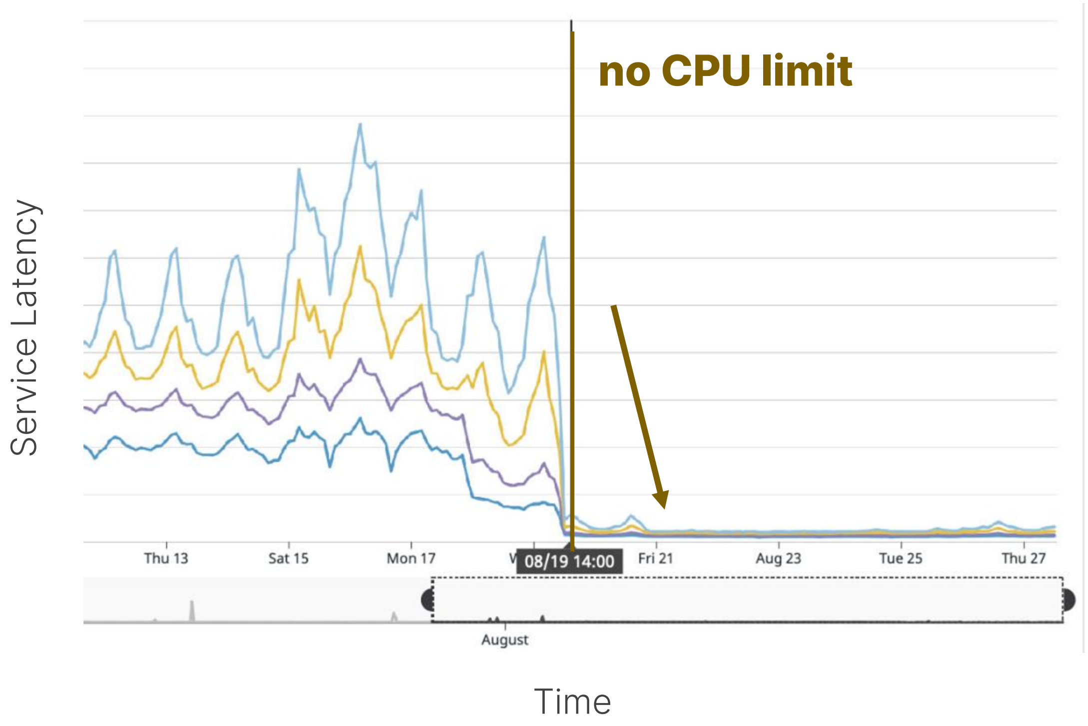
KubeCon CloudNativeCon Europe 2022

Limits restrict CPU speed at very low CPU usage (~30%)



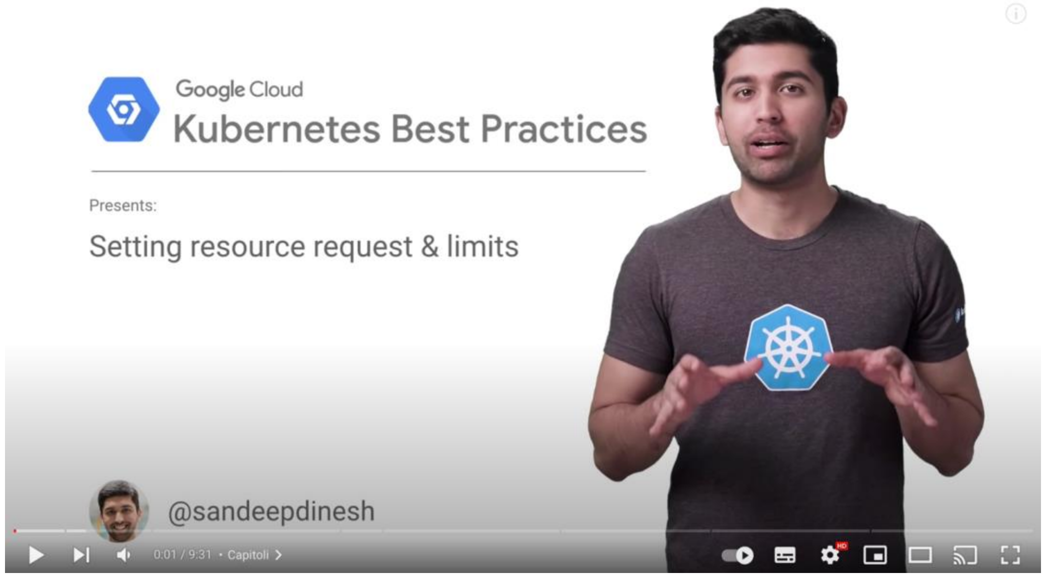
Source: Moviri Research

22x faster services with no CPU limits (Buffer Inc.)



https://erickhun.com/posts/kubernetes-faster-services-no-cpu-limits

Fact #4: Setting resource requests and limits is required to ensure K8s stability

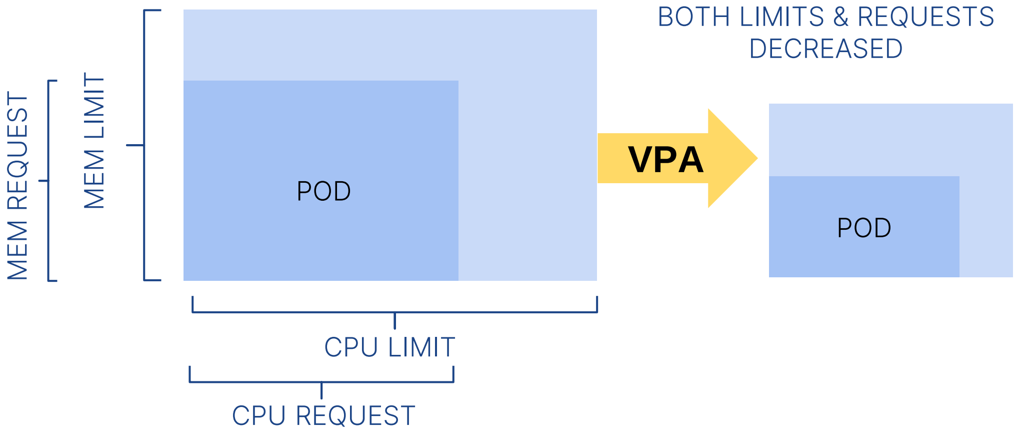


https://www.youtube.com/watch?v=xjpHggHKm78&t=18s

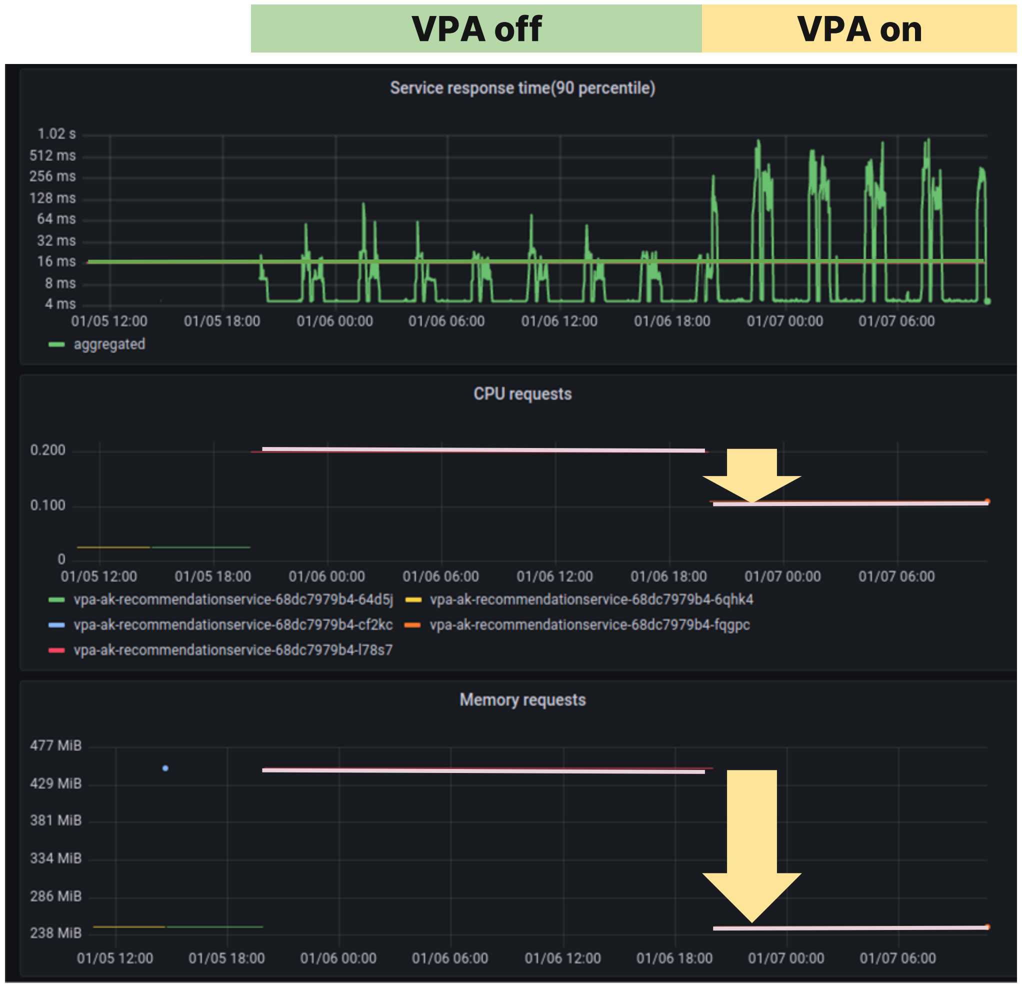
"While your Kubernetes cluster might work fine ***without setting resource requests and limits****, you will start running into* ***stability issues*** *as your teams and projects grow"*

- Developer Advocate, Google

Fact #5: K8s Vertical Pod Autoscaler do not address service reliability



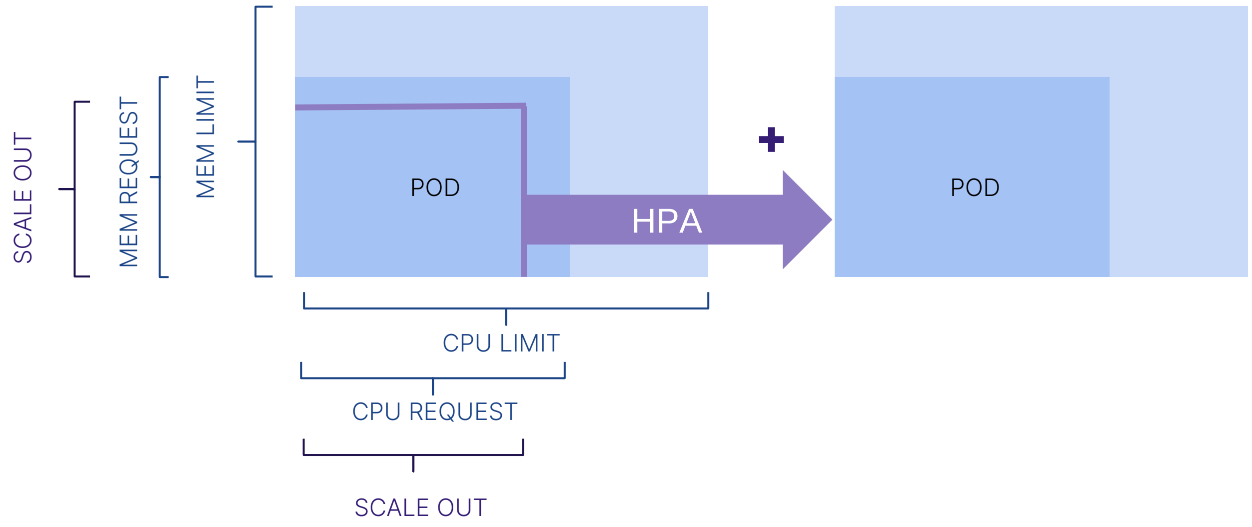
Vertical Pod Autoscaling IS NOT service aware -> SLOs can be breached



SLO: response time (p90) < 16ms

Fact #6: K8s Horizontal Pod Autoscaler multiply inefficiency

With HPA inefficiencies of a single pod get multiplied when new replicas are added



=

2x out-of-memory kills
2x risk of other pods starvage
2x wasted resources

AI-powered optimization:
a real-world case

A real-world case:
European SaaS provider of financial services

1,7M users

400M invoices / year

B2B Stateless Authorization Service

Digital Service App

Digital Service App

Digital Service App

frequent updates for business & regulatory compliance

Azure Kubernetes Service (AKS)

AWS Elastic Kubernetes Service (EKS)

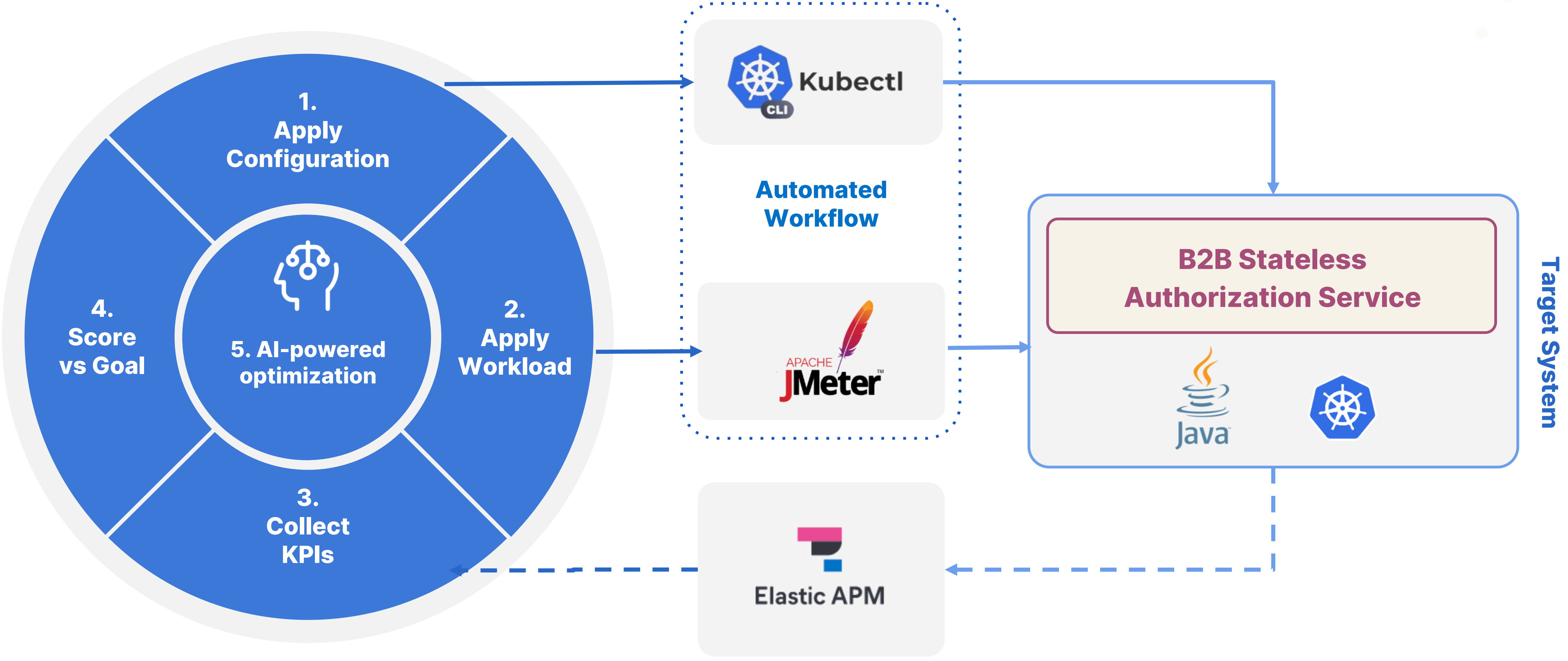
The customer tuning practice

- Over-provisioning by dev team to avoid risks
- Manual tuning approach
- About 2 months to tune one single microservice
- Hard to find tradeoffs among performance, resilience and cost

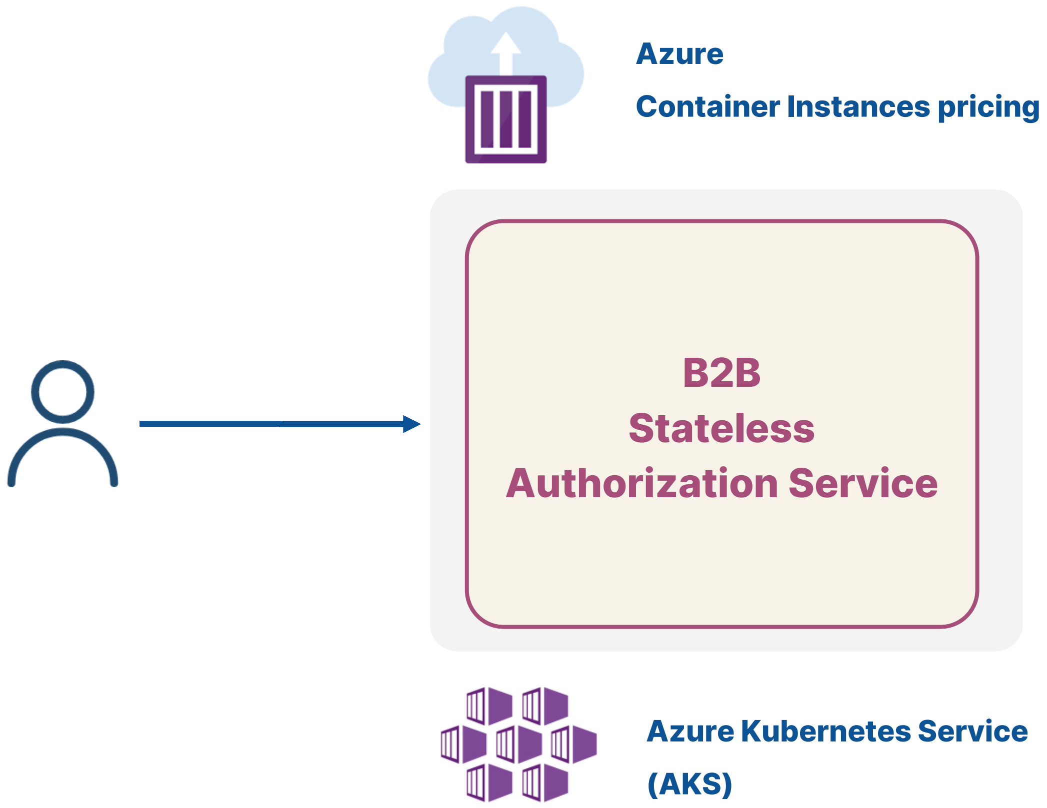
LEADING TO

- IT Overspending
- Low operational efficiency
- Low Business Agility

Movìri optimization methodology applied



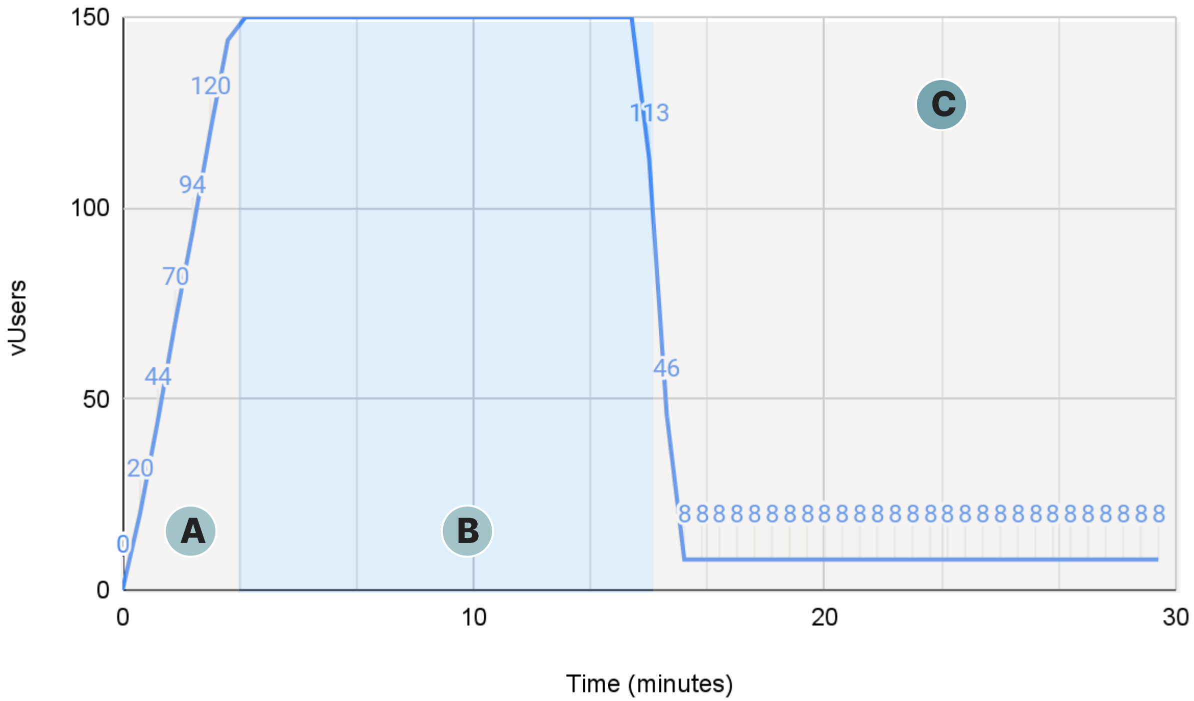
Stating optimization goals & constraints (SLOs)



Optimization goal & constraints

MINIMIZE
application_cost
WITH
transaction_throughput > *baseline* - 10%
AND
transaction_errore_rate < *baseline* + 10%
AND
transaction_response_time < *baseline* +10%

Defining Load Testing scenarios



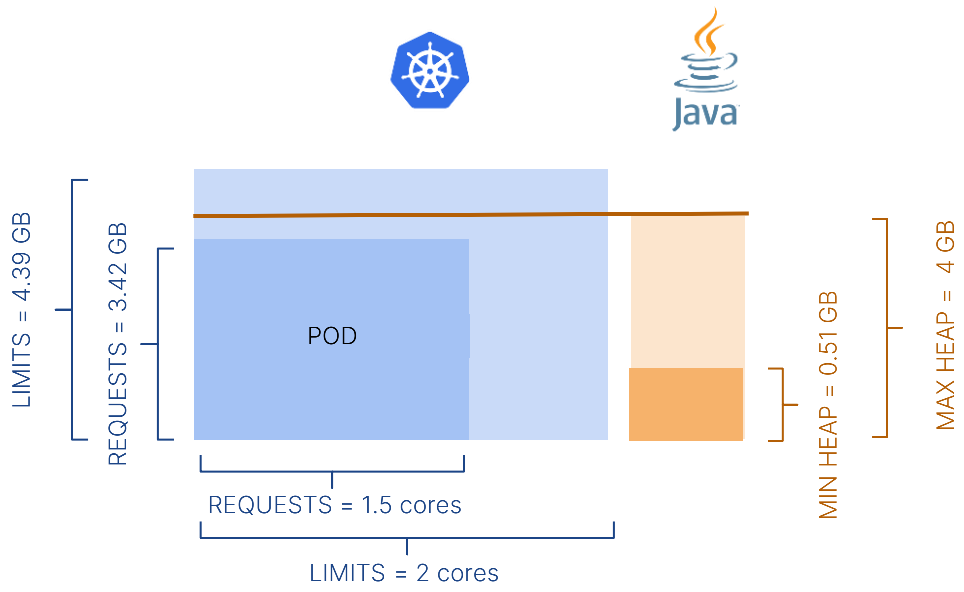
The **load scenario** was designed to replicate the daily behavior in a 30m time window:

- A **ramp-up** [3m]
- B **steady state with 150 Users and ~1200 requests/s** [~12m] corresponding to productive hours of the day
- C **steady state with 8 Users and ~65 requests/s** [~14m] corresponding to out of working hours

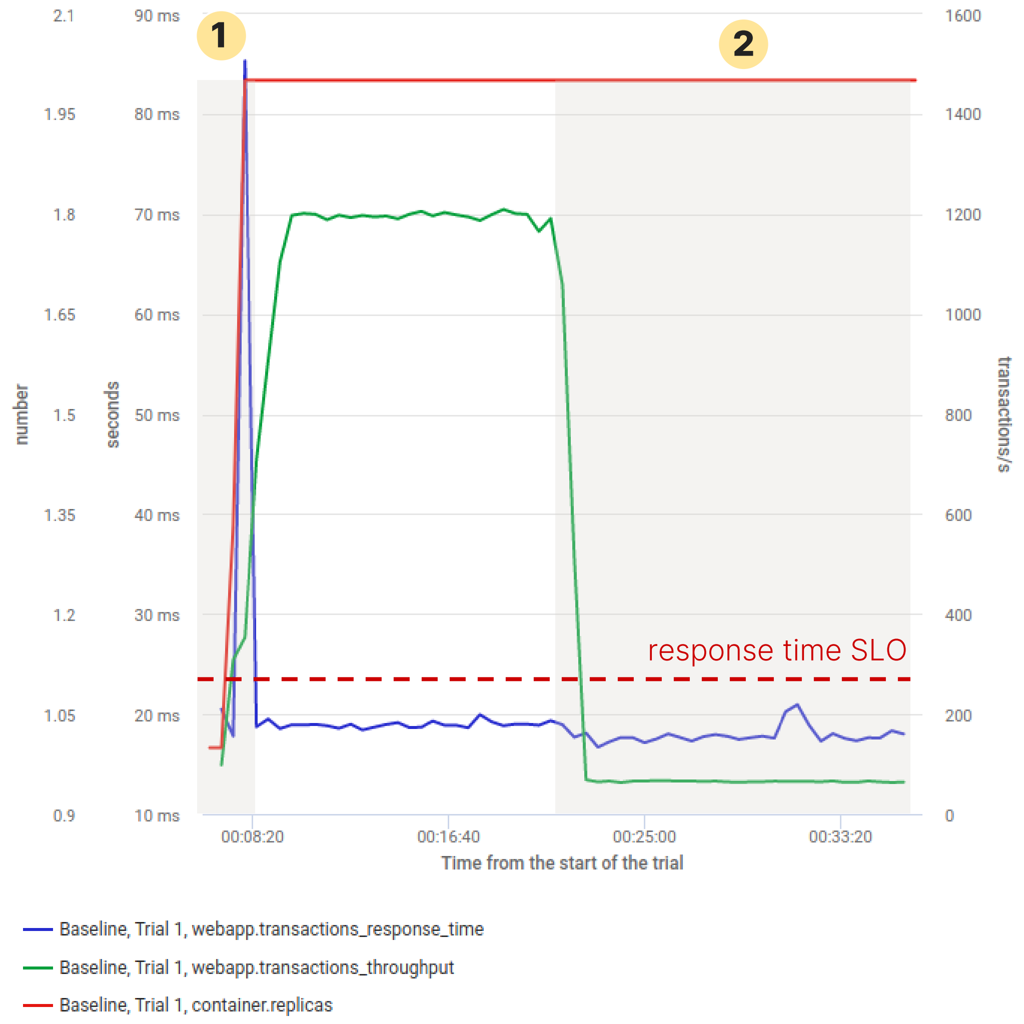
The **testing script** was also designed to respect the **user distribution** (as provided by a dataset is composed by 20K unique credentials) and the **API calls distribution** (as calculated from production log analysis) with each API call delayed from the previous one by a random pause (think time) between 250ms and 750ms.

The starting configuration aka Baseline

Component	Parameter	BASELINE
container	limits_cpu	2 cores
container	limits_memory	4.39 GB
container	requests_cpu	1.5 cores
container	requests_memory	3.42 GB
jvm	jvm_activeProcessorCount	1 CPUs
jvm	jvm_gcType	G1
jvm	jvm_maxHeapSize	4 GB
jvm	jvm_minHeapSize	512 MB
jvm	jvm_newSize	300 MB

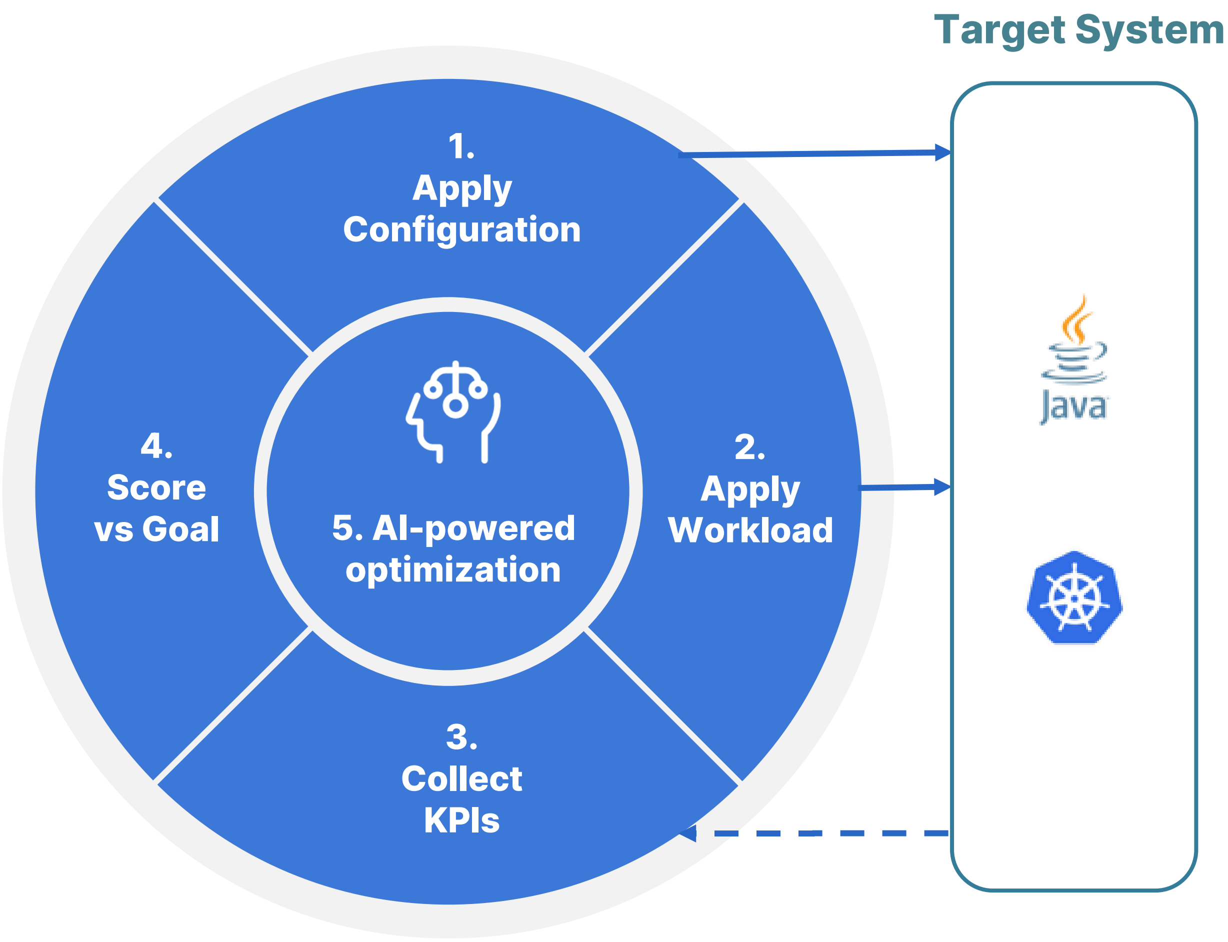


How the Baseline was behaving

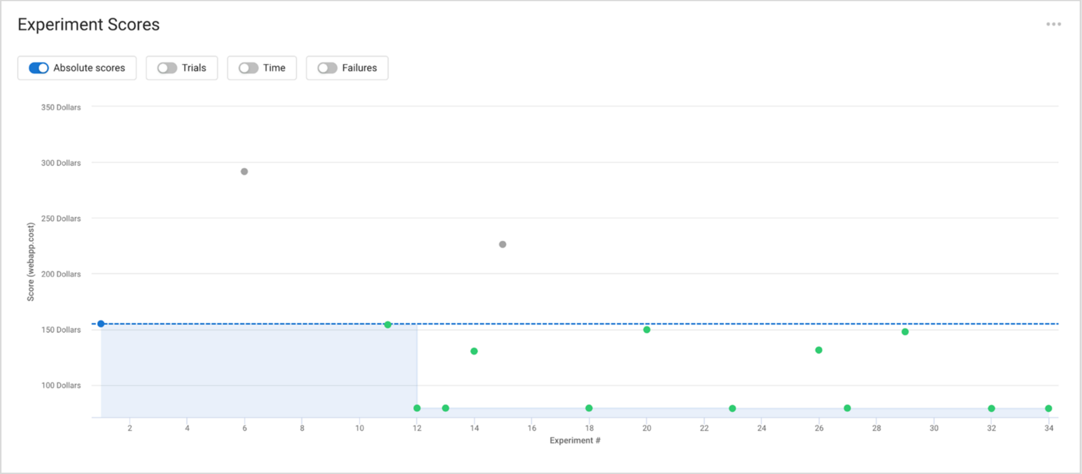


1. Response time peak breaching service reliability SLO due to high CPU usage and throttling during the scale out (JVM startup)
2. When load drops the number of replicas didn’t scale down, despite low resource use (CPU) – this clearly impacted cloud bill

Applying AI-powered optimization



Goal: application_cost



34 experiments (< 24 hours elapsed time)

AI-driven results: Best (Lowest Cost) conf

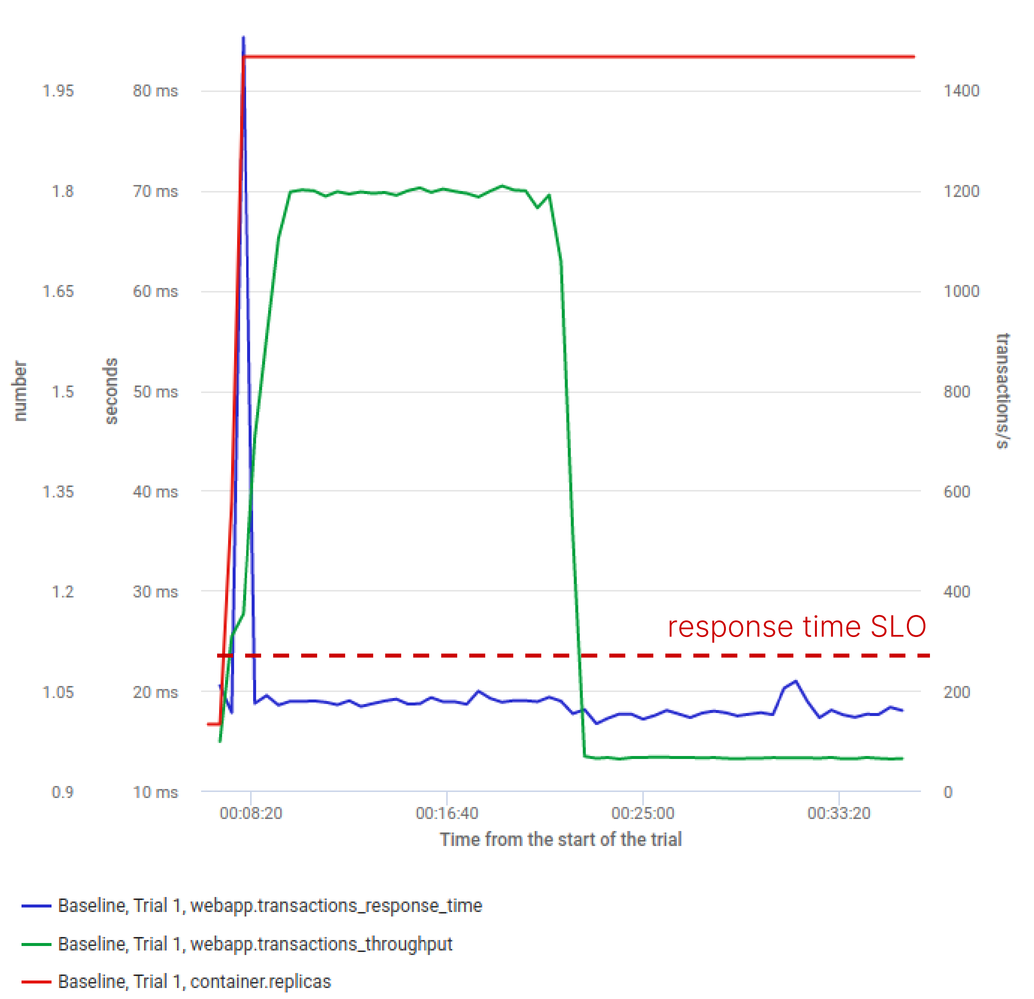
CONFIGURATION #34
(after 19h)

-49.1%

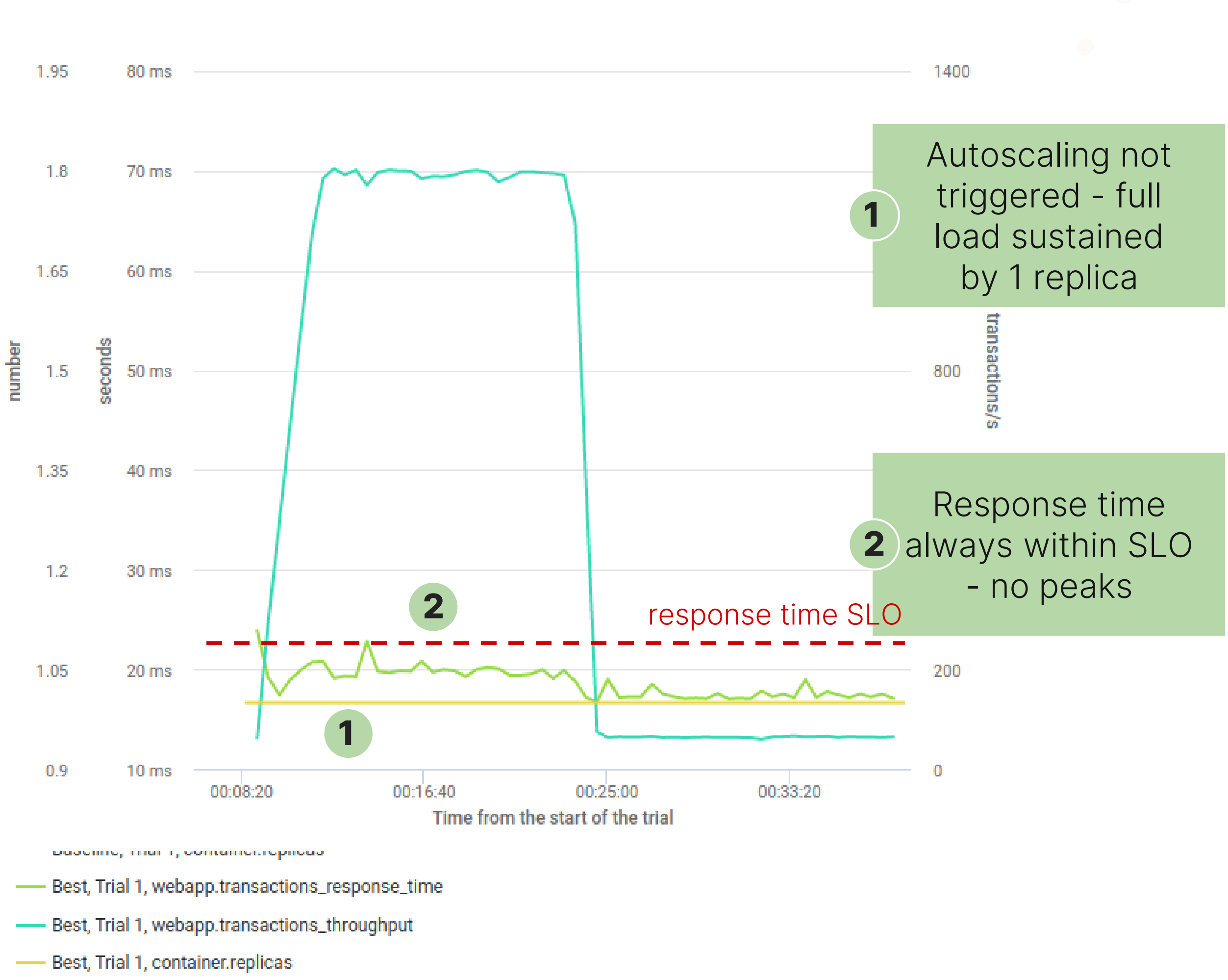
Parameter	Relevance	BEST	BASELINE
container requests_cpu		2.77 cores (+84.9%)	1.5 cores
container limits_cpu		3.67 cores (+83.3%)	2 cores
container limits_memory		5.16 GB (+17.5%)	4.39 GB
container requests_memory		5.08 GB (+48.8%)	3.42 GB
jvm jvm_activeProcessorCount		1 CPUs	-
jvm jvm_gcType		G1	-
jvm jvm_maxHeapSize		4.76 GB (+19%)	4 GB
jvm jvm_minHeapSize		4.37 GB (+774%)	512 MB
jvm jvm_newSize		1.71 GB	-

AI-driven results: Best vs Baseline behaviour

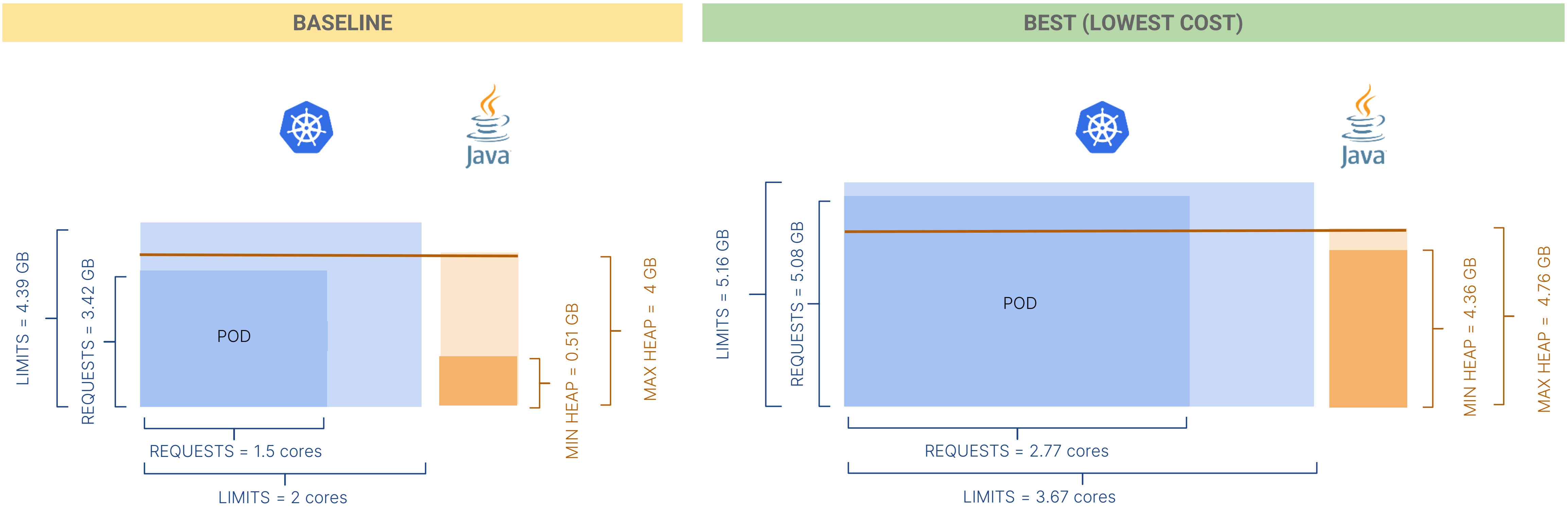
BASELINE



BEST (LOWEST COST)



AI-driven results: Best vs Baseline analysis



larger pod (higher fixed cost but less scaling) + container & runtime aligned conf

AI-driven results: High Resilience configuration

KubeCon CloudNativeCon Europe 2022

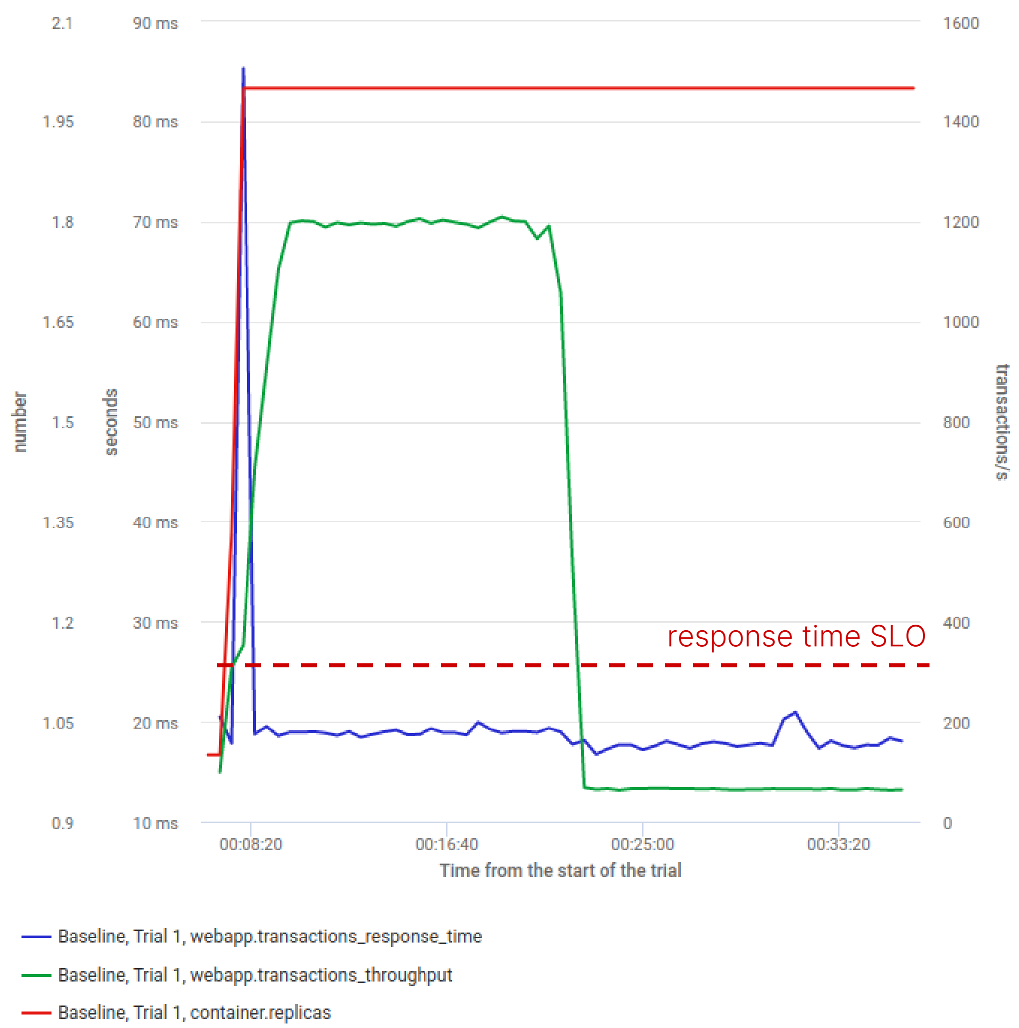
CONFIGURATION #14
(after 8h)

-15.9%

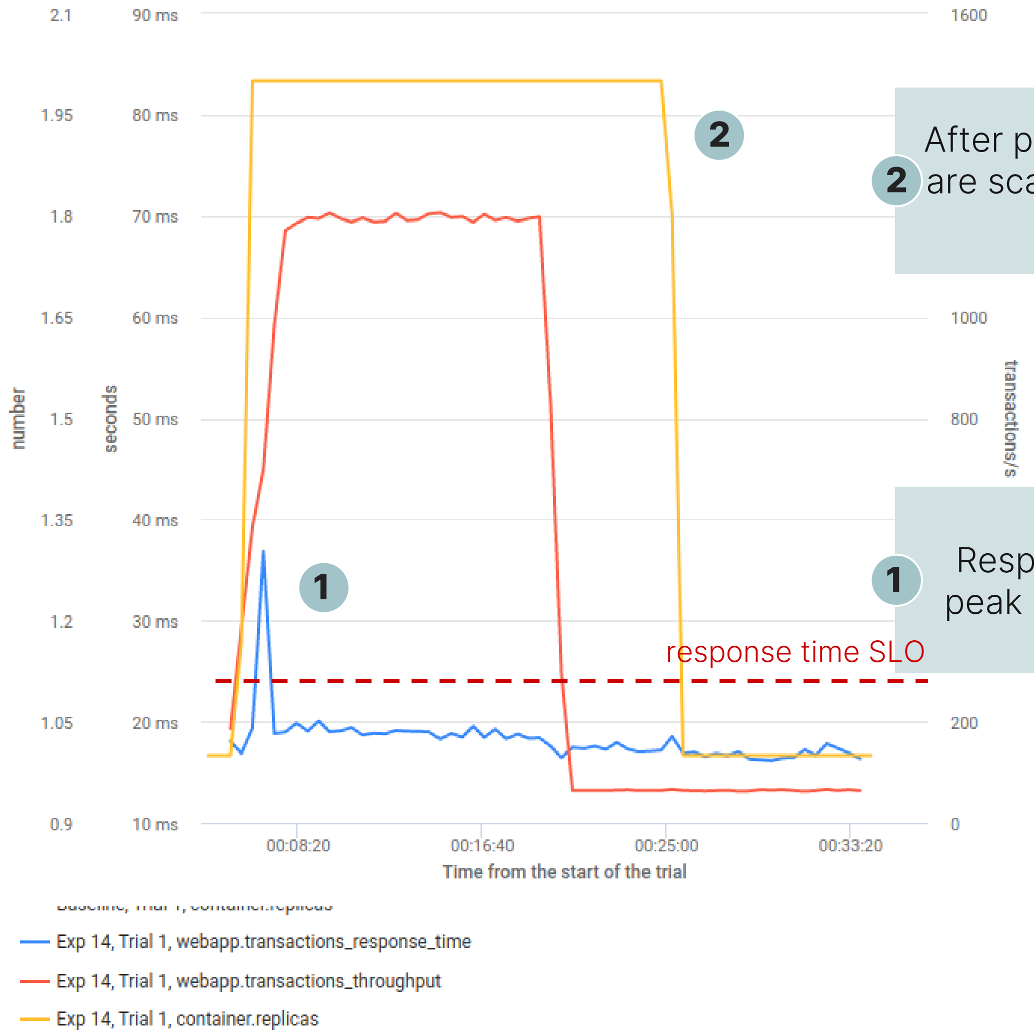
Component	Parameter	HIGH RESILIENCE	BEST	BASELINE
container	limits_cpu	3.7 cores	3.67 cores	2 cores
container	limits_memory	5.69 GB	5.16 GB	4.39 GB
container	requests_cpu	1.17 cores	2.77 cores	1.5 cores
container	requests_memory	5.6 GB	5.08 GB	3.42 GB
jvm	jvm_activeProcessorCount	6 CPUs	1 CPUs	-
jvm	jvm_gcType	Parallel	G1	-
jvm	jvm_maxHeapSize	3.45 GB	4.76 GB	4 GB
jvm	jvm_minHeapSize	1.94 GB	4.37 GB	512 MB
jvm	jvm_newSize	1,000 MB	1.71 GB	-

AI-driven results: High-Resilience conf

BASELINE



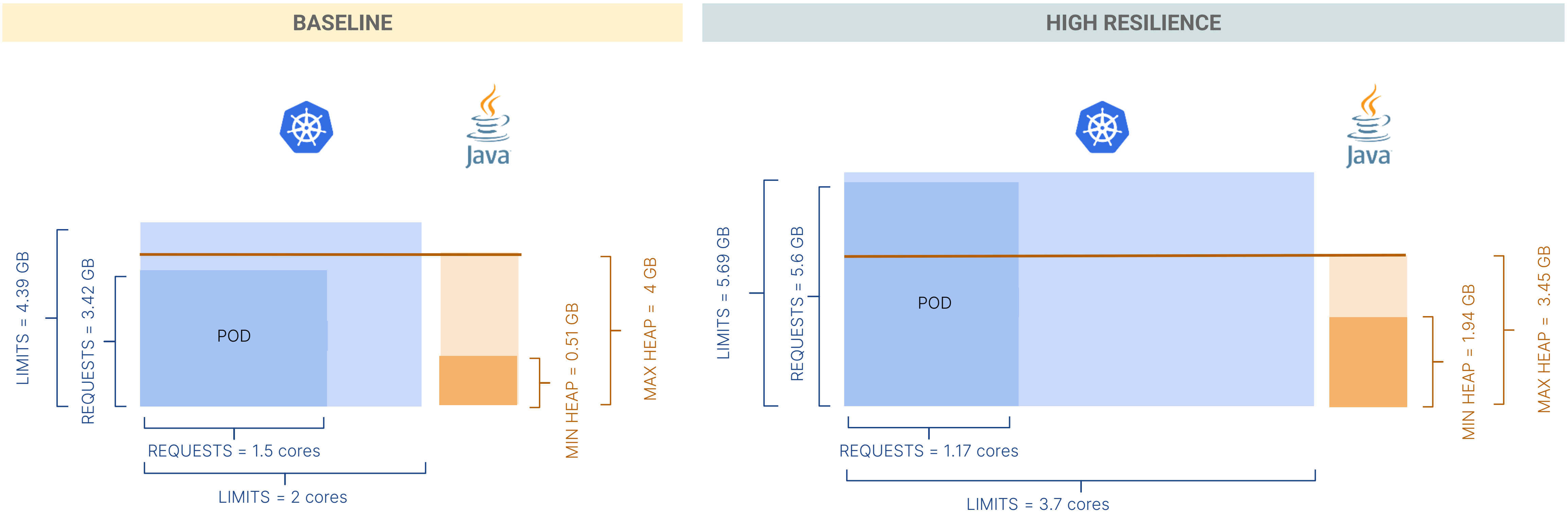
HIGH RESILIENCE



2 After peak replicas are scaled back to 1

1 Response time peak is 2x lower

AI-driven results: High Resilience vs Baseline



higher memory requests and lower CPU requests (but higher limits) than baseline

Customer results

- Right-sizing of service pods - no overprovisioning
- Automated tuning approach
- <1 day vs 2 months to tune a critical microservice
- Zero degradation on service quality wrt baseline

LEADING TO

- Cost Reduction
- Lower application latency
- Better User Experience
- Higher Op Efficiency

Conclusions and Q&A

Key takeaways

1. Tune, tune, tune - any (cost) inefficiency is not going to be addressed by K8s
2. AI-powered optimization enables experts to deal with today's complex apps

KubeCon | CloudNativeCon

Europe 2022

WELCOME TO VALENCIA