Tuning Apache Spark Resource Usage For Fun And Efficiency

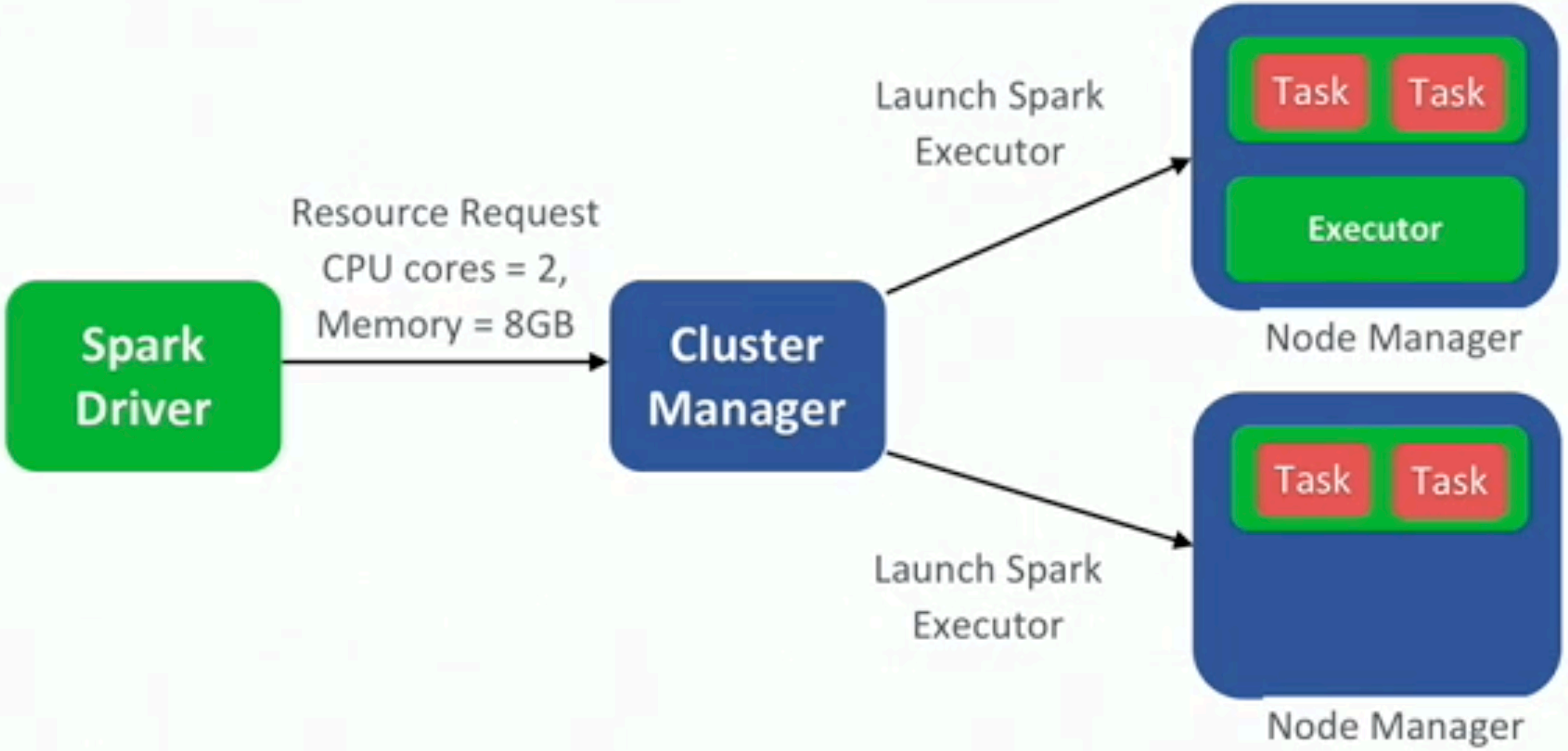
By Sital Kedia and Sergey Makagonov
Facebook

Agenda

- Spark Execution & Memory Model
- Resource Efficiency Metrics
- Resource Inefficient Applications
- History-based Resource Tuning
- Results

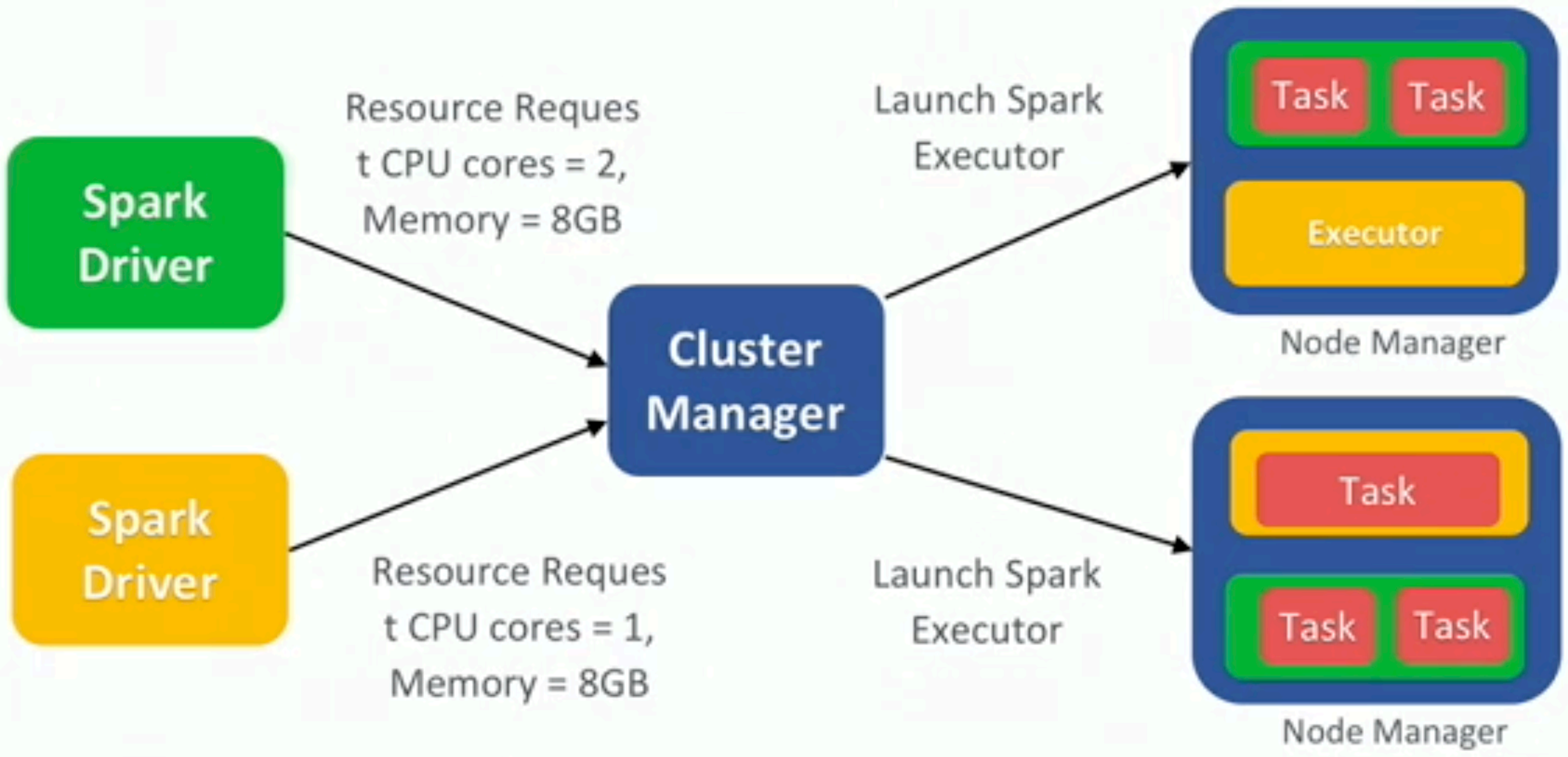
Spark Execution Model

Spark Execution Model



- Cores per executor = *`spark.executor.cores`*
- Memory per executor = *`spark.exector.memory + spark.*.memory.overhead`*
- *Cores per task* = *`spark.task.cpus (default is 1)`*
- Tasks per executor = *`spark.executor.core / spark.task.cpus`*

Spark Execution Model



- Separate driver per application
- Executors/Tasks are not shared across applications

Spark Execution Model

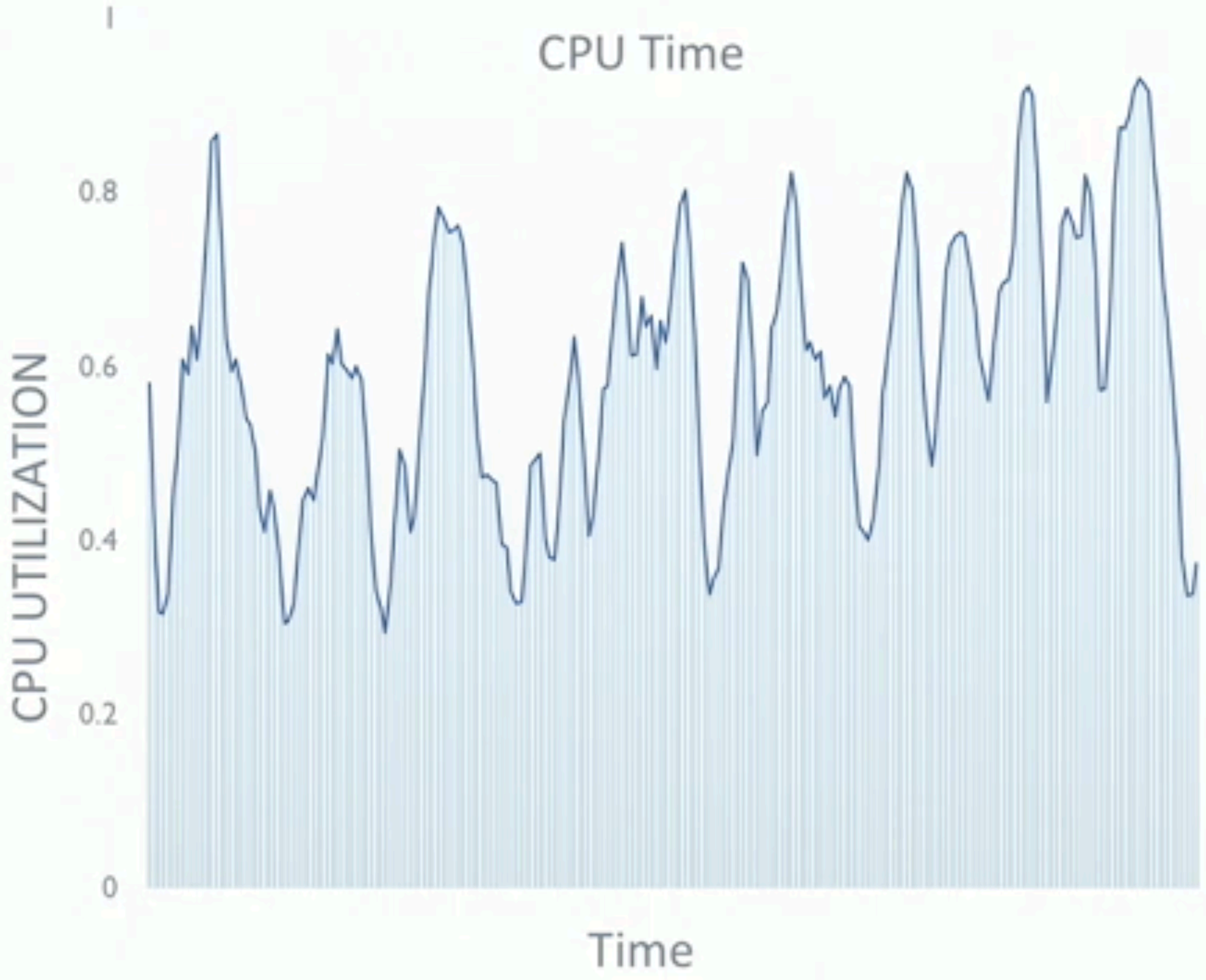
Cores = 4
spark.task.cpus = 1

- Each task is allocated one CPU cores at a minimum
- Tasks can be I/O bound which can lead to wastage of CPU

CPU Efficiency Metrics

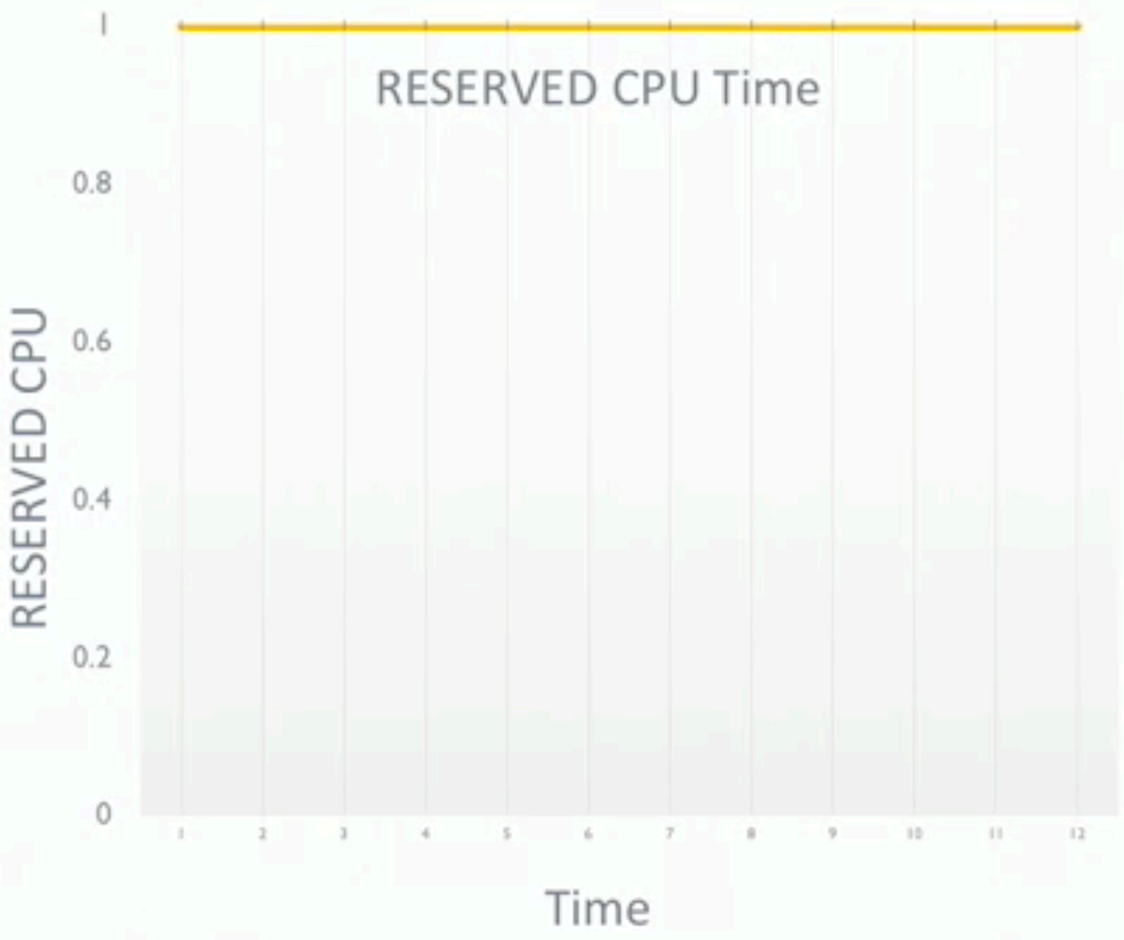
CPU Time

- CPU usage from the perspective of the OS
- Aggregated across all executors to calculate CPU Time for a Spark application
- Area under the curve for CPU usage over time



CPU Reservation Time

- Allocated CPU from the perspective of Resource Manager
- Aggregated across all executors to calculate CPU Reservation Time for a Spark application

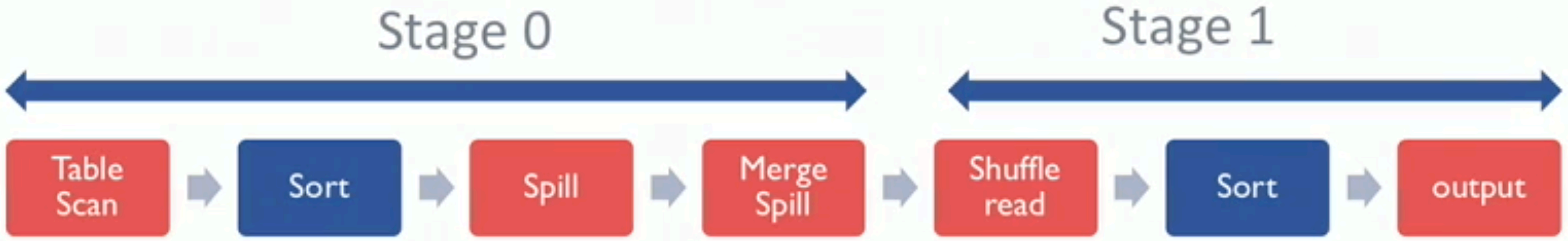


CPU Efficiency

- CPU reservation time can be significantly higher than CPU time for I/O bound applications

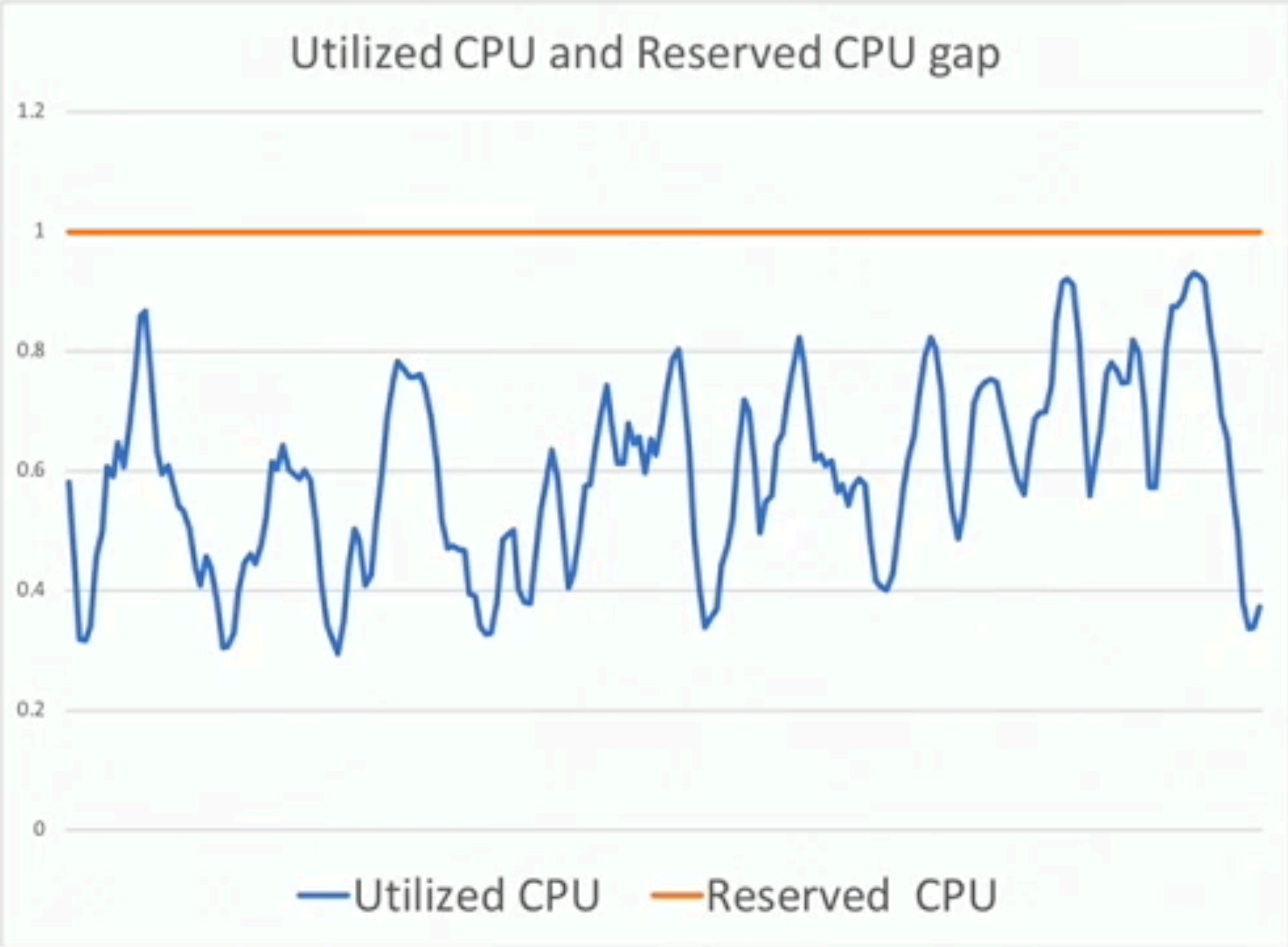
$$\text{cpu efficiency} = (\text{cpu time}) / (\text{cpu reservation time})$$

CPU Inefficient Application Example

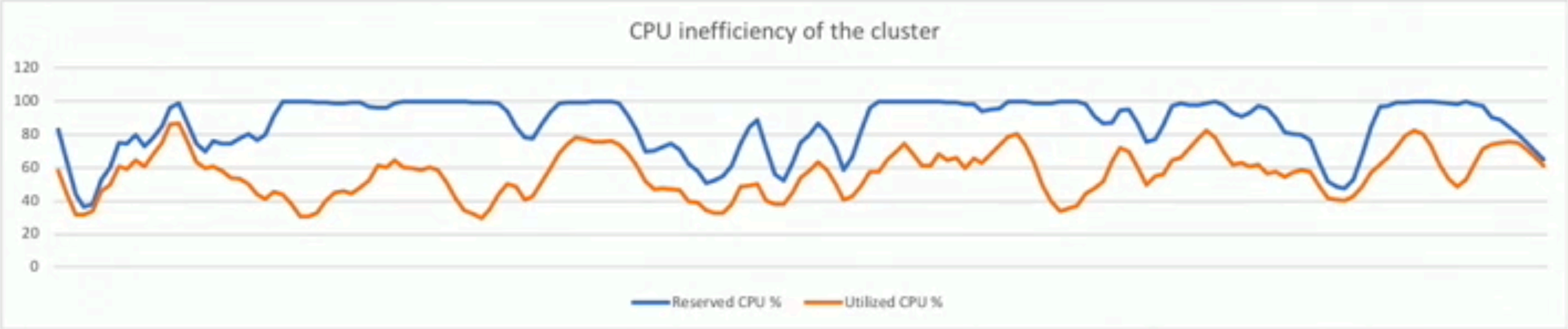



```
INSERT INTO output_table
SELECT  *
FROM big_table
ORDER   BY   column1
```

CPU Inefficient Application

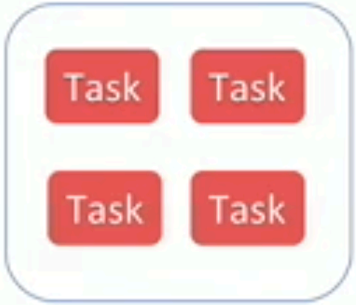


Why CPU Efficiency Matters?

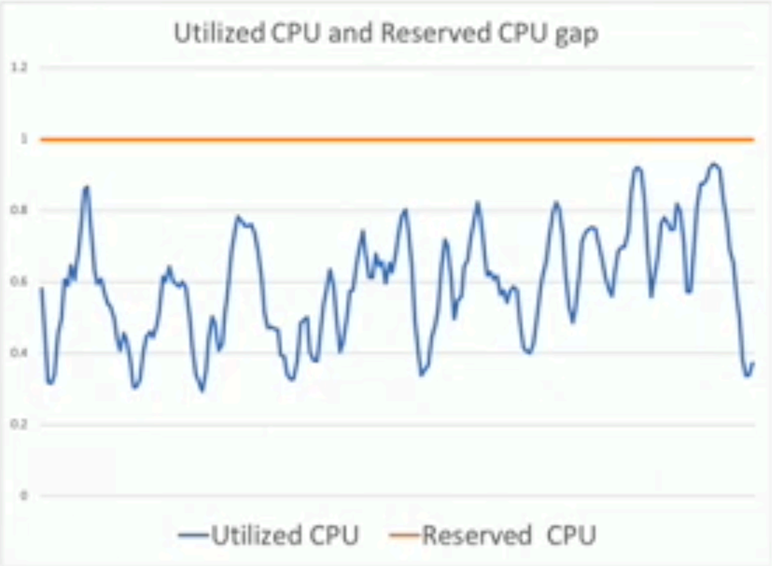


CPU Oversubscription

Before CPU oversubscription

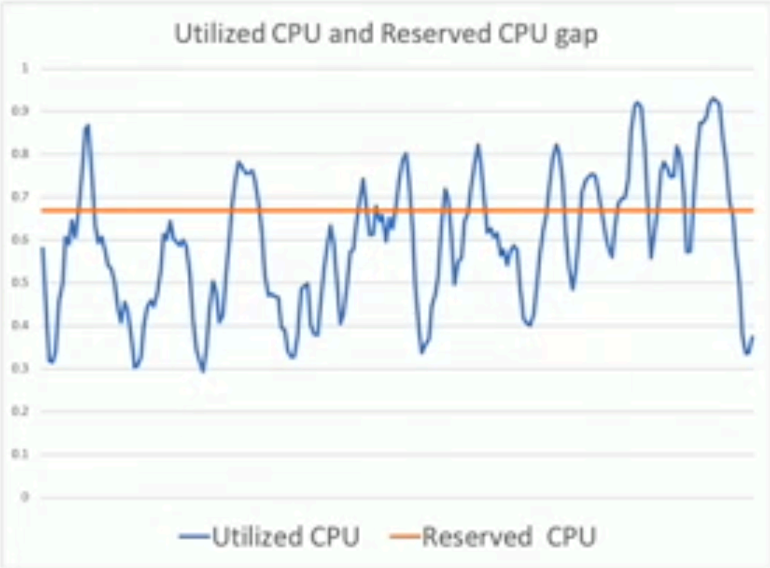


Cores = 4
spark.task.cpus = 1



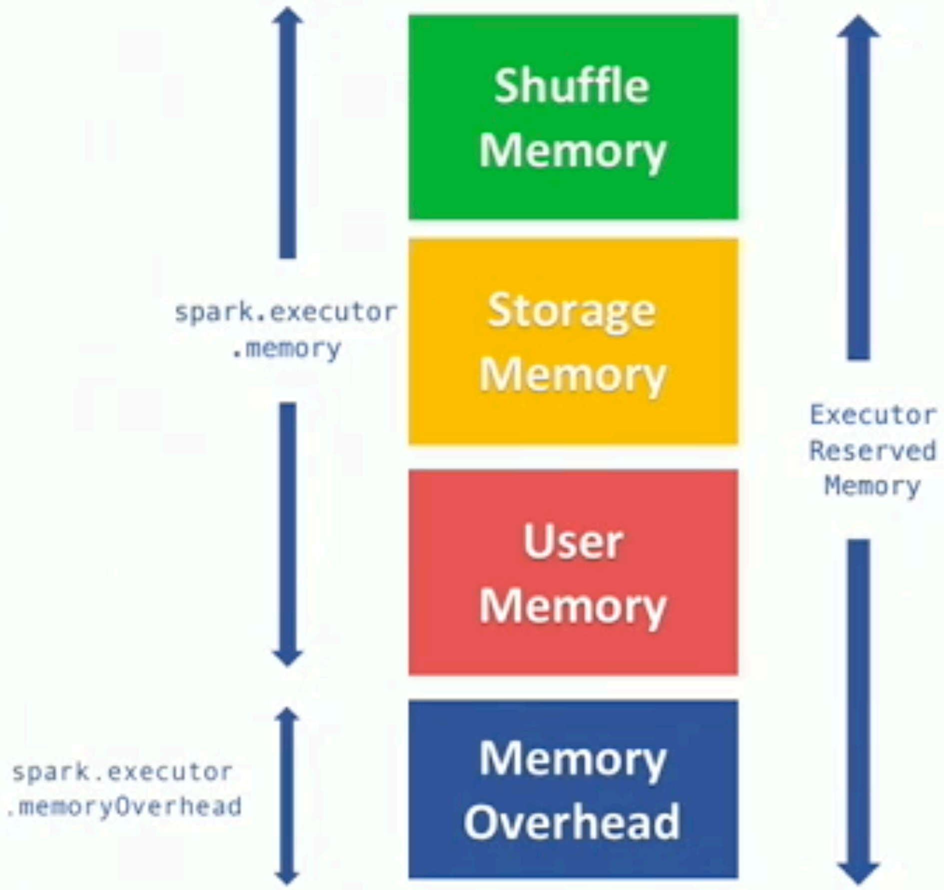
After CPU oversubscription

Cores = 4
spark.task.cpus = 0.66



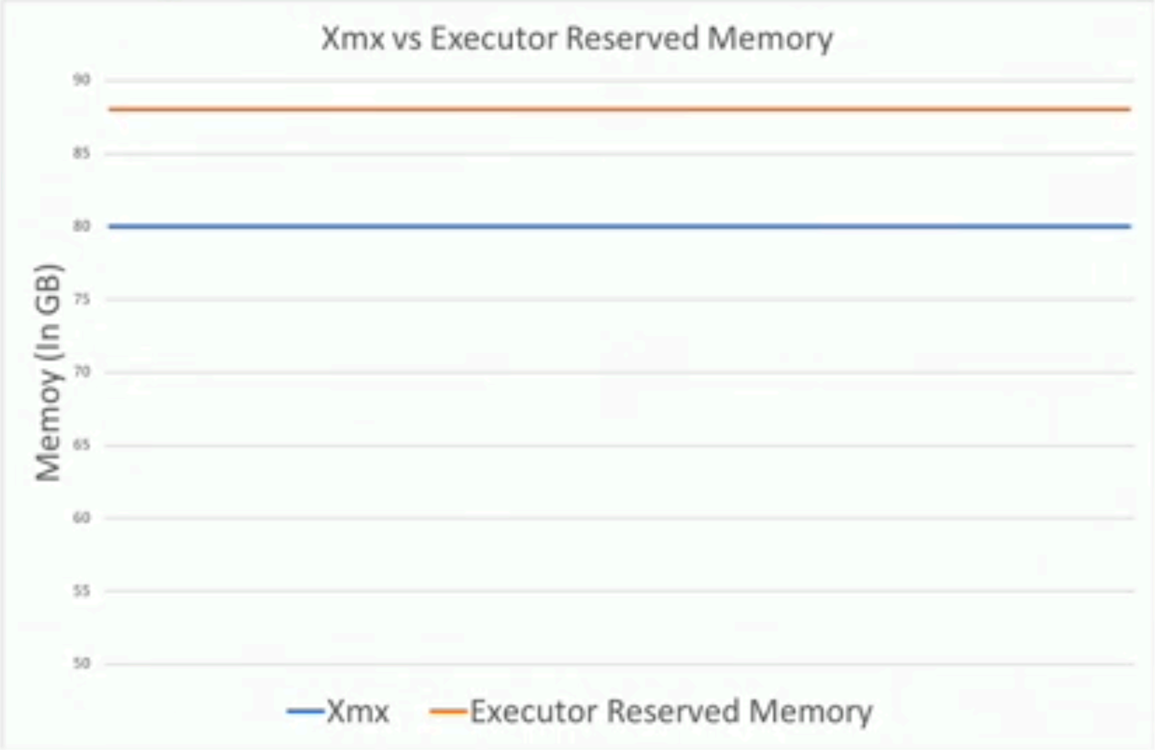
Spark Memory Model

Spark Memory Model



- Executor JVM with *XMX* equal to *spark.executor.memory*
- Concurrent running tasks share the memory pool.
- Memory reserved per task = Container Reserved Memory / Tasks per executor

Spark Memory Model



- Memory reserved for the executor is sum of Java heap size (XMX) and memory overhead factor (default is 7%)

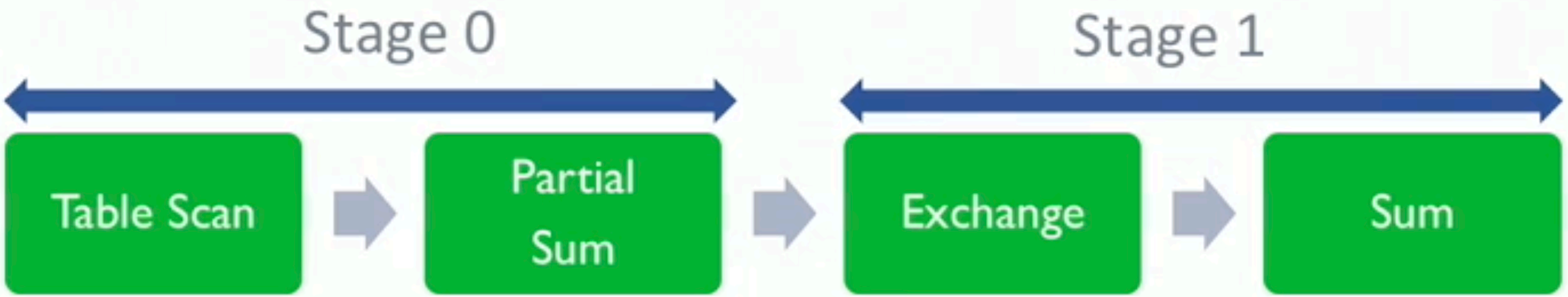
Memory Efficiency Metrics

Memory Efficiency

- Memory Time – Actual memory usage from the perspective of the OS
- Memory Reservation Time - Allocated memory from the perspective of Cluster Manager

memory efficiency = (memory time) / (memory reservation time)

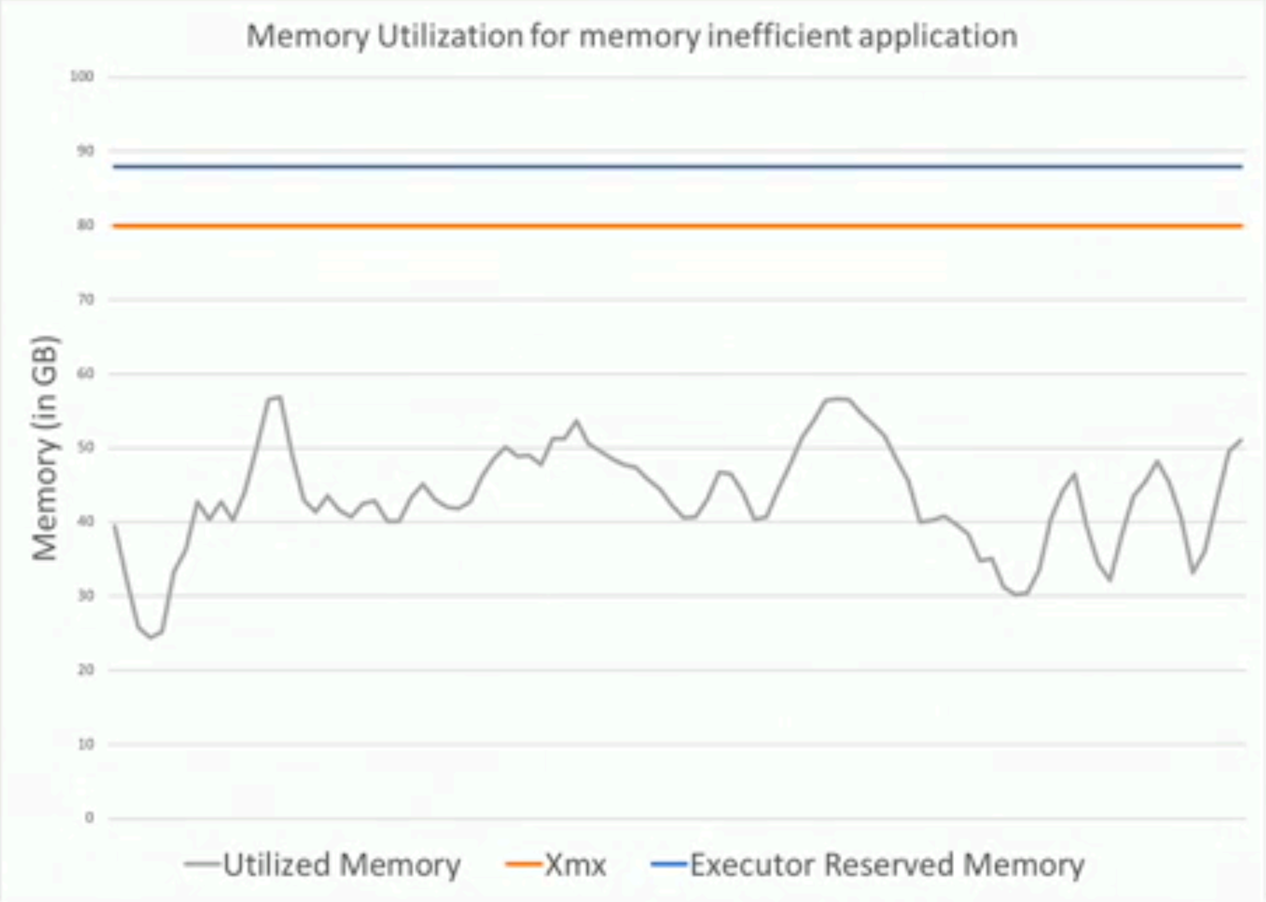
Memory Inefficient Application Example




```
SELECT  count(*)
FROM table
```

- Shuffle memory stores only the aggregated sum
- High shuffle memory can lead to significant memory wastage
- Need to tune shuffle memory for better memory efficiency

Memory Inefficient Application



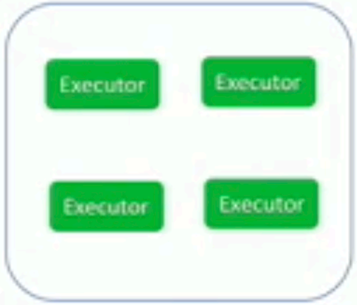
- Applications can be made memory efficient by tuning various memory configurations (XMX, reserved memory and memory fraction)
- Manually tuning each and every application is not scalable

Need for Memory Oversubscription

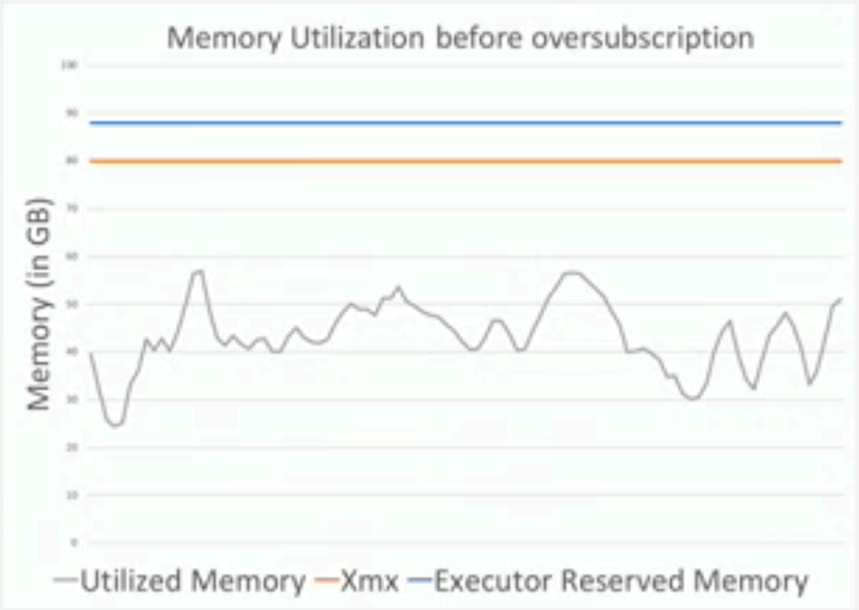
- Tune Reserved Memory so that its close to the utilized memory
- Reserved Memory can be smaller than XMX
- Need to change the Cluster Manager behavior to allow executors temporarily go over reserved memory

Memory Oversubscription

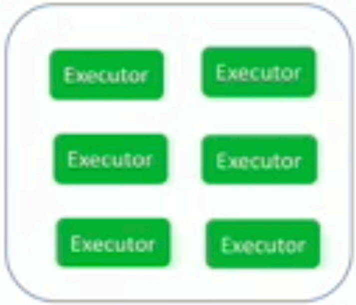
Before Memory oversubscription



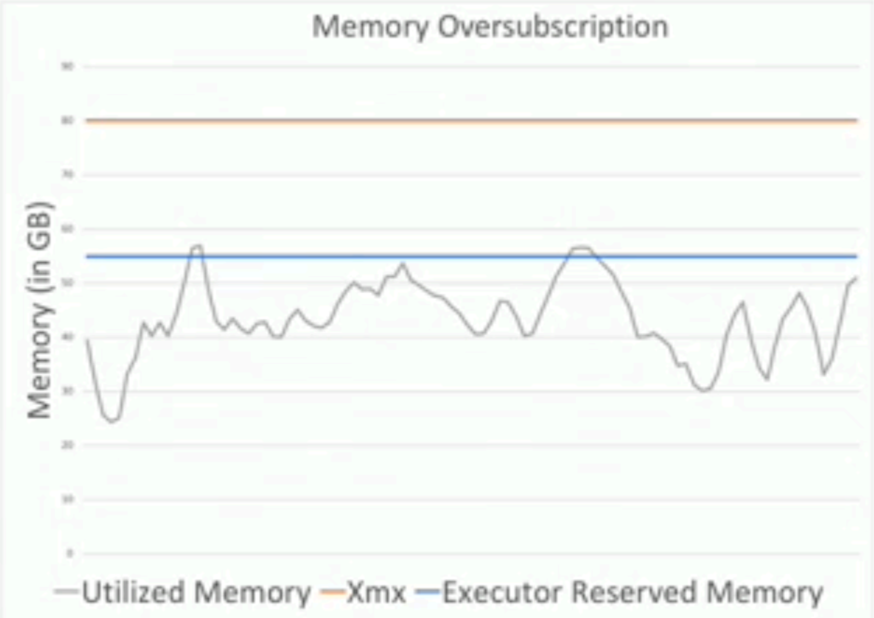
Total Memory = 32GB
spark.executor.memory = 7GB
Reserved memory = 8GB



After Memory oversubscription



Total Memory = 32GB
spark.executor.memory = 7GB
Reserved Memory = 5GB



Need for History-based Resource Auto-tuning

- Default configurations for all jobs lead to significant CPU and Memory underutilization
- More than 15k unique periodic jobs run on Spark daily with different resource requirement
  - Manual tuning each job individually is not scalable
- Jobs are periodic in nature and resource usage pattern does not change significantly
  - Leverage history to predict the resource usage for next run

History-based Resource Auto-tuning

High-level Idea

- Most workload is generated by **periodic** jobs
- For each periodic job, **predict memory and CPU requirements per executor** based on utilization in previous job runs
- Request containers from Cluster Manager based on prediction

Standard container

Request

4 CPU cores, **8GB** RAM

spark.executor.cores = 4
spark.task.cpus = 1
spark.executor.memory = 8g

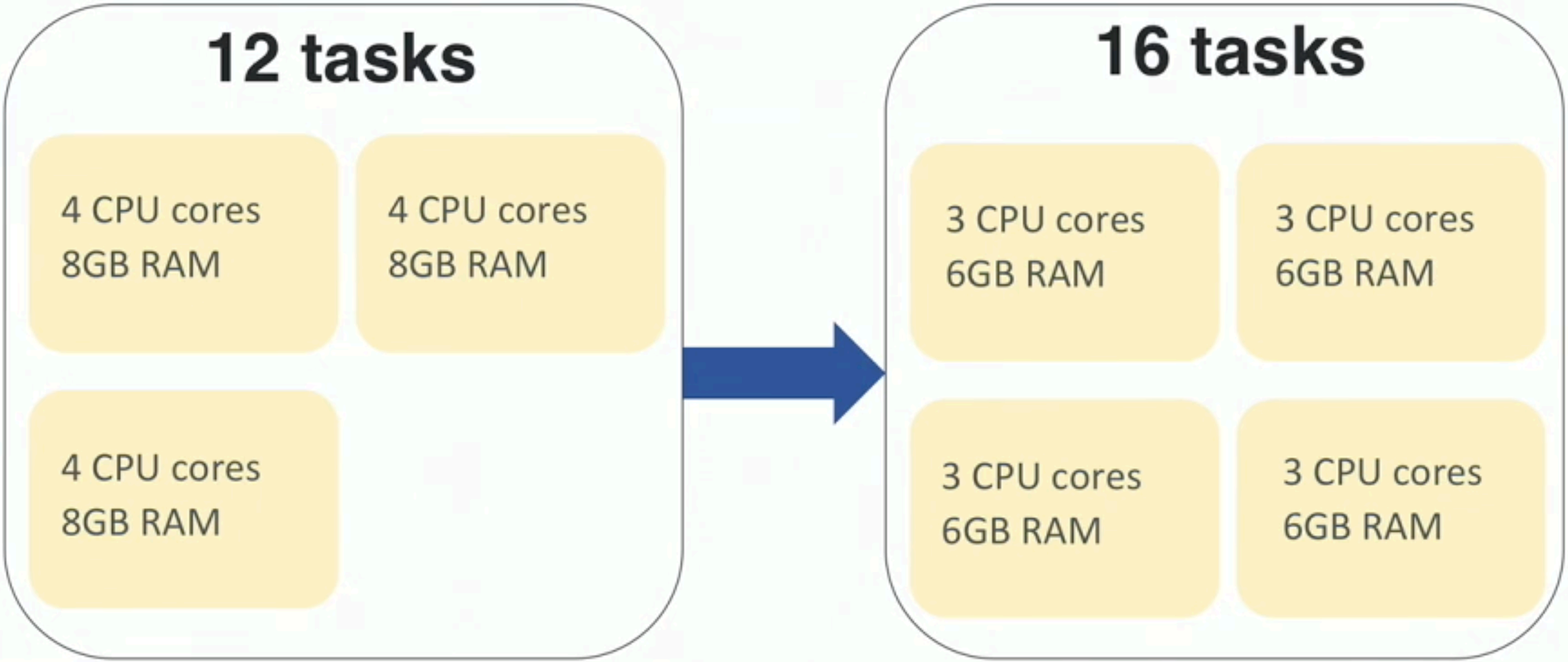
Oversubscribed container

Request

3 CPU cores, **6GB** RAM

spark.executor.cores = 4
spark.task.cpus = 1
spark.executor.memory = 8g

Better Container Packing: Machine with 12 CPU Cores and 24GB of RAM



Cluster Manager Requirements

- Should allow fractions of CPU to be allocated (example: 3.6 cores)
- Should periodically log resource usage stats per each container

Architecture

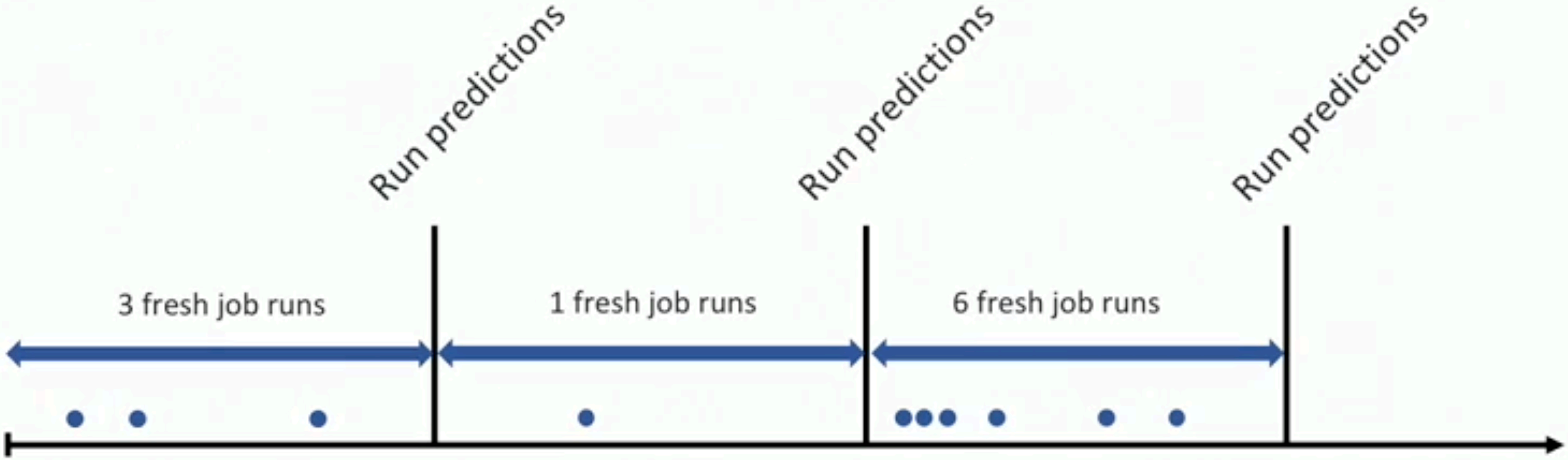
Two independent phases:

- Computing and persisting predicted memory and CPU per executor for periodic jobs (hourly)
- Apply predictions in Spark Driver when job launches and requests containers from Cluster Manager

Applying Predictions

Computing and Persisting Predictions

Resource predictions are run hourly in order to include jobs that finished since last computation



Inputs for Predictions

For each periodic job run:

- CPU time and CPU reservation time
- Executors max used memory stats
- Job identity hash to distinguish between runs of the same job

What is Job Identity Hash and Why We Need It?

- Two types of periodic jobs: SQL queries and Scala applications
- Changes to query or Scala implementation can significantly change memory and CPU footprint
- Same is true for memory, CPU, and other spark configurations, like *spark.sql.shuffle.partitions*
- Job identity hash represents state of job's **implementation** and **configuration**

Example for a SQL job

```sql
INSERT OVERWRITE TABLE spark_test
PARTITION (ds = '2017-06-05')
SELECT id, title
FROM spark_talks
WHERE LOWER(title) like '%memory%'
```

Identity hash: 123456

```sql
INSERT OVERWRITE TABLE spark_test
PARTITION (ds = '2017-06-06')
SELECT id, title
FROM spark_talks
WHERE LOWER(title) like '%memory%'
```

Identity hash: 123456

```sql
INSERT OVERWRITE TABLE spark_test2
PARTITION (ds = '2017-06-06')
SELECT speaker, count(*) as num_talks
FROM spark_talks
GROUP by 1
```

Identity hash: 642352

Prediction algorithm

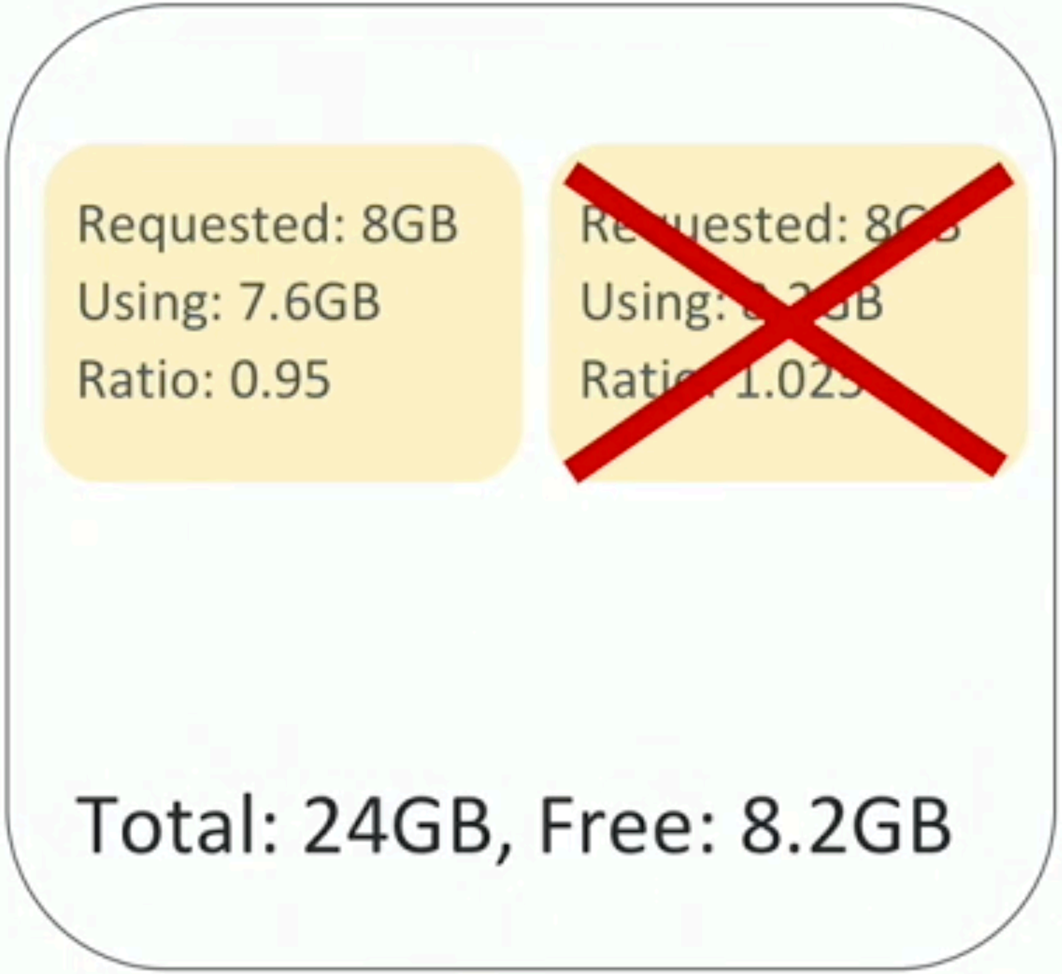
- Prediction is computed for each job identity hash separately
- For each run in the past 10 days, obtain CPU and memory usage aggregates:
  - For memory: ***`p99 of max used memory bytes`*** across all containers
  - For CPU: ***`(total CPU time)/(Reserved CPU time) * (actual CPU cores)`***
- Separately for CPU and memory:
  - Sort values in ascending order
  - Do line smoothing to avoid outliers
  - Take p90 of values – **will be used in the next run of the job**

Potential issues

- Increased CPU time due to context switching
- Containers could be killed by Cluster Manager if used memory exceeds requested

More on container kills

- Periodic jobs are not exactly the same – they run on different data
- Memory usage can go up compared to previous runs of the job
- In classic approach, Cluster Manager kills container if memory usage goes over the limit



How to reduce container kills?

- On Cluster Manager side, allow containers to go over the limit
- When machine is running out of memory, kill the highest offender

Low load on cluster

Requested: 8GB
Using: 8.8GB
Ratio: 1.1

Requested: 8GB
Using: 9.6GB
Ratio: 1.2

Total: 24GB, Free: 5.6GB

High load on cluster

Requested: 8GB
Using: 8.8GB
Ratio: 1.1

~~Re[illegible]uested: 8[illegible]B~~
~~Using: [illegible]GB~~
~~Rati[illegible]: 1.2~~

Requested: 8GB
Using: 5.6GB
Ratio: 0.7

Free: 0GB

How else to reduce container kills?

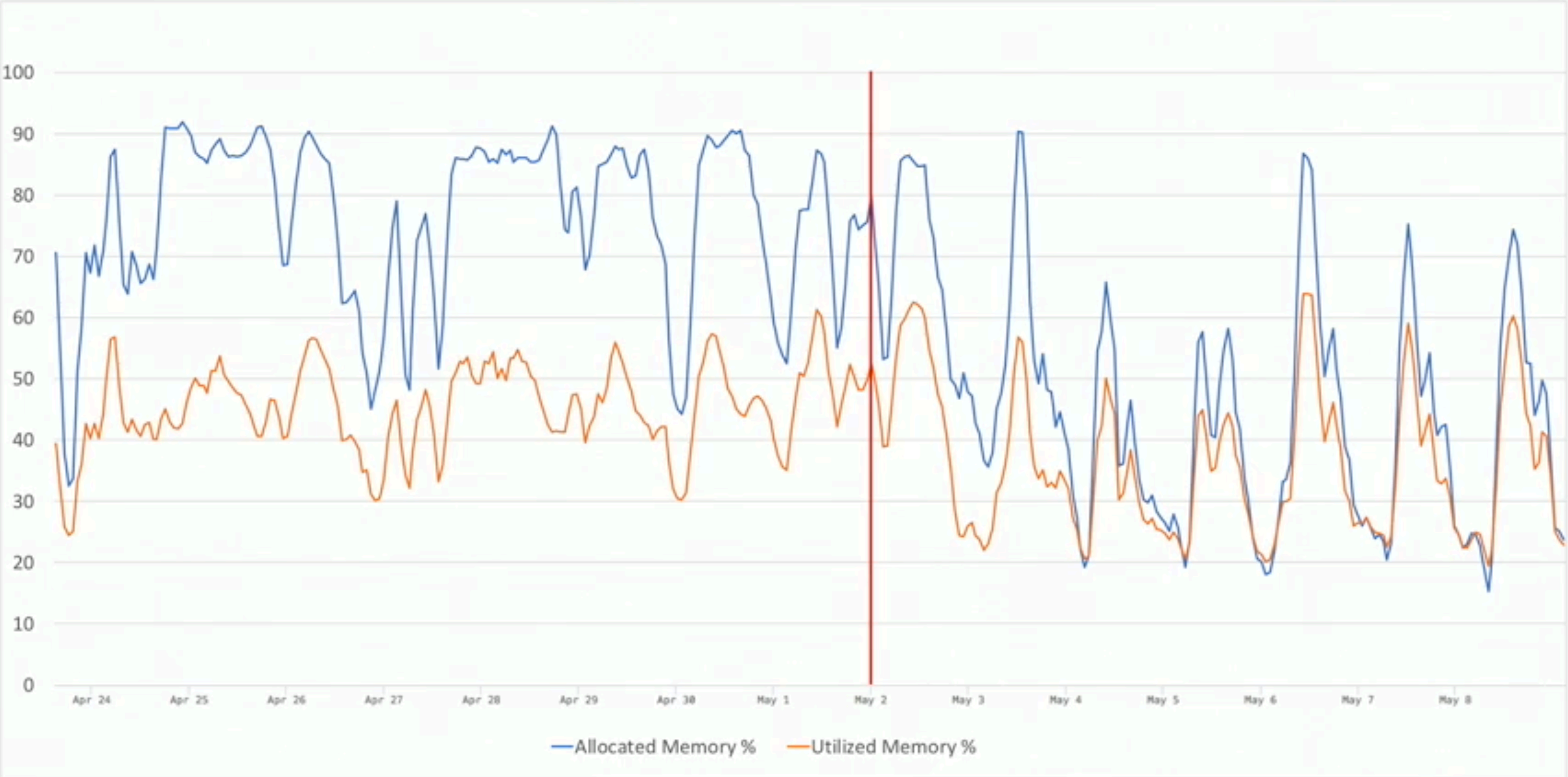
- Introduce a threshold for the maximum number of container kills due to resource quota exceeded
- When number of container kills exceeds the threshold, dynamically disable resource tuning while job is still running
- Next iteration of prediction computation will take the higher memory usage into account

Results

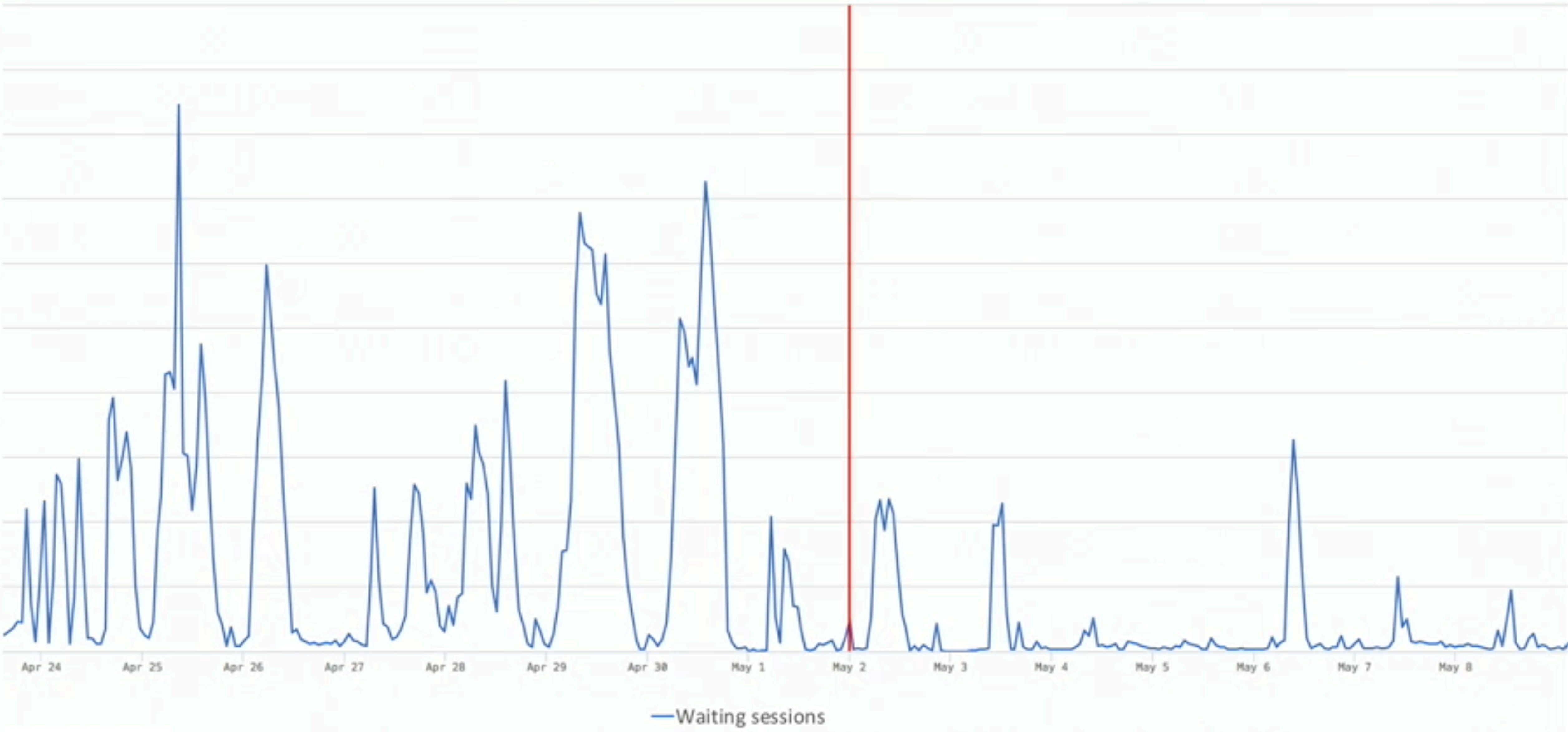
Improvements In Cluster Metrics

Allocated CPU % vs Utilized CPU %

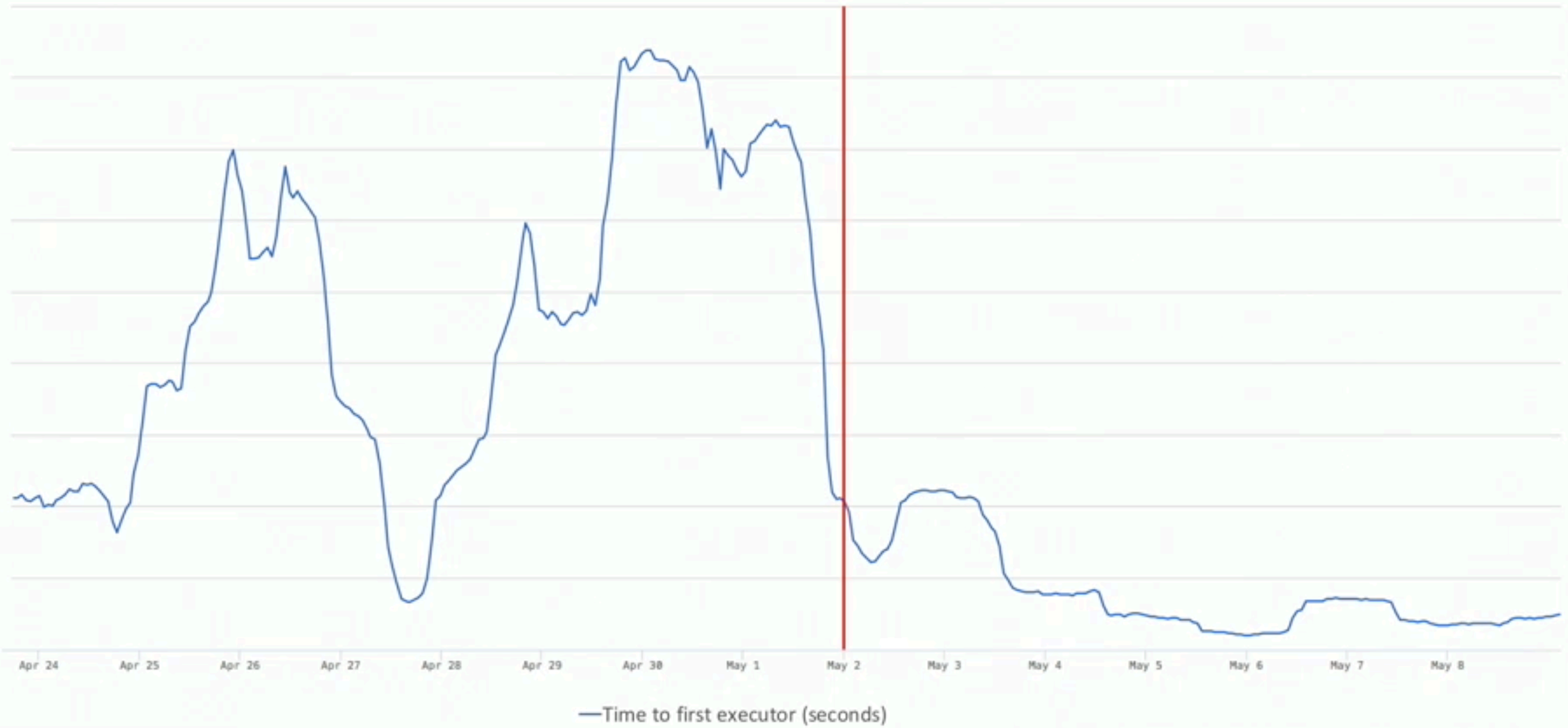
Allocated Memory % vs Utilized Memory %



Cluster Backlog



Resource Waiting Time

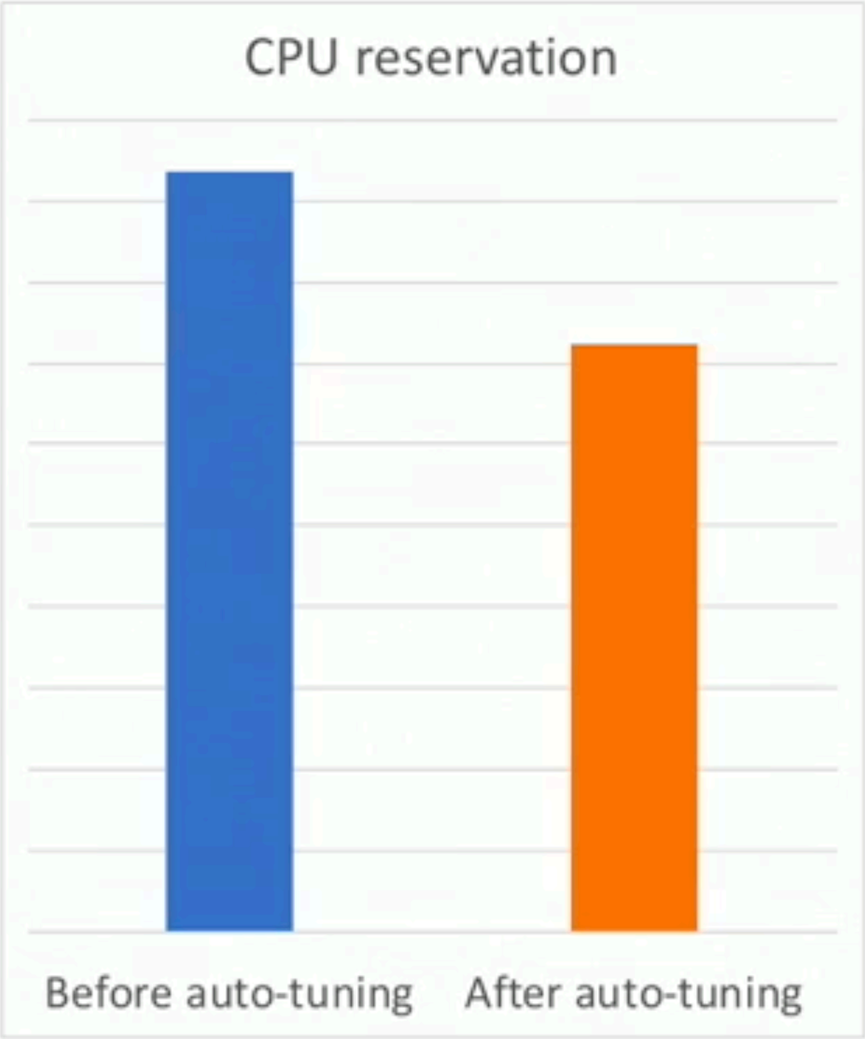


Improvements In Performance Of Jobs

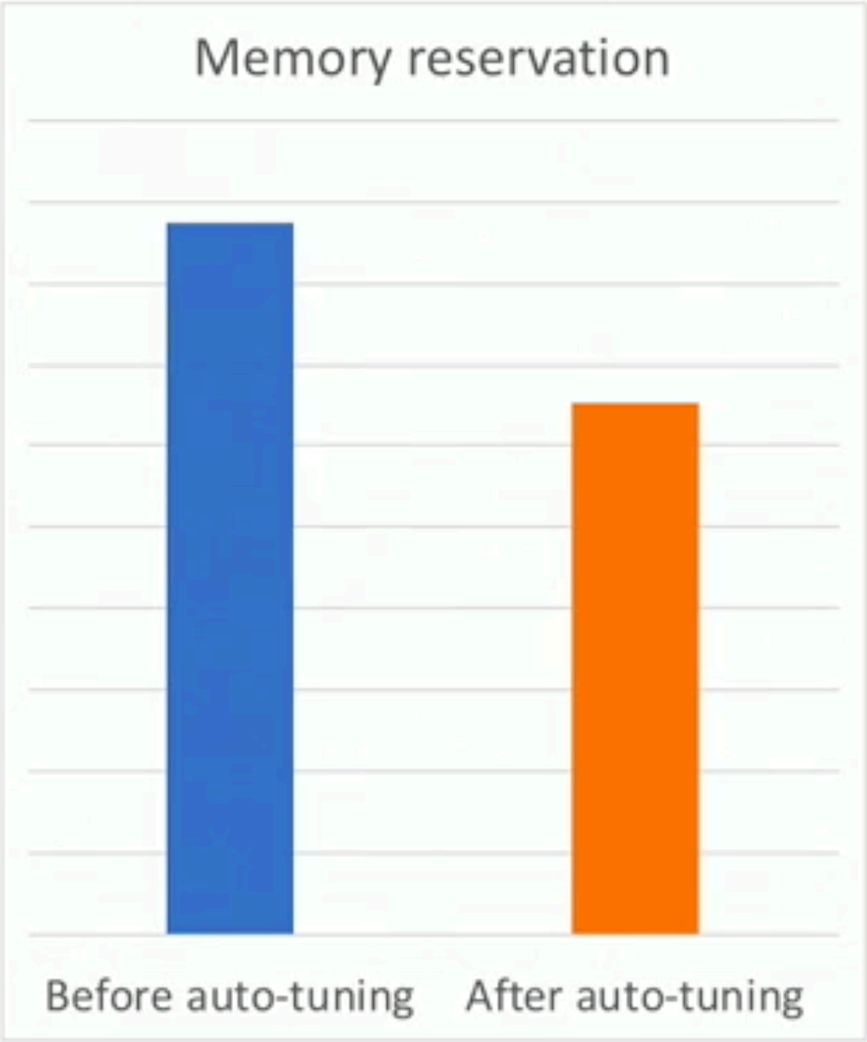
How comparison was made

- Took two time segments of **3 day length**, one without auto-tuning, the other – with auto-tuning
- Defined a set of common jobs that ran during both segments (over 1000 jobs)
- For each segment, computed averages for key metrics (s.a. CPU time, CPU reservation time, memory reservation time)

CPU & Memory Reservation

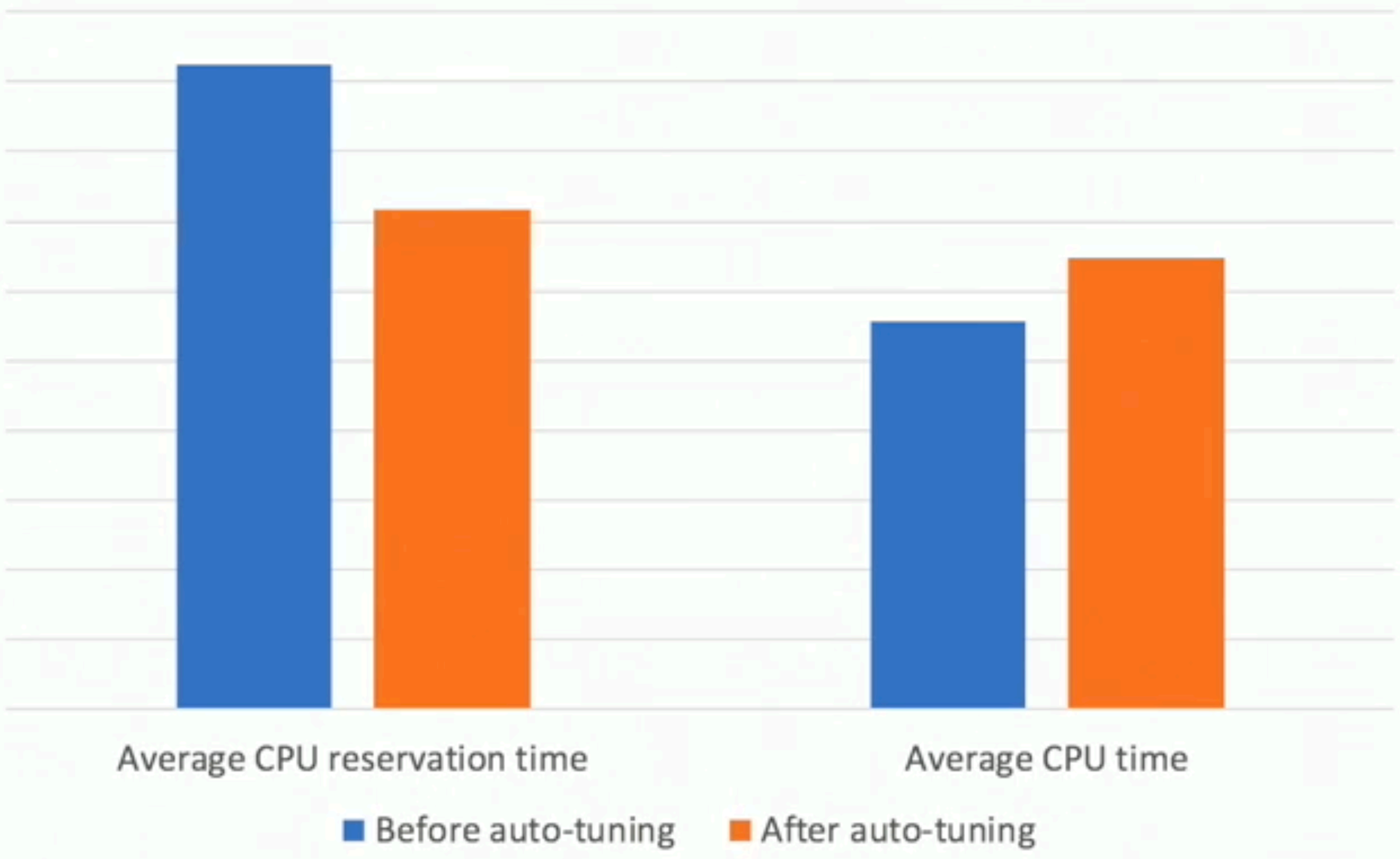


Reduced by 22%

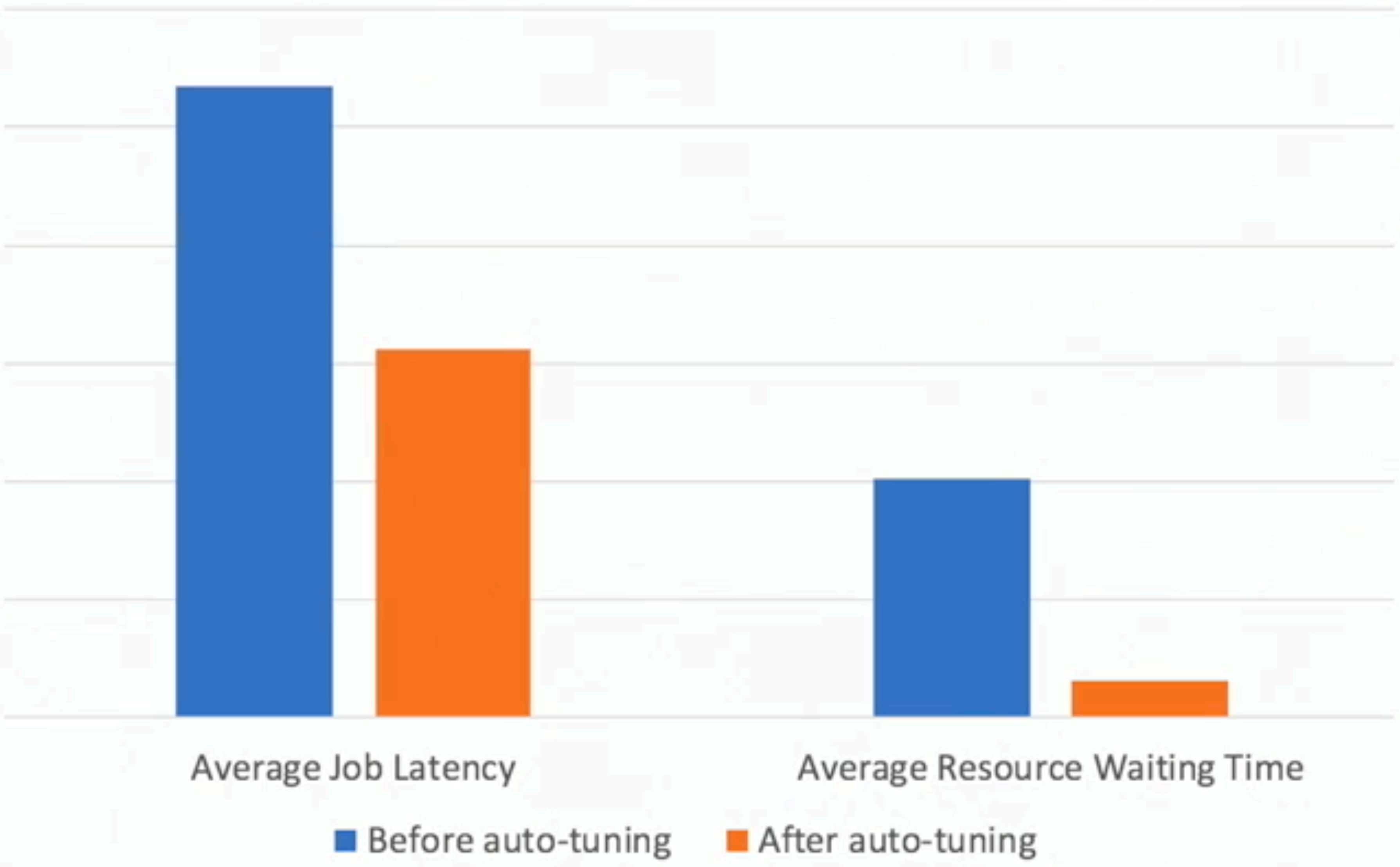


Reduced by 25%

CPU Reservation & CPU Time



Job Latency & Resource Waiting Time



Questions?