



Cloud Native Java with OpenJ9:

Fast, Lean and
Definitely Mean



About me

Steve Poole

IBM Lead Engineer / Developer advocate

Making Java Real Since Version 0.9

Open Source Advocate

DevOps Practitioner (whatever that means!)

Driving Change

@spoole167



Eclipse OpenJ9

Created Sept 2017

<http://www.eclipse.org/openj9>

<https://github.com/eclipse/openj9>

Dual License:

Eclipse Public License v2.0

Apache 2.0

Users and contributors very welcome

<https://github.com/eclipse/openj9/blob/master/CONTRIBUTING.md>



Outline

- This talk is about a JVM but its also about evolution, new environments, new economics and new opportunities



Evolution

This is a manatee

Lives in warm shallow waters .

Has no natural predators

Sleeps 50% of the day

Moves at 3 MPH

They may look like they
have blubber but its
actually stomach and
intestines..





In a stable environment
creatures evolve to an
optimal shape and
behavior.

The best economic fit.
The most reward for the
least effort.



The best shape for a
manatee is round

Staying in the same place can make you over fit

- The manatee will die if the temperature gets too cold.
- They need water at least 18 degrees C
- Their environment is surrounded by cold water so moving out will kill them
- Staying where they are will eventually lead to extinction if they don't evolve
- But evolution comes from new challenges.
- If their environment changes too fast it will kill.



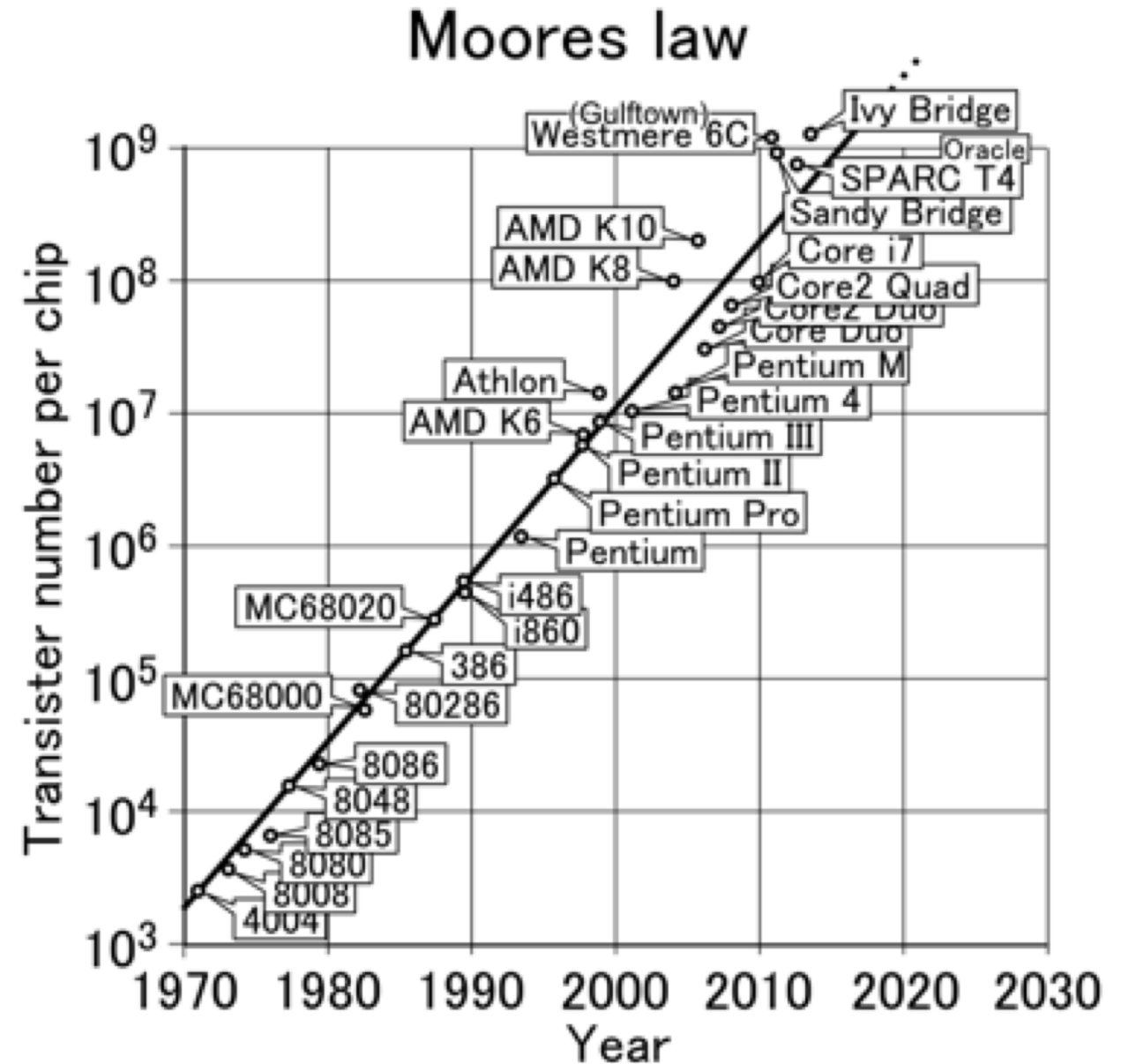


What's that got to do with the JVM?

@spoole167

For a very long
time Java and the
JVM have had one
evolutionary
pressure

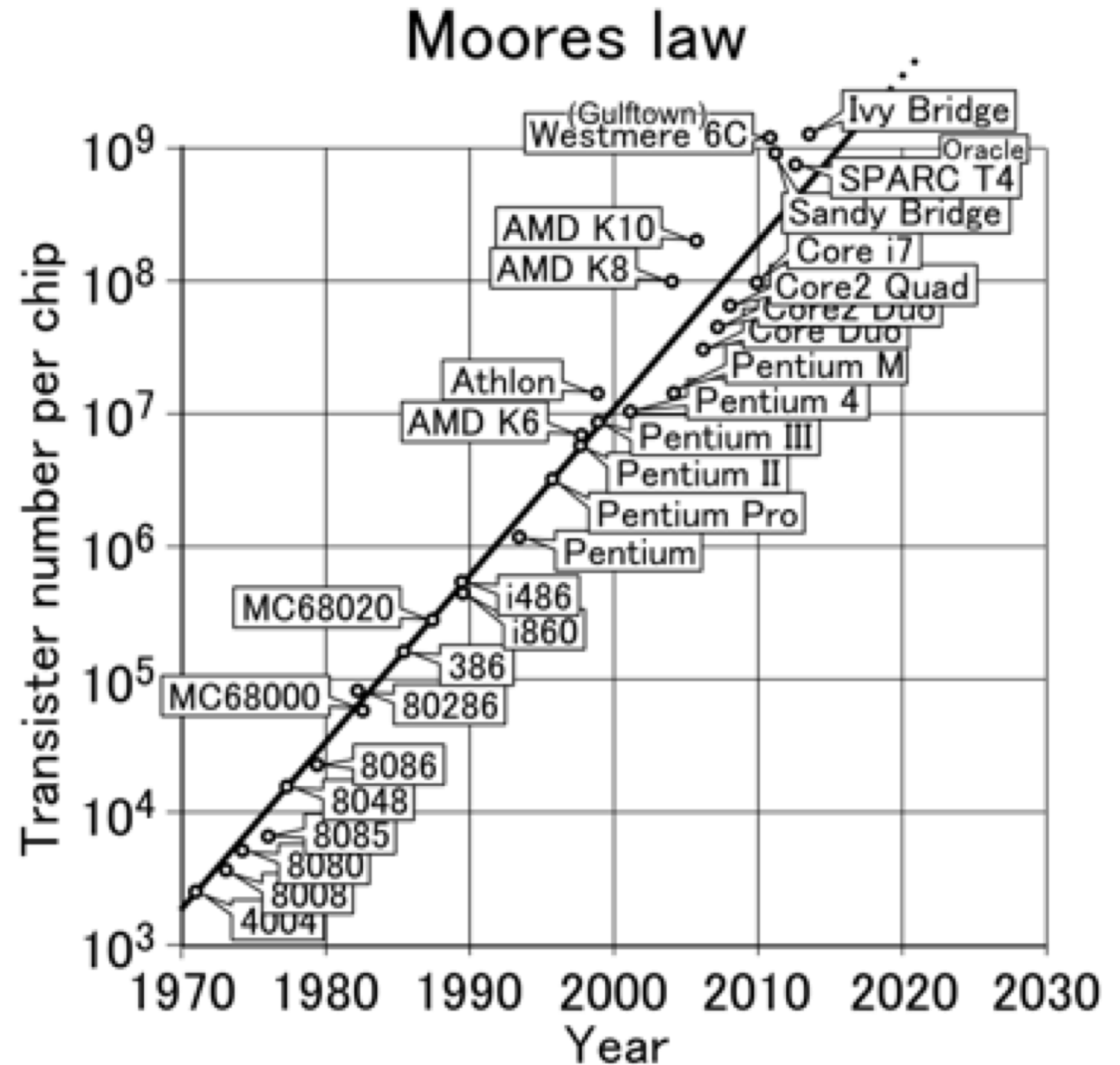
@spoole167

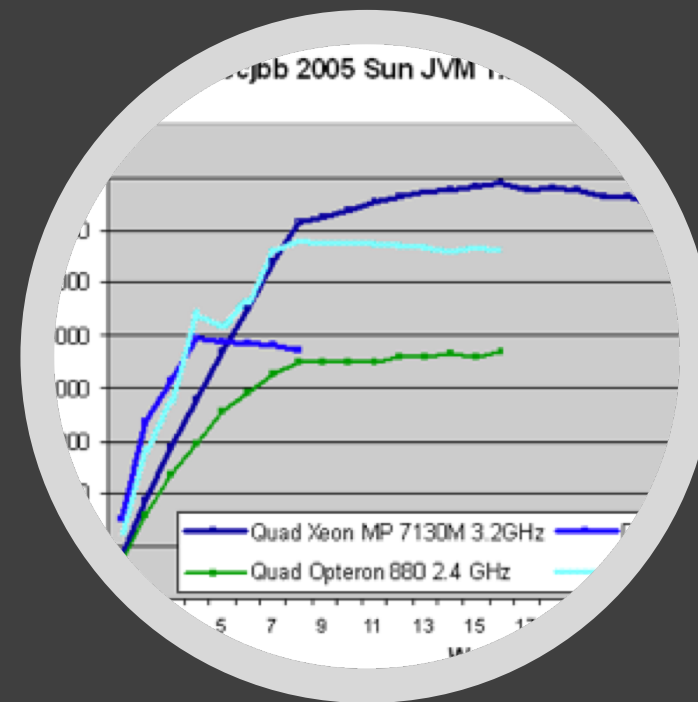
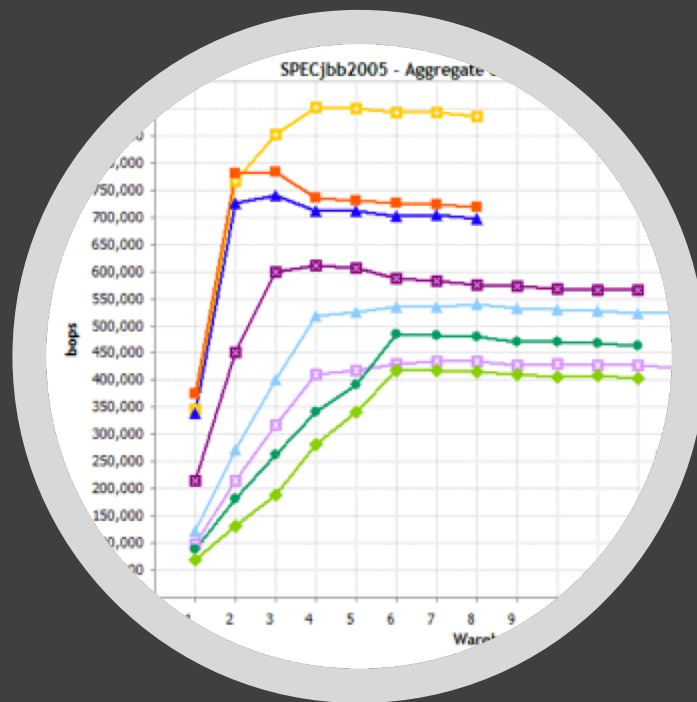
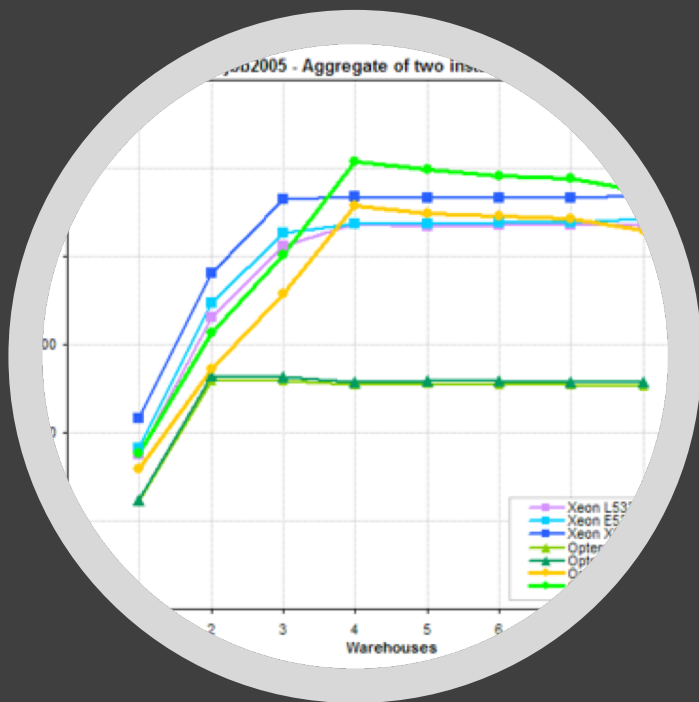


Maximize the opportunities offered by Moore's Law

Forever...

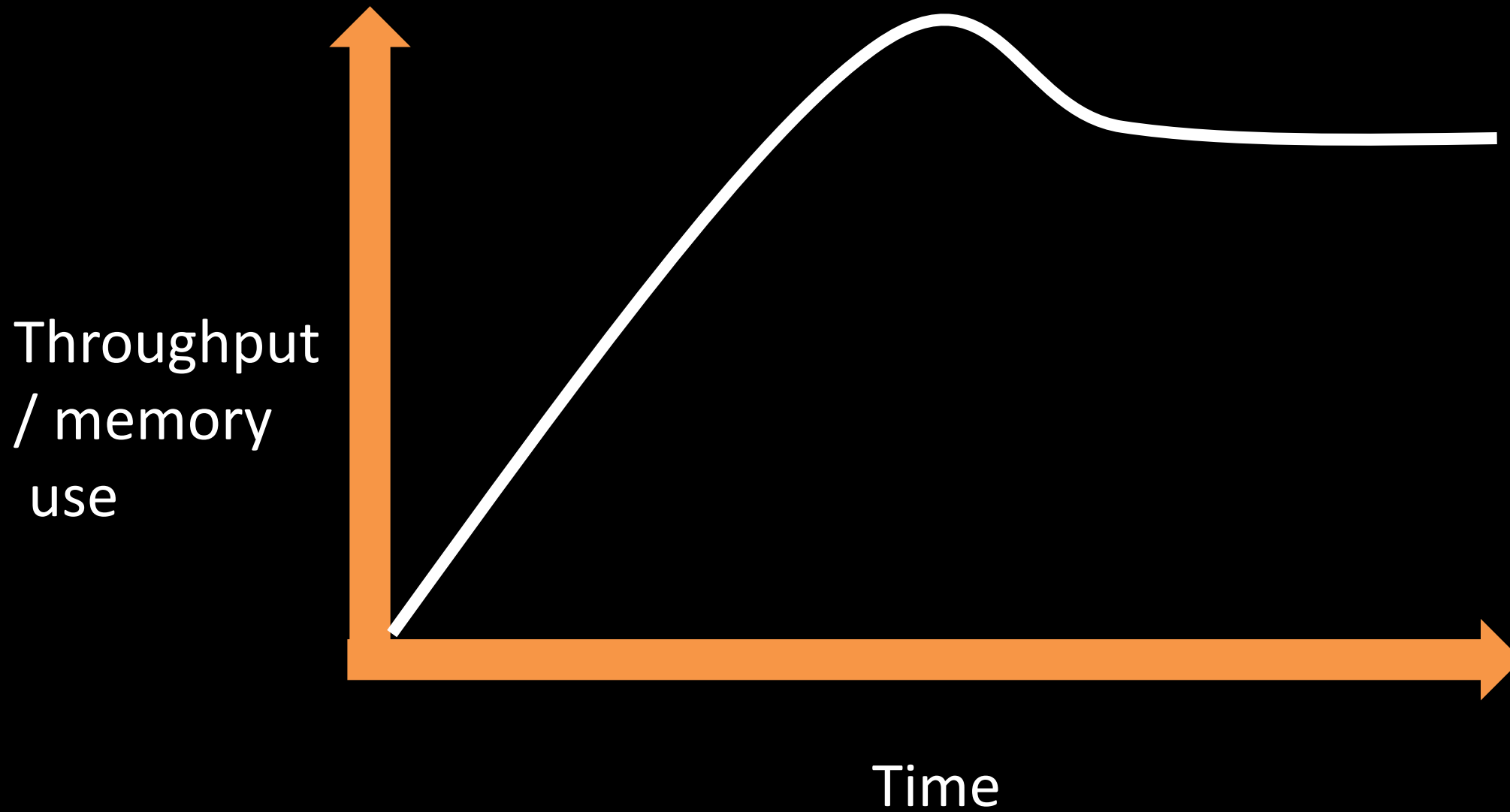
@spoole167





Java has tuned itself to this pressure

Traditional profile





Java has tuned itself
to this need

With great names like

Java SE 5 'tiger'

Java SE 6 'mustang'

Java SE 7 'dolphin'



We see Java as strong like the
tiger, with stamina like the
mustang and fast like the dolphin



Even when moores law broke

- The JVM didn't have to do much – it already had a good multi-cpu story.
- But Java needed to change - and we added streams and lambdas..
- Not as quick to deliver as we'd like
- Just about made it.
- Our Java survived



So far our Java animals have
given us great performance

For long running, multi-core
applications

Exactly what we needed

Exactly what the economics
required.



So far our Java animals have
given us great performance

From the smallest devices to the
largest

As well as real-time and soft-real-time

An ecosystem of JVMs





But things have
changed.

A new race is here

new economics
new environments



We got cloud

What 'Cloud' promises

a *virtual, dynamic* environment
which maximizes use, is *infinitely*
scalable, always available and
needs *minimal upfront*
investment or commitment

Take your code – host it on someone else's
machine and pay **only** for the resource you use
for the time you use it

AND be able to do that very quickly and
repeatedly in parallel



“Compute on demand” – it’s what we’ve always wanted



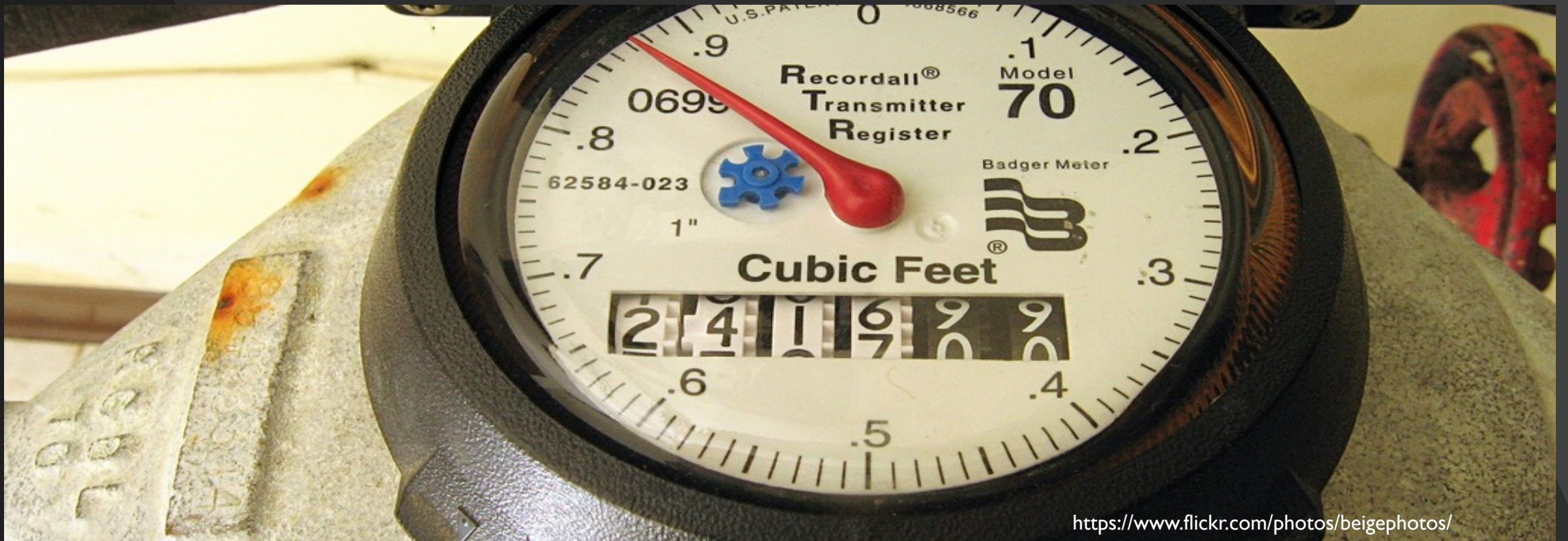
Cloud Economics

We really are getting closer all the time to
'Compute on Tap'



Cloud Economics

But with taps come meters...



Cloud computing: compute == money

Money changes everything

With a **measurable** and direct relationship between \$£€¥ and CPU/RAM, disk etc the financial success or failure of a project is even easier to see

And that means...

Even more focus on value for money.

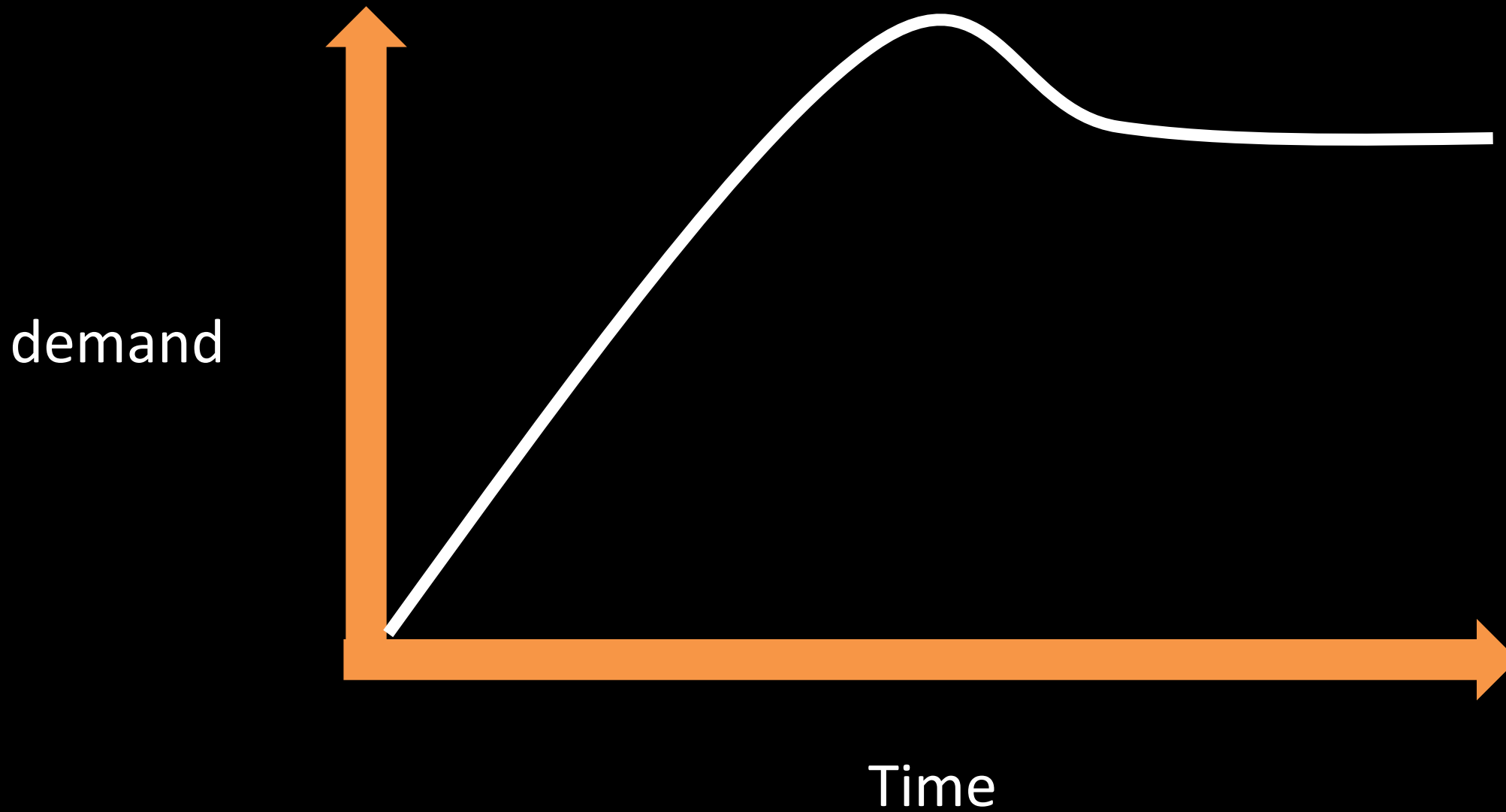


compute == money

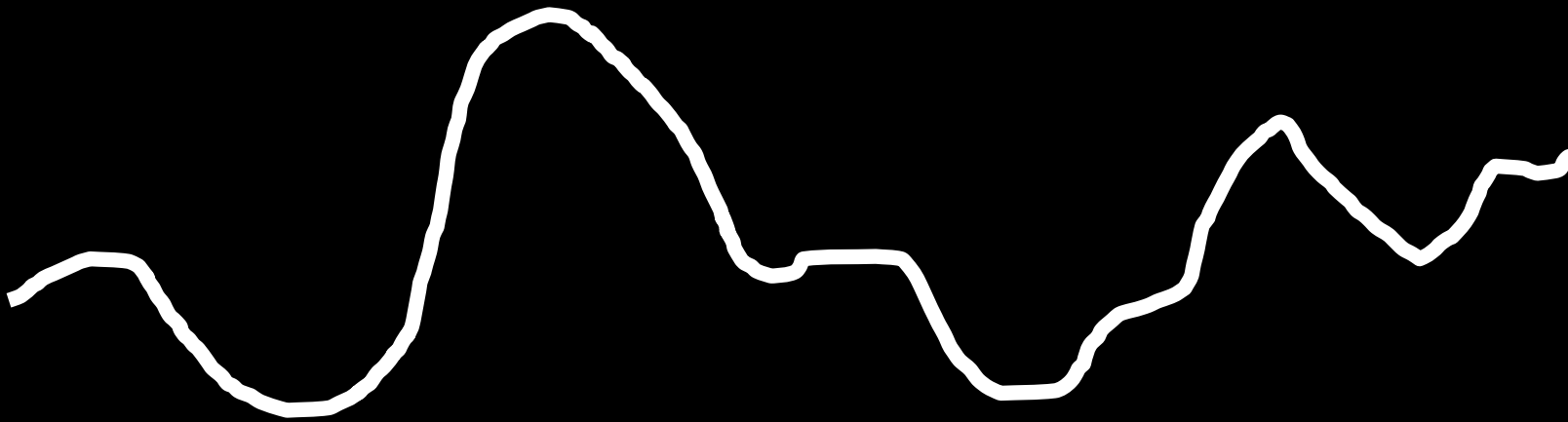
\$ == GB/hr

-Xmx: \$100

Traditional profile

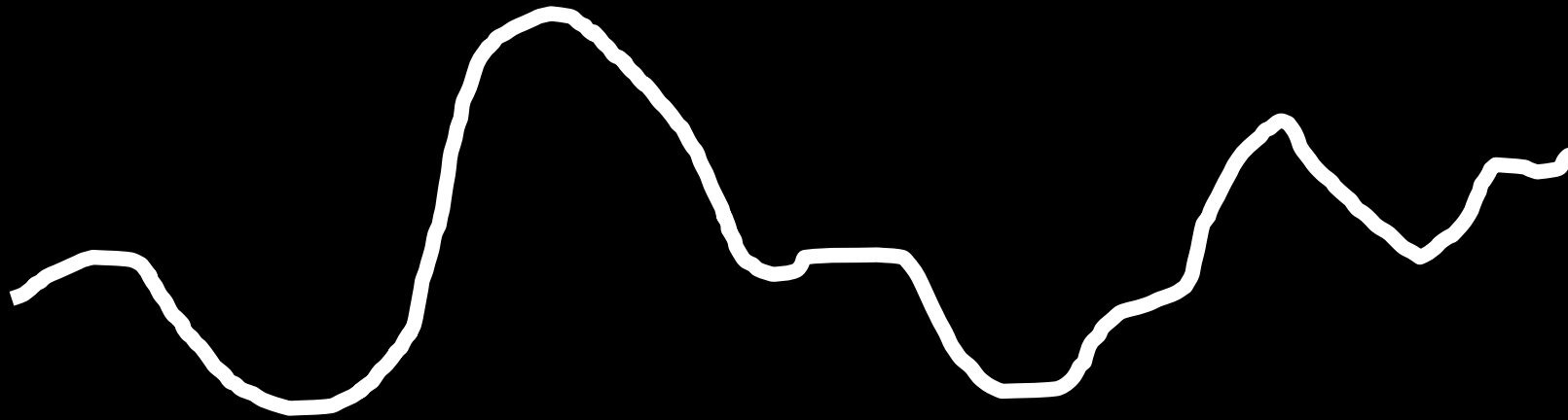


Demand



time

Demand



How does your application respond to demand?



time

Demand



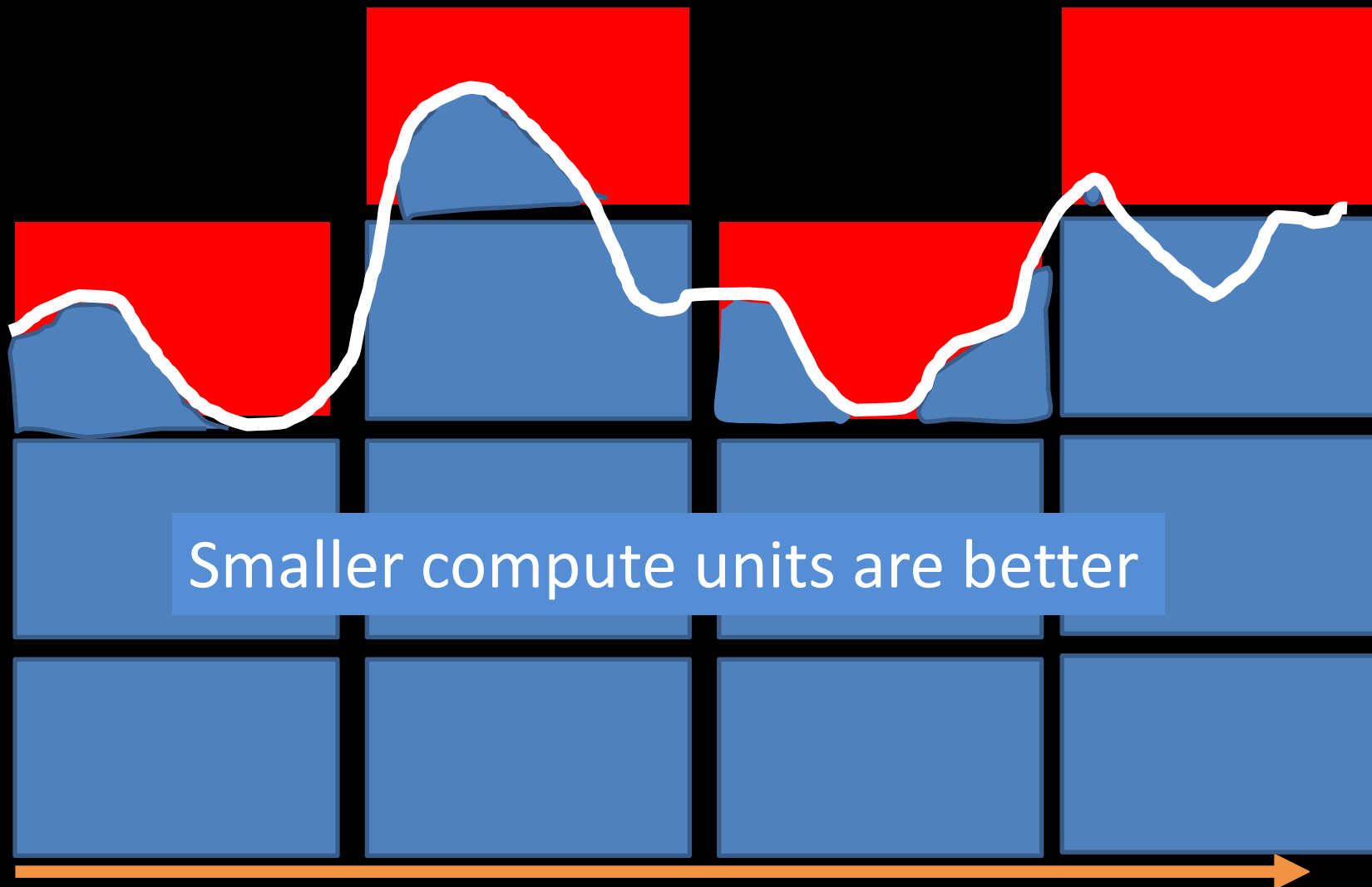
time

Demand

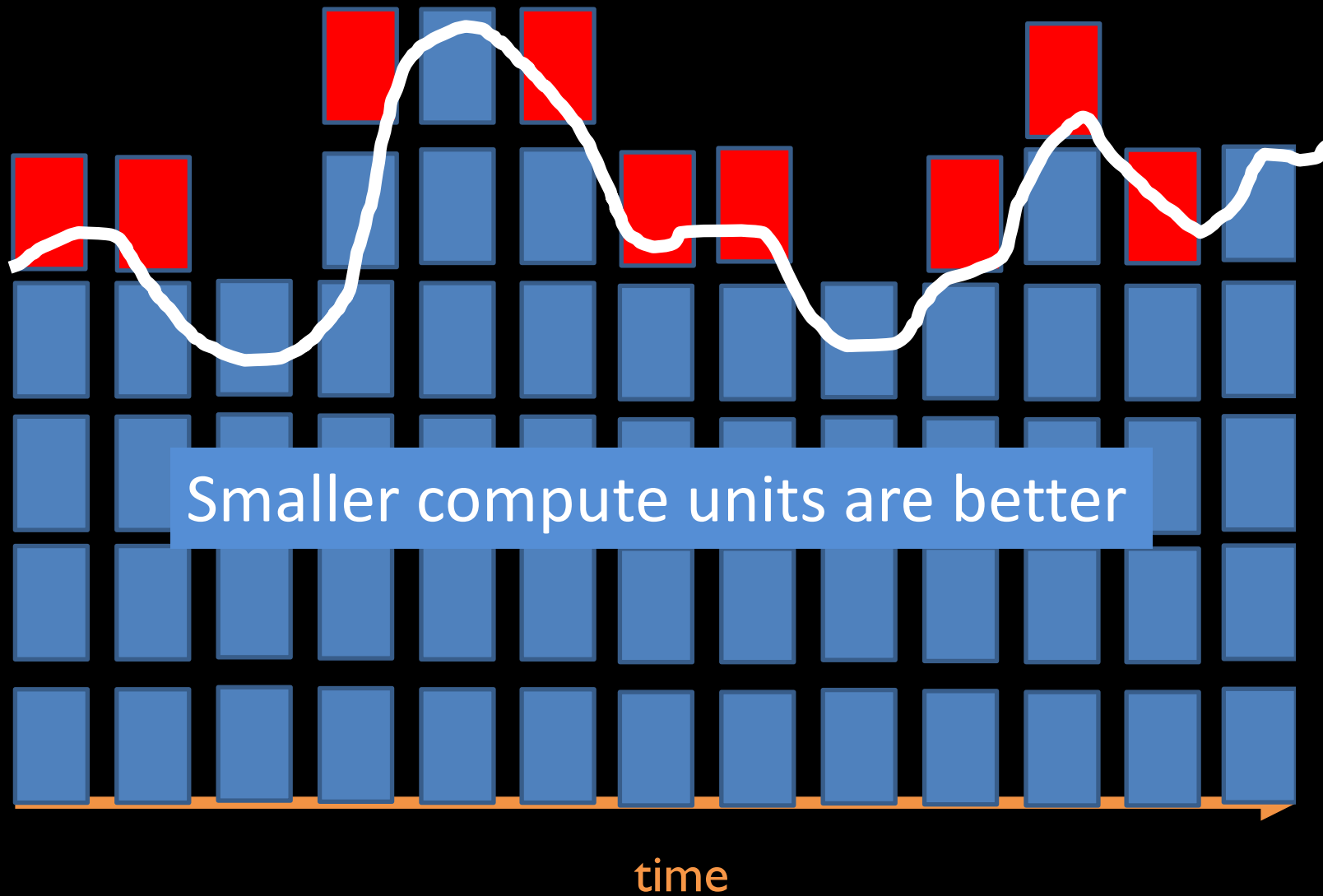
Look at all that wasted money!

time

Demand

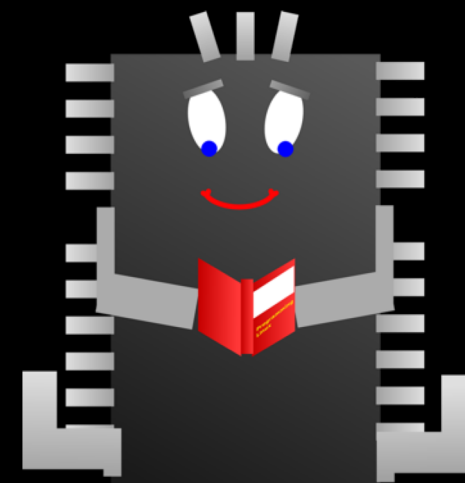


Demand



Cloud demands:

- Small runtime memory footprint
- Small deployment sizes
- Fast starting applications
- No resource usage when idle



What does that mean for the JVM and
for Java applications?

Simply

Java applications are going to be running in a **remote**, **constrained** and **metered** environment

There will be precise limits on how much disk, CPU, RAM, Bandwidth an application can use and for how long

Whether your application is large or small, granular or monolithic. Someone will be paying for each unit used

That person will want to get the most out of that investment

Your application is going on a diet
The JVM needs to change

@spoole167

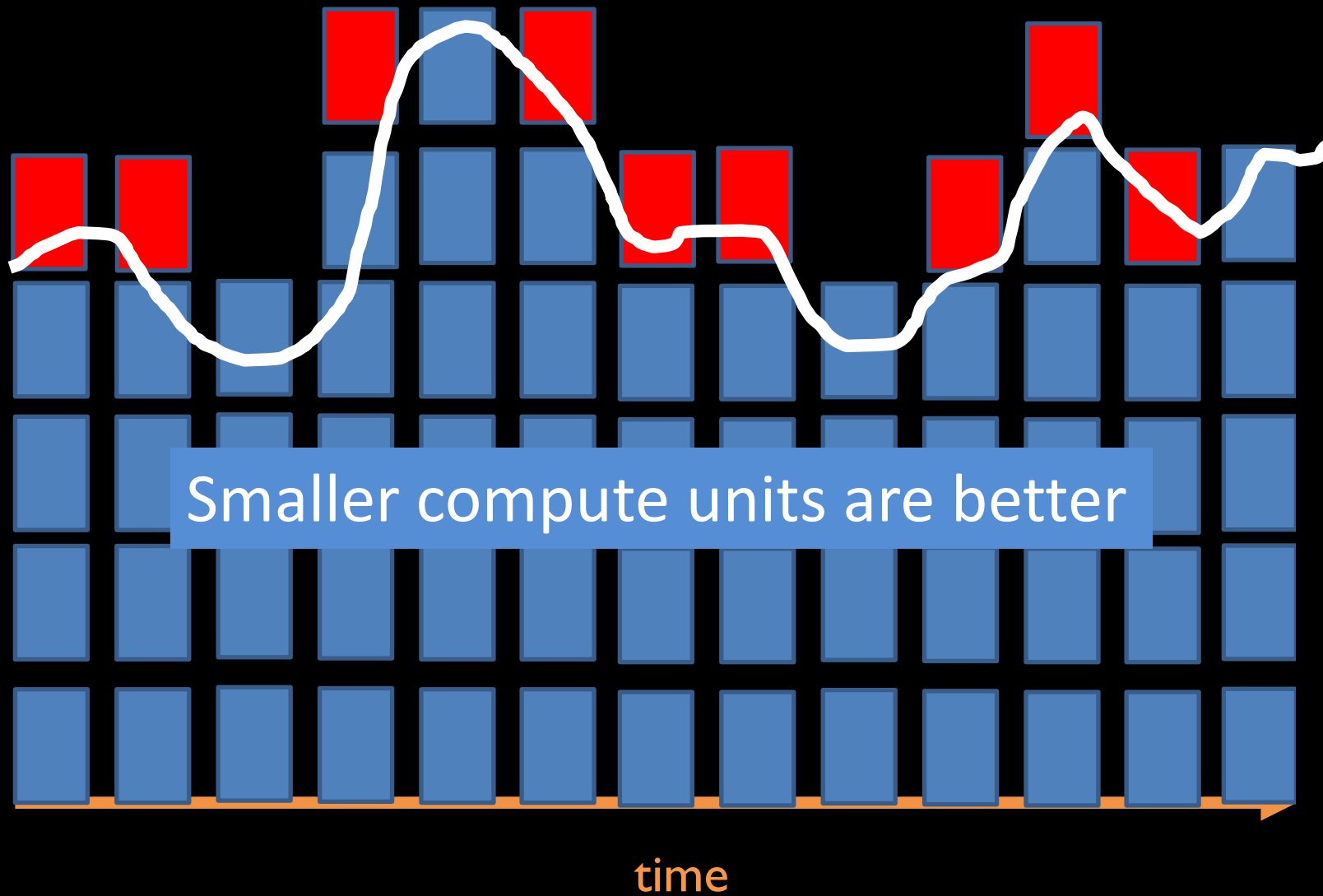
<https://www.flickr.com/photos/rvoegtl/>



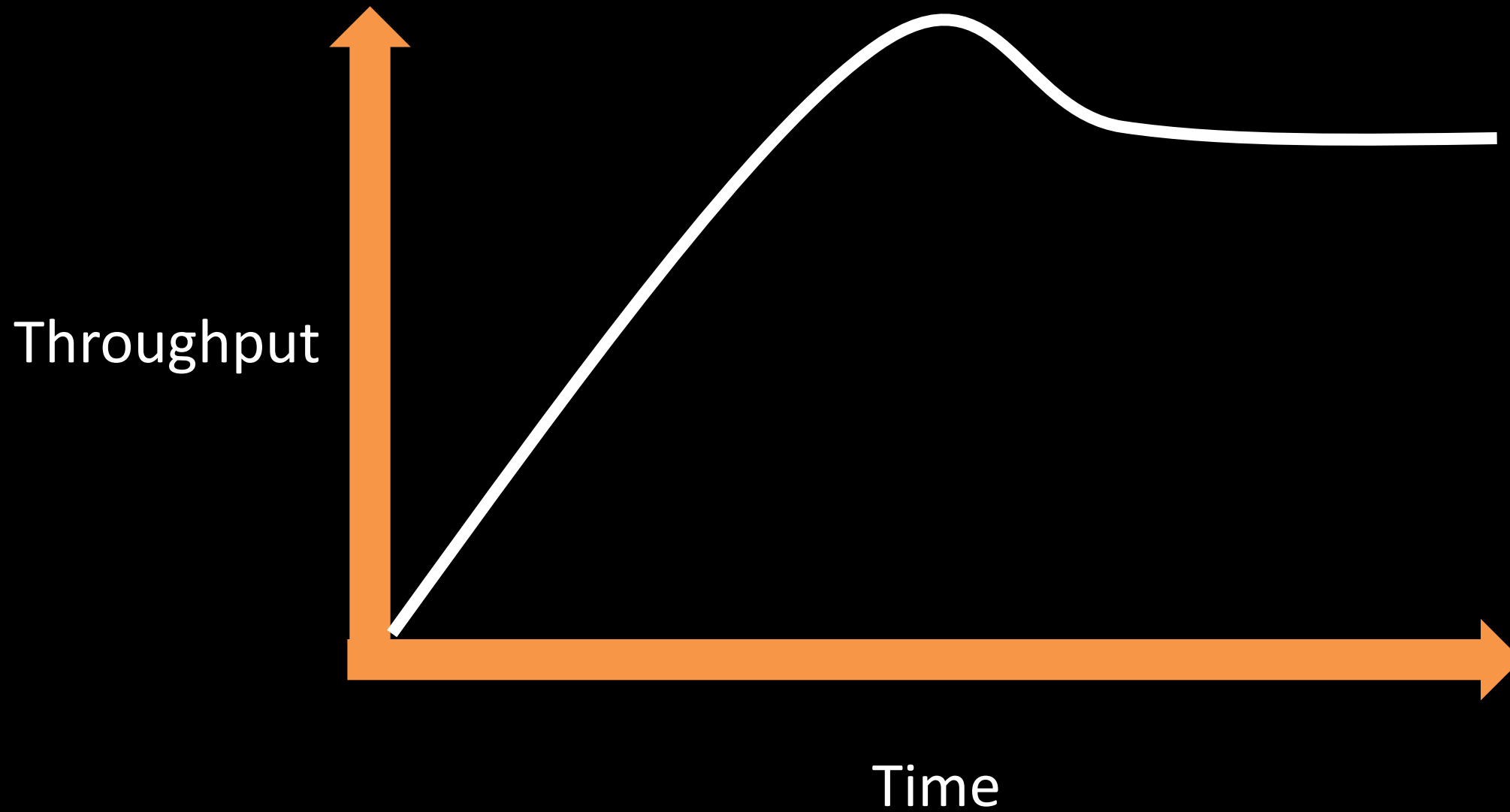
-

Let's put that differently

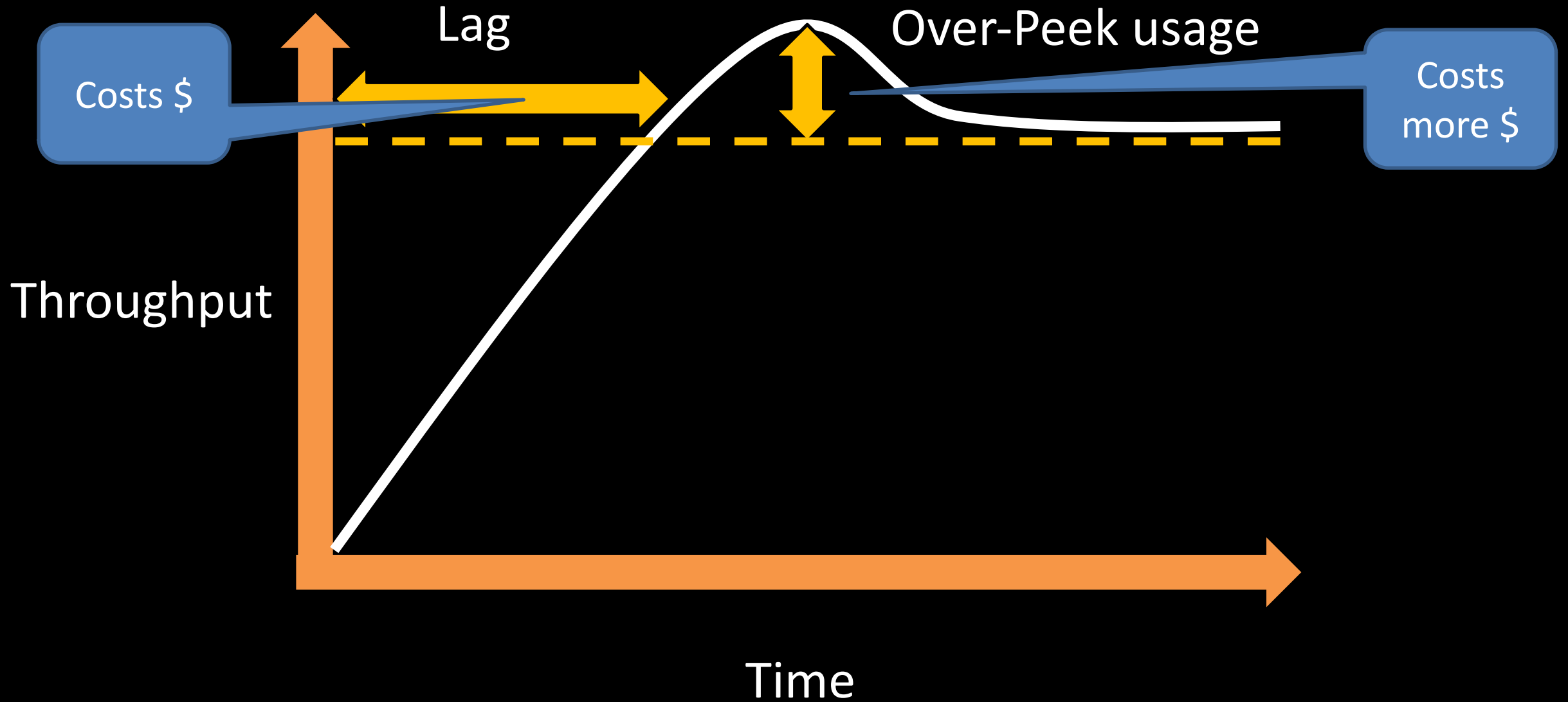
Demand



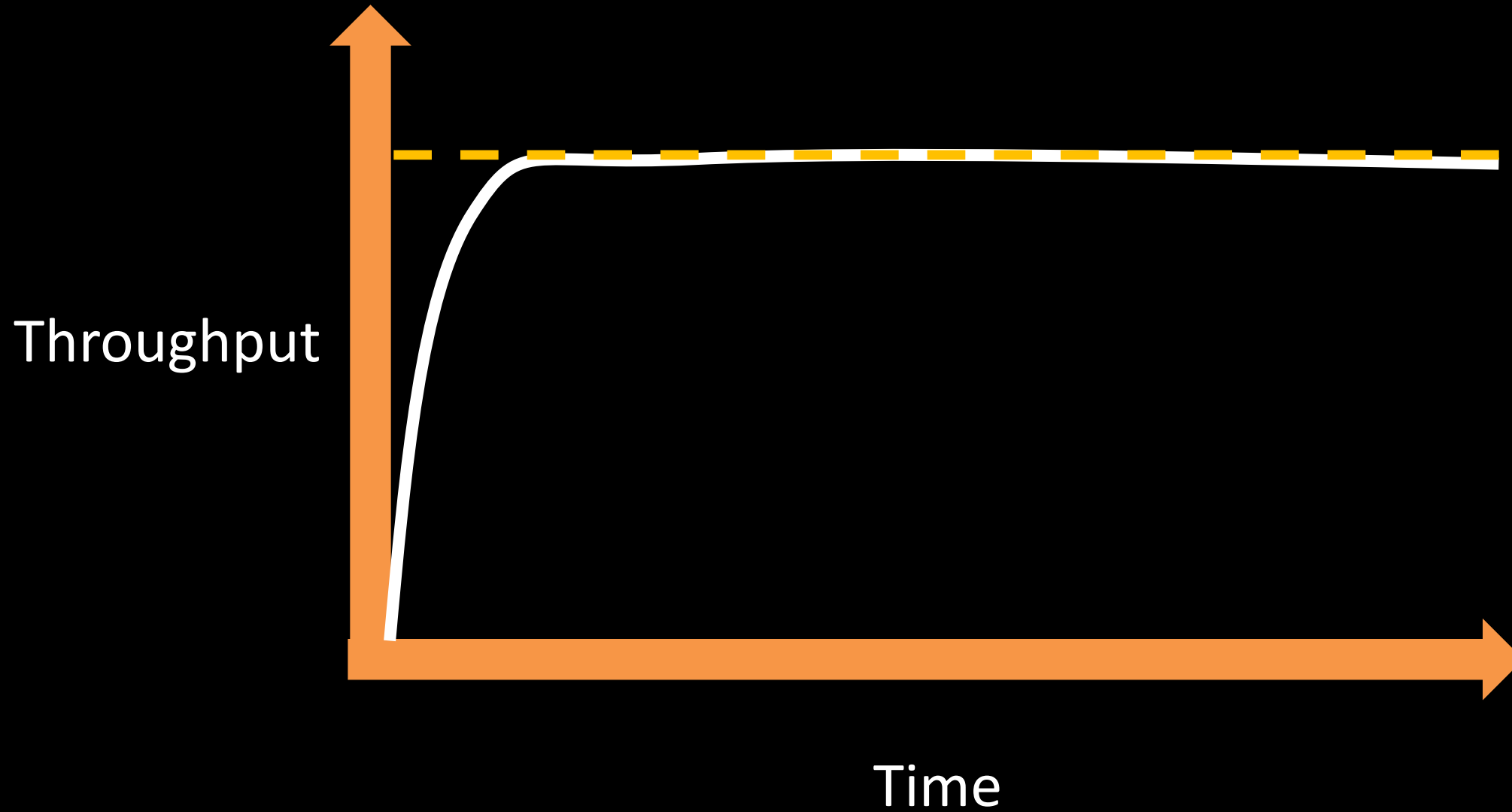
Traditional profile



Doesn't fit new model

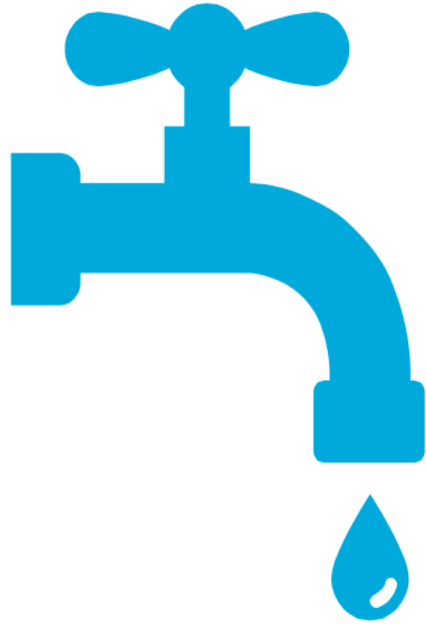


More like this please



What does this mean to Cloud Java developers?

- Changing `-Xmx` directly effects cost!
 - Very easy for businesses to understand
- Net effect: You'll be tuning your application to fit into specific RAM sizes
 - Smaller than you use today
- You need to understand where memory is being used.
 - You'll be picking components based on memory footprint
- Increased memory usage for 1 service increases the bill by the number of concurrent instances!



A faucet that drips just
once per second wastes

2,700

gallons of water annually.⁴

Someone
will be
looking at
your leaky
app





In the Cloud footprint is king

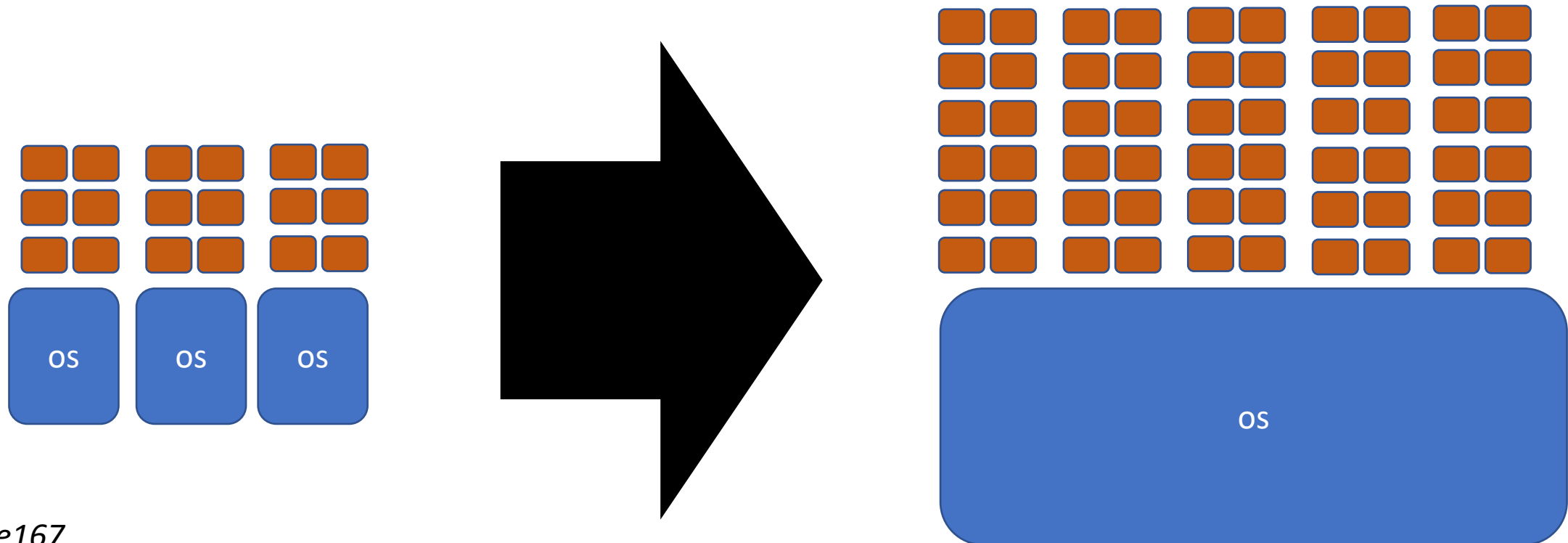
Application footprint is very important

Cloud users pay for resources

For Cloud providers: higher app density means lower operational costs

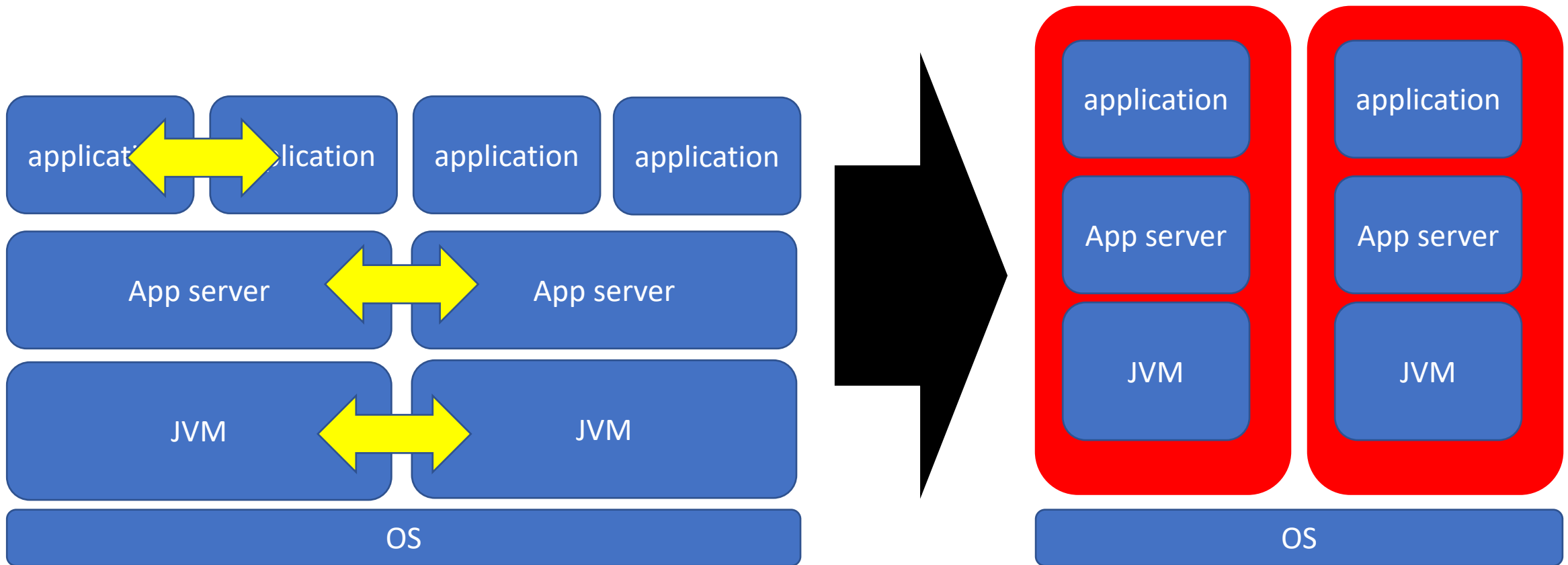
Providers : two opposing trends

1: Virtualization: more big machines partitioned into many smaller VMs
Opportunities to share across VMs increased



Providers : two opposing trends

2: Microservices / containers / serverless cause increased memory usage through less opportunities for sharing



Providers

1

need new ways of
reducing memory
usage

2

need to reduce tick-
over costs so that
CPU time only gets
used for running
applications

3

Need runtimes that
are ready to use as
quickly as possible

The challenge

- Hotspot needs to evolve in different ways
- Along paths that are difficult to take
- But, like the manatee, need to be followed to keep Java alive and vital





How does OpenJ9 Help?



Eclipse OpenJ9

Created Sept 2017

<http://www.eclipse.org/openj9>
<https://github.com/eclipse/openj9>

Dual License:
Eclipse Public License v2.0
Apache 2.0

Users and contributors very welcome

<https://github.com/eclipse/openj9/blob/master/CONTRIBUTING.md>

Java ME Inside!



Java ME requirements

- Small footprint
 - On disk and runtime.
 - Very limited RAM, usually more ROM
- Fast startup
 - Everybody wants their games to start quickly
- Quick / immediate rampup
 - Your game should not play better the longer you play



Java Cloud requirements

- Small footprint
 - Improves density for providers
 - Improves cost for applications
- Fast startup
 - Faster scaling for increased demand
- Quick / immediate rampup
 - GB/hr is key, if you run for less time you pay less money



**OpenJ9 may
may have its
roots in
small devices..**



IBM z14 at a glance

System, Processor, Memory

Five hardware models: M01, M02, M03, M04, M05

10 core 5.2GHz 14nm PU SCM

1 - 170 PUs configurable as CPs, zIIPs, IFLs, ICFs

Increased Uniprocessor capacity

Up to 33 sub capacity CPs at capacity settings 4, 5, or 6

CPC Drawers and backplane Oscillator

Enhanced SMT and new instructions for SIMD

Enhanced processor/cache design with 1.5x more on-chip cache sizes

Up to 32 TB DRAM, protected by Redundant Array of Independent Memory (RAIM)

Virtual Flash Memory (VFM)

192 GB HSA

Improved pipeline design and cache management



Announce: July 17, 2017

I/O Subsystem, Parallel Sysplex, STP, Security

PCIe Gen3 I/O fanouts with 16 GBps Buses

6 CSS, 4 Subchannel sets per CSS

0 - 5 PCIe I/O Drawer Gen3 (no I/O Drawer)

Next generation FICON Express16S+

10 GbE RoCE Express2

Integrated Coupling Adapter (ICA SR) and Coupling express LR for coupling links

Support for up to 256 coupling CHPIDs per CPC

CFCC Level 22

Crypto Express6S and CMPSC compression and Huffman Coding compression

STP configuration and usability enhancements (GUI)

IBM zHyperLink Express

OSA-Express6S

Secure Service Container

RAS, simplification and others

L3 Cache Symbol ECC

Acoustic and thin covers (space saving)

But it runs just as well on the largest

Largesum TCP/IP hardware Checksum (OSA-Express6S)

Pause-less garbage collection

Universal Spare SCM s (CP and SC)

Simplified and enhanced functionality for STP configuration

Enhanced Dynamic Memory Relocation for EDA and CDR

Virtual Flash Memory (replaces IBM zFlash Express)

IBM Dynamic Partition Manager updates

Up to 85 LPARs

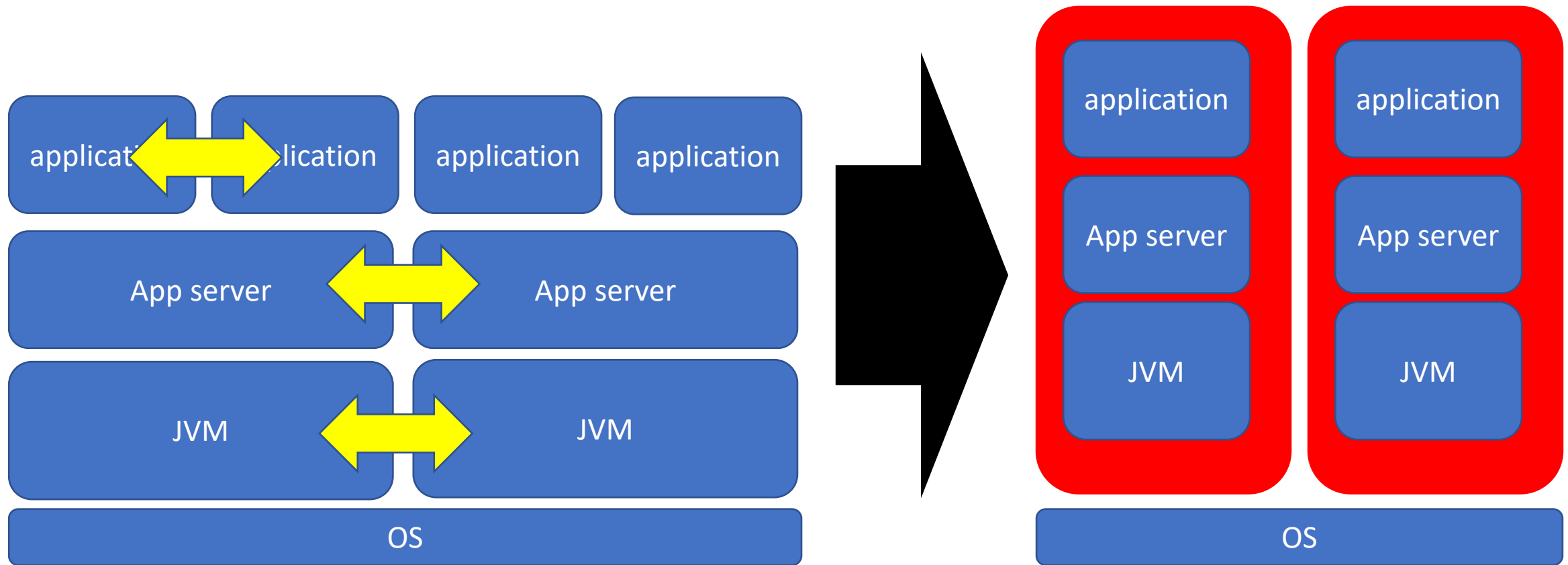
16 TB Memory per partition

Key elements

- Designed for scaling from the smallest to the largest
- Comes with several custom garbage collectors (even a soft-real time one)
- Has a class sharing approach that allows sharing of state and constant data in

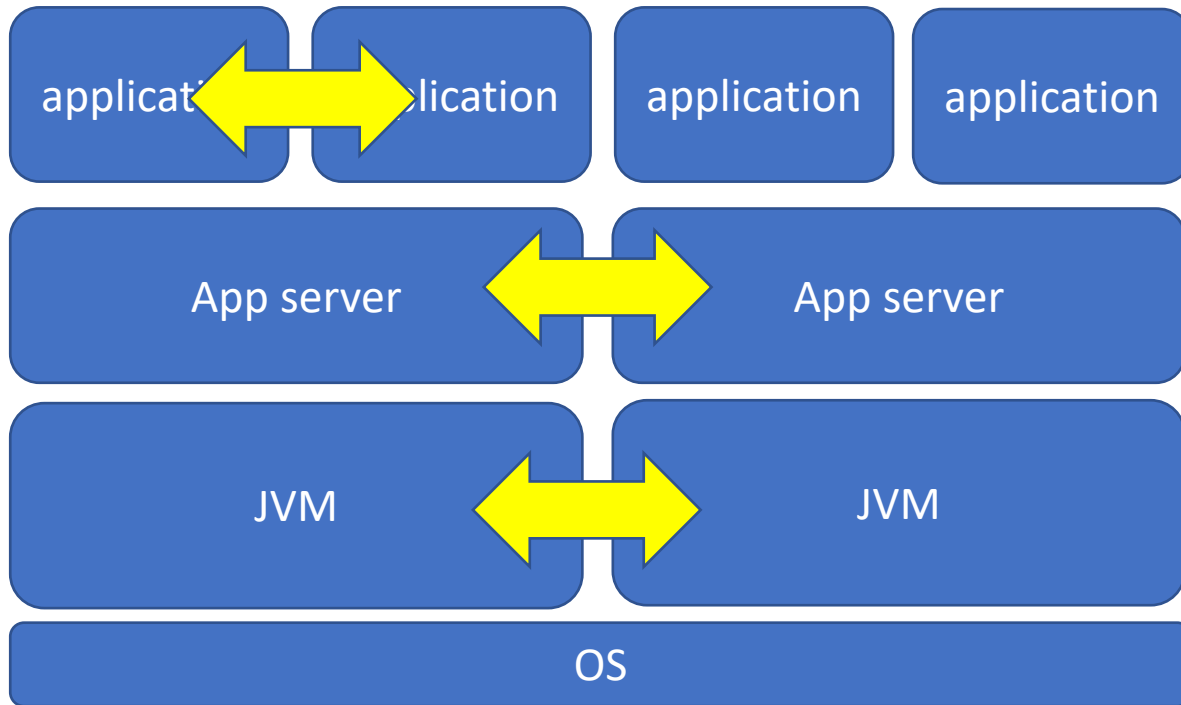


OpenJ9 Shared Classes can work at all levels

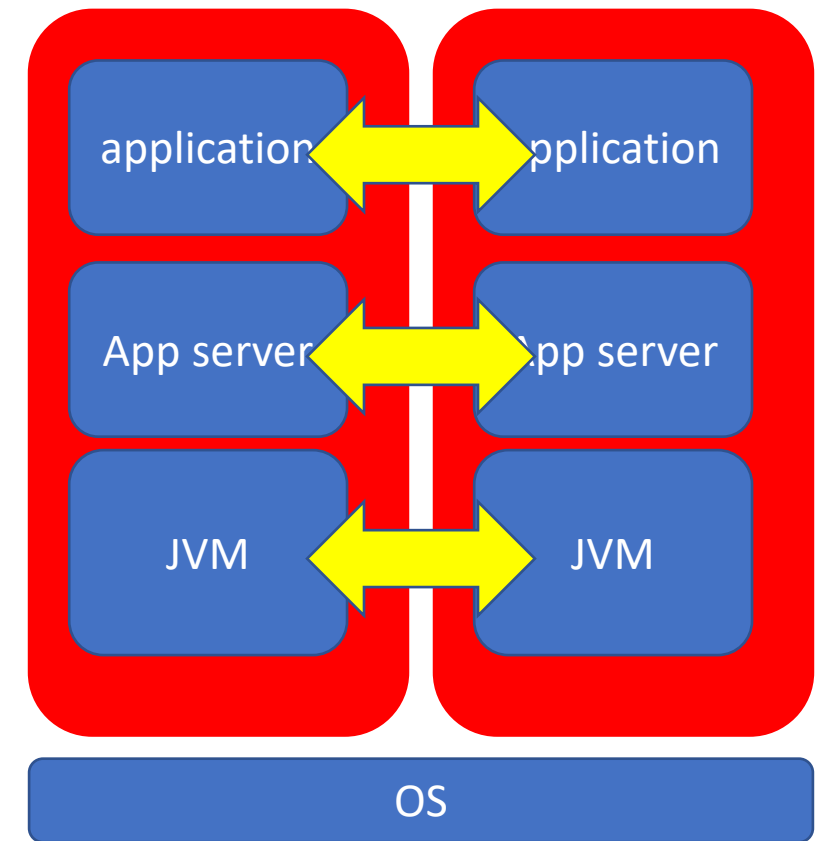


OpenJ9 Shared Classes can work at all levels

Can help you with traditional workloads



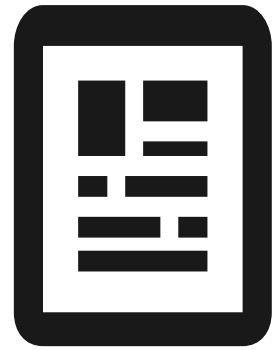
And with new containers etc



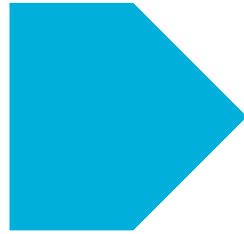
Shared Classes cache

- Xshareclasses
 - enables the share classes cache
- Xscmx50M
 - sets size of the cache

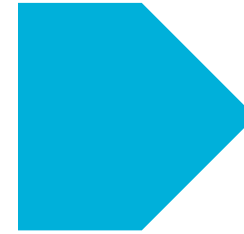
ShareClasses cache



Classfile

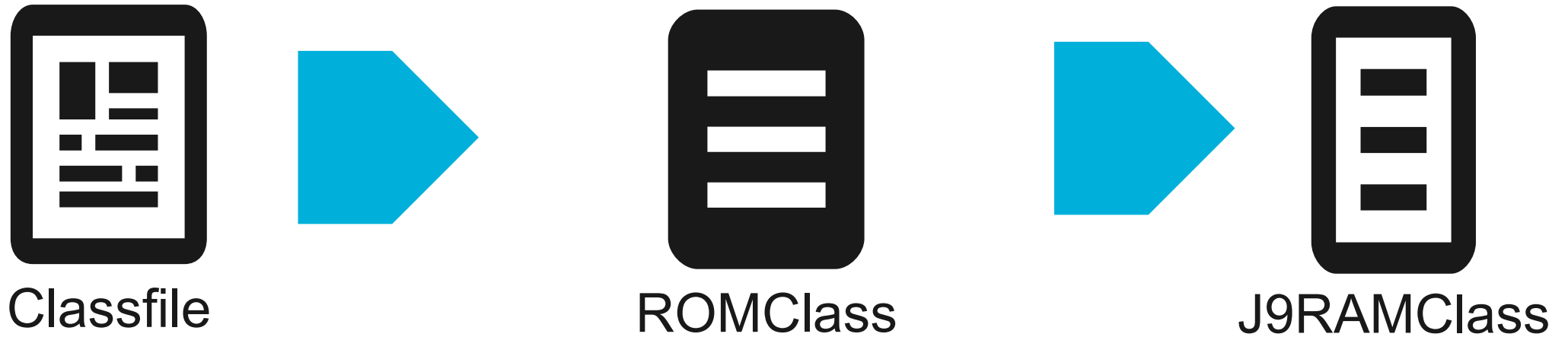


ROMClass



J9RAMClass

ShareClasses cache



This isn't the greatest format for running, compiling
or even caching

ShareClasses cache



So when loading J9 splits it into two parts

The read only part
Position independent

The stateful part

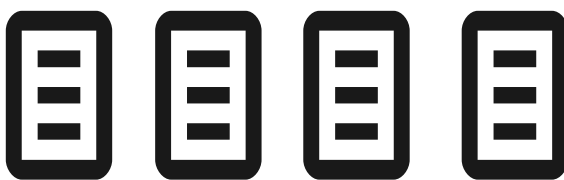
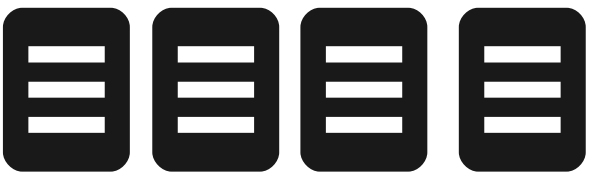
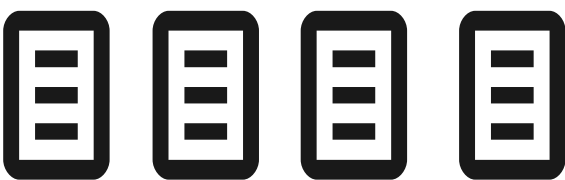
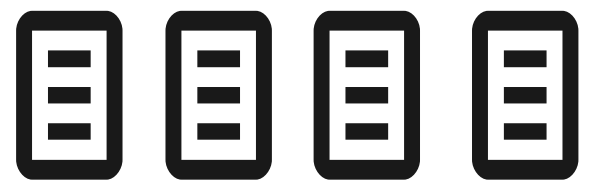
ShareClasses: ROM pays off

JVM 1

JVM 2

JVM 3

Three JVMs running the same code – on the same machine



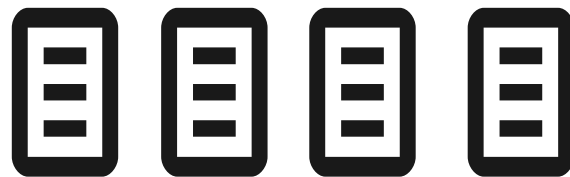
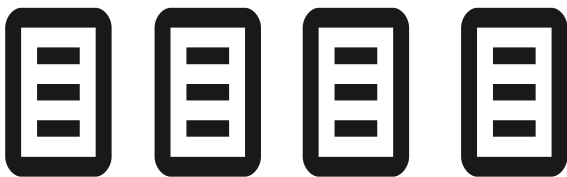
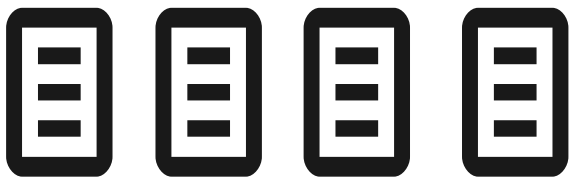
ShareClasses: ROM pays off

JVM 1

JVM 2

JVM 3

All the ROM classes are shared – position independent, non stateful



ShareClasses: ROM pays off

JVM 1

JVM 2

JVM 3

Giving faster startup, smaller footprint



Shared Classes
Cache



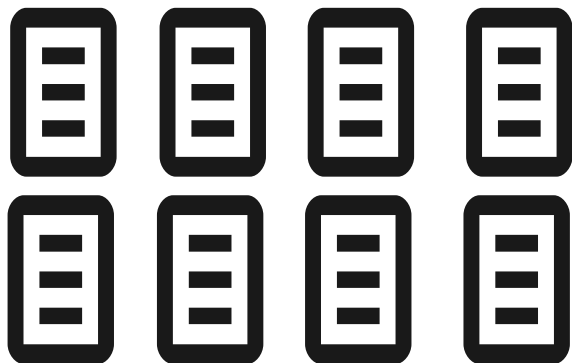
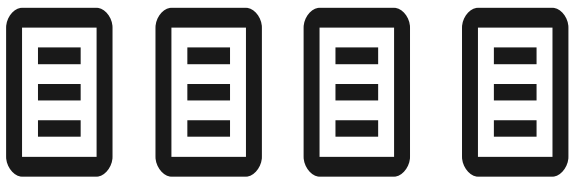
ShareClasses: ROM pays off

JVM 1

JVM 2

JVM 3

And J9 can share the rom classes across any boundary – VM or Container



Shared Classes
Cache



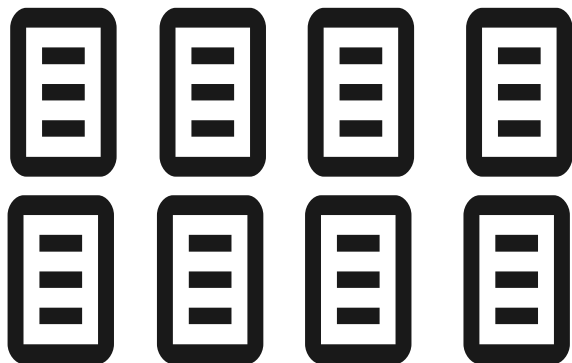
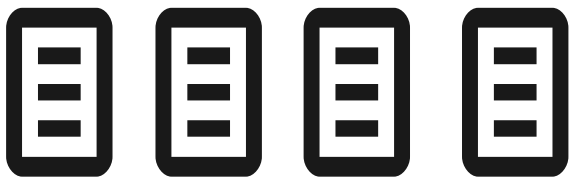
ShareClasses: ROM pays off

JVM 1

JVM 2

JVM 3

Sharing readonly data this way improves startup and footprint
Up to 20% footprint just by enabling shared classes



Shared Classes
Cache



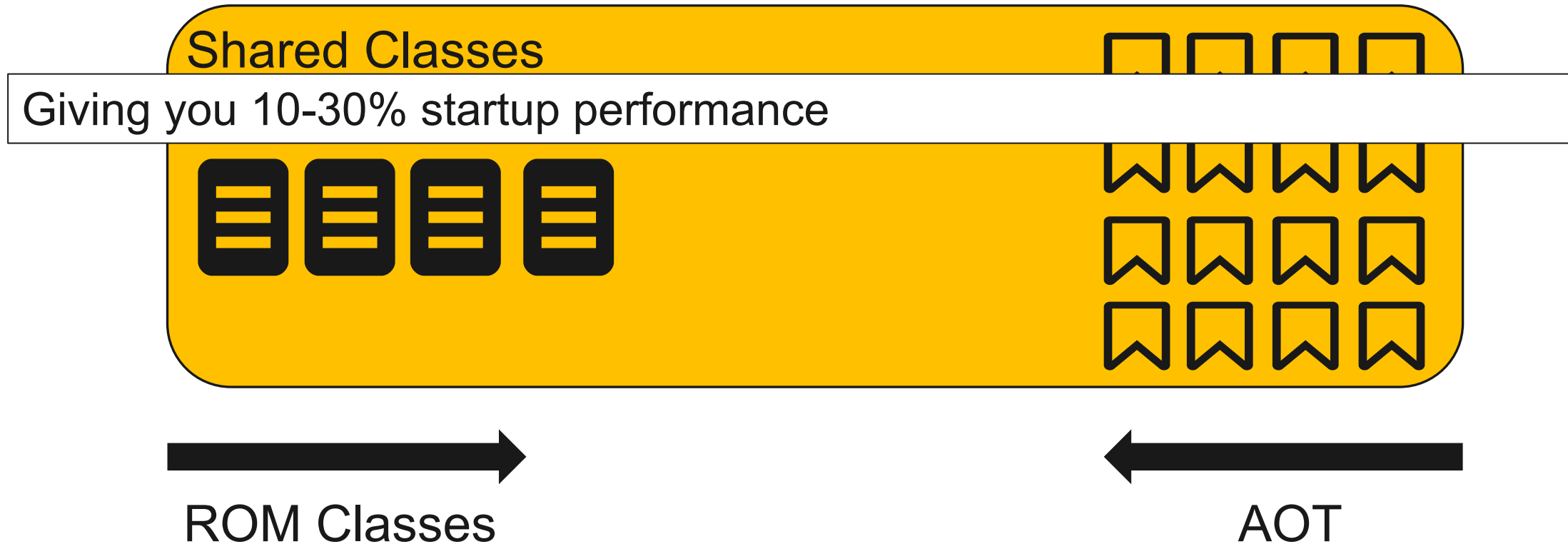
“And J9 can share JITed code too
“Dynamic” AOT through ShareClasses



```
$ java -Xshareclasses ...
```

“And J9 can share JITed code too

“Dynamic” AOT through ShareClasses



```
$ java -Xshareclasses ...
```

ShareClasses and AOT

- Distinction between 'cold' and 'warm' runs
- Dynamic AOT compilation
 - Relocatable format
 - AOT loads are ~100 times faster than JIT compilations
 - More generic code → slightly less optimized
 - Generate AOT code only during start-up
 - Recompilation helps bridge the gap

More tuning options

- -Xquickstart
 - Designed for the fastest start-up
 - Ideal for short-lived tasks
 - May limit peak throughput
- -Xtune:virtualized
 - Tuning for containers
 - Enables VM idle management
 - Improves start-up and ramp-up. Trade-off of small throughput loss

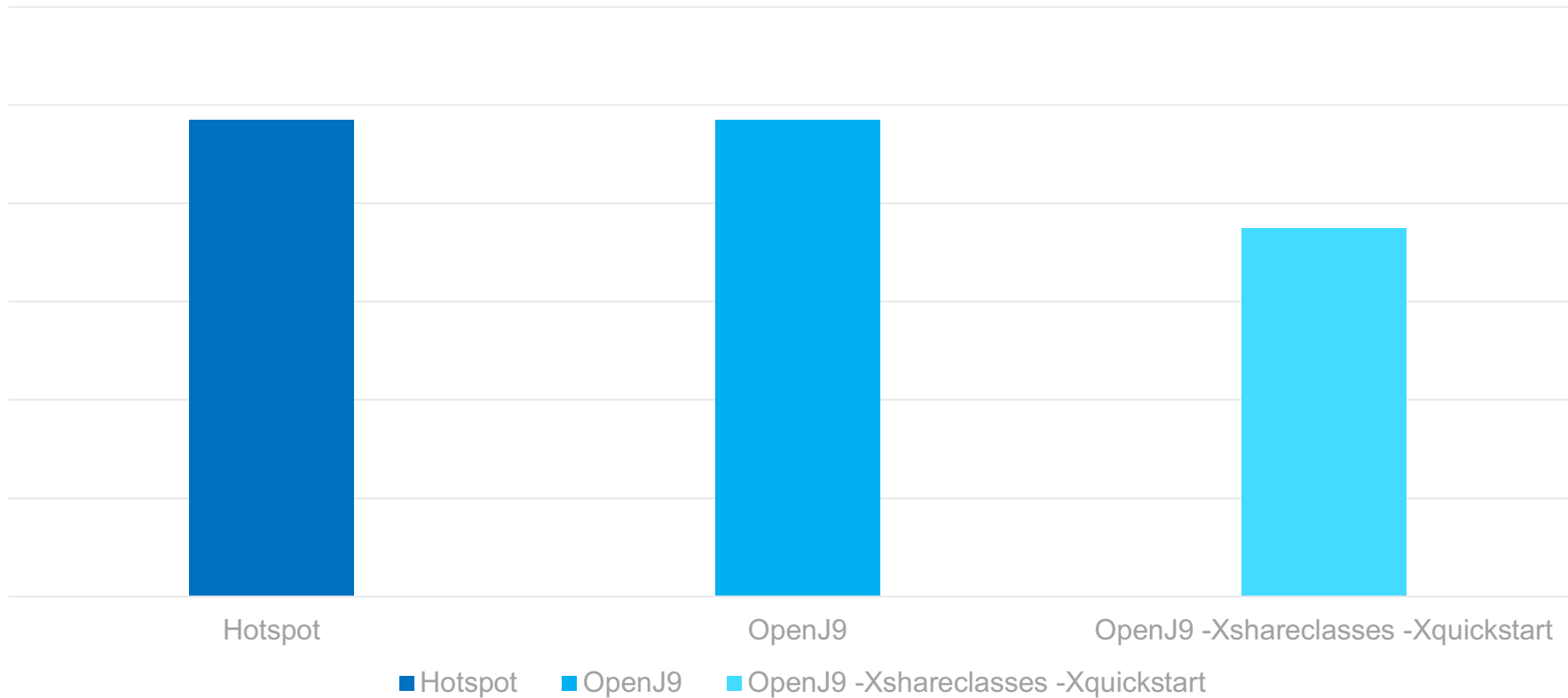


Want to see
what that all
means for Java?



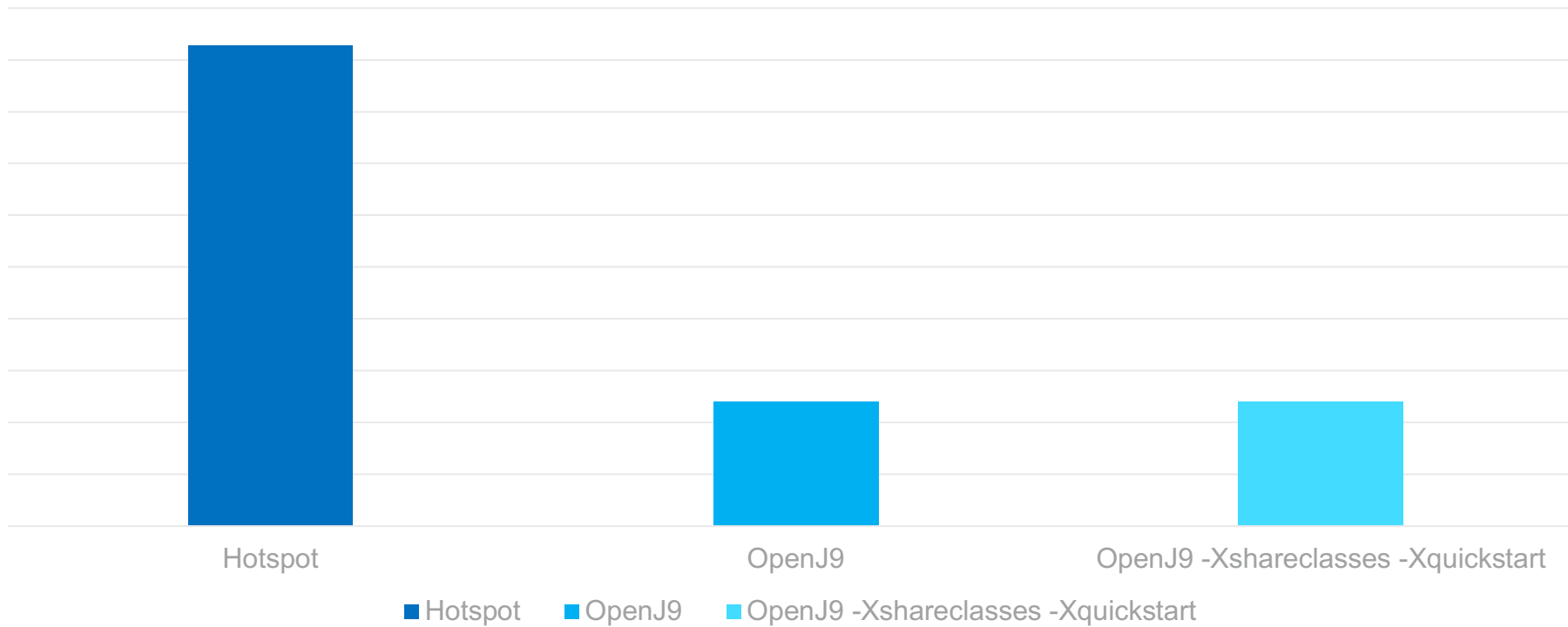
Results

Startup time is 30% faster with OpenJ9 -Xshareclasses -Xquickstart



Results

Footprint is 60% smaller with OpenJ9



Results

OpenJ9 triggers ~55% fewer wakeups

- OpenJDK9 with HotSpot – 0.168% CPU

▪ Summary: 84.7 wakeups/second, 0.0 GPU ops/seconds, 0.0 VFS ops/sec and 0.3% CPU use.

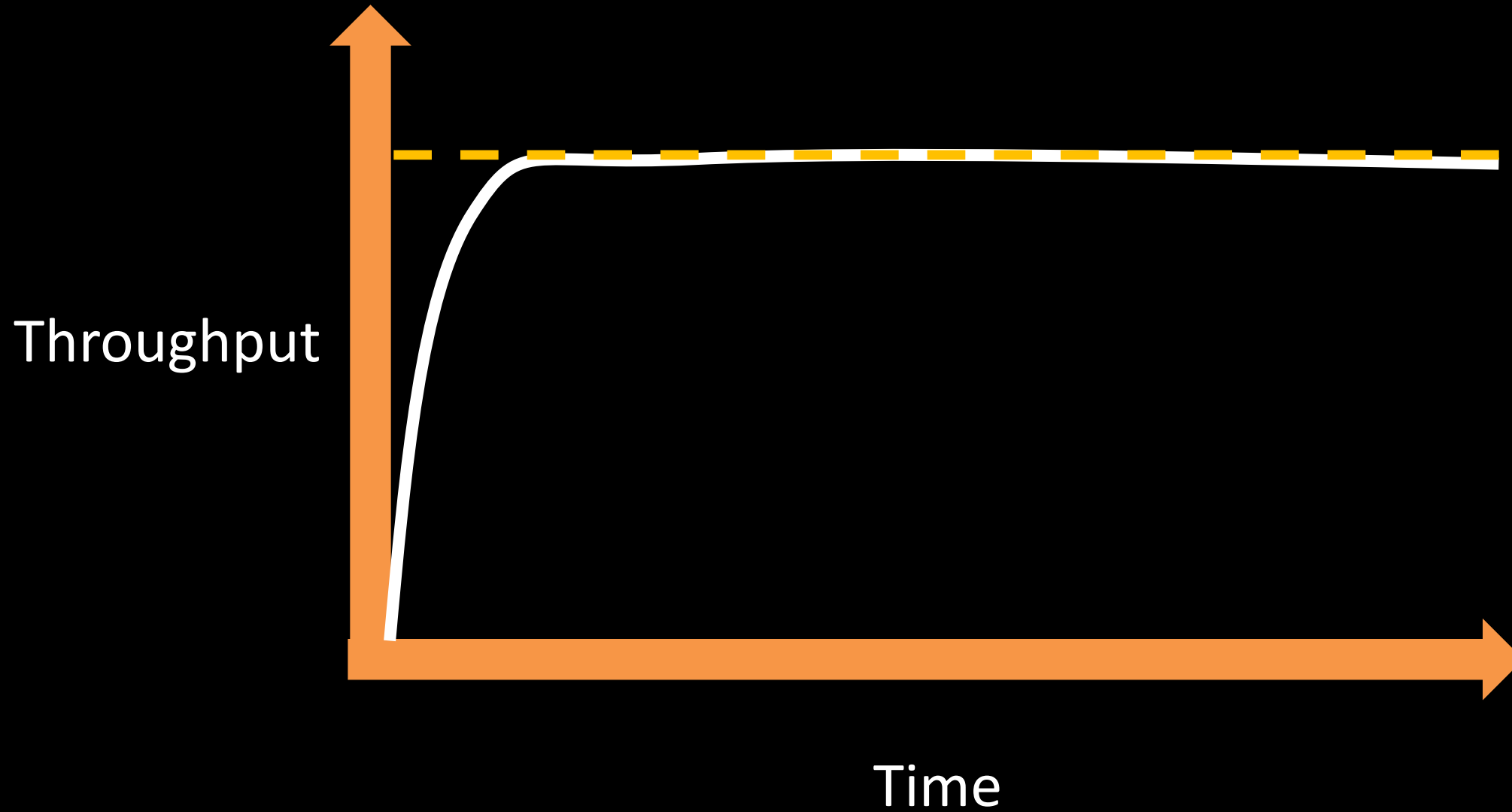
Usage	Events/s	Category	Description
0.9 ms/s	44.2	Process	/sdks/OpenJDK9-x64_Linux_20172509/jdk-9+181/bin/java
119.5 µs/s	20.0	Process	[xfsaild/dm-1]
138.6 µs/s	7.4	Timer	tick_sched_timer
10.5 µs/s	1.6	Process	[rcu_sched]
190.4 µs/s	1.5	Timer	hrtimer_wakeup
- OpenJDK9 with OpenJ9 – 0.111% CPU

▪ Summary: 38.5 wakeups/second, 0.1 GPU ops/seconds, 0.0 VFS ops/sec and 0.2% CPU use

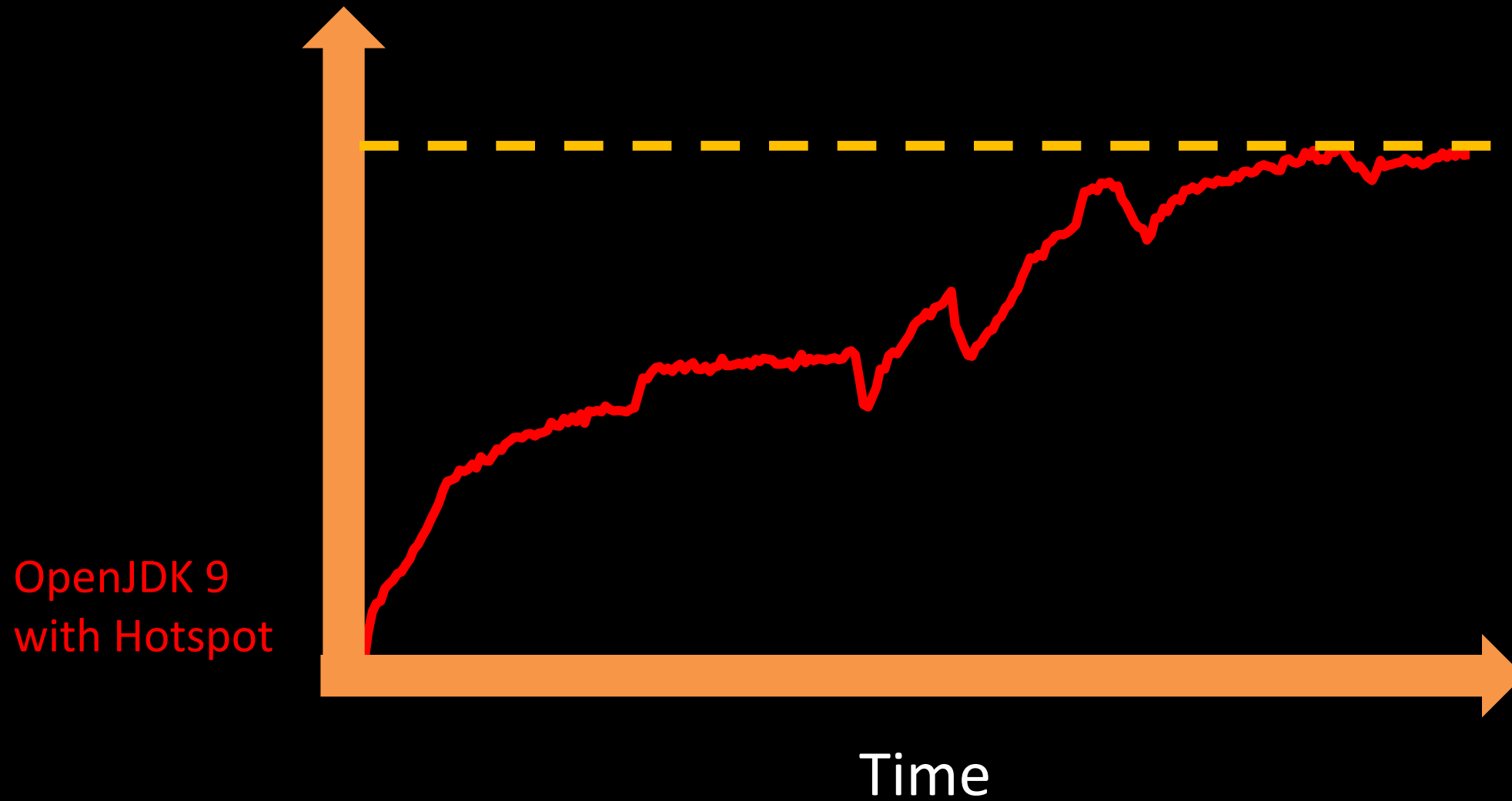
Usage	Events/s	Category	Description
681.2 µs/s	19.2	Process	/sdks/OpenJDK9-OPENJ9_x64_Linux_20172509/jdk-9+181/bin/java
58.3 µs/s	5.2	Timer	tick_sched_timer
21.9 µs/s	3.6	Process	[rcu_sched]
39.3 µs/s	2.0	Timer	hrtimer_wakeup
157.1 µs/s	1.0	kWork	ixgbe_service_task



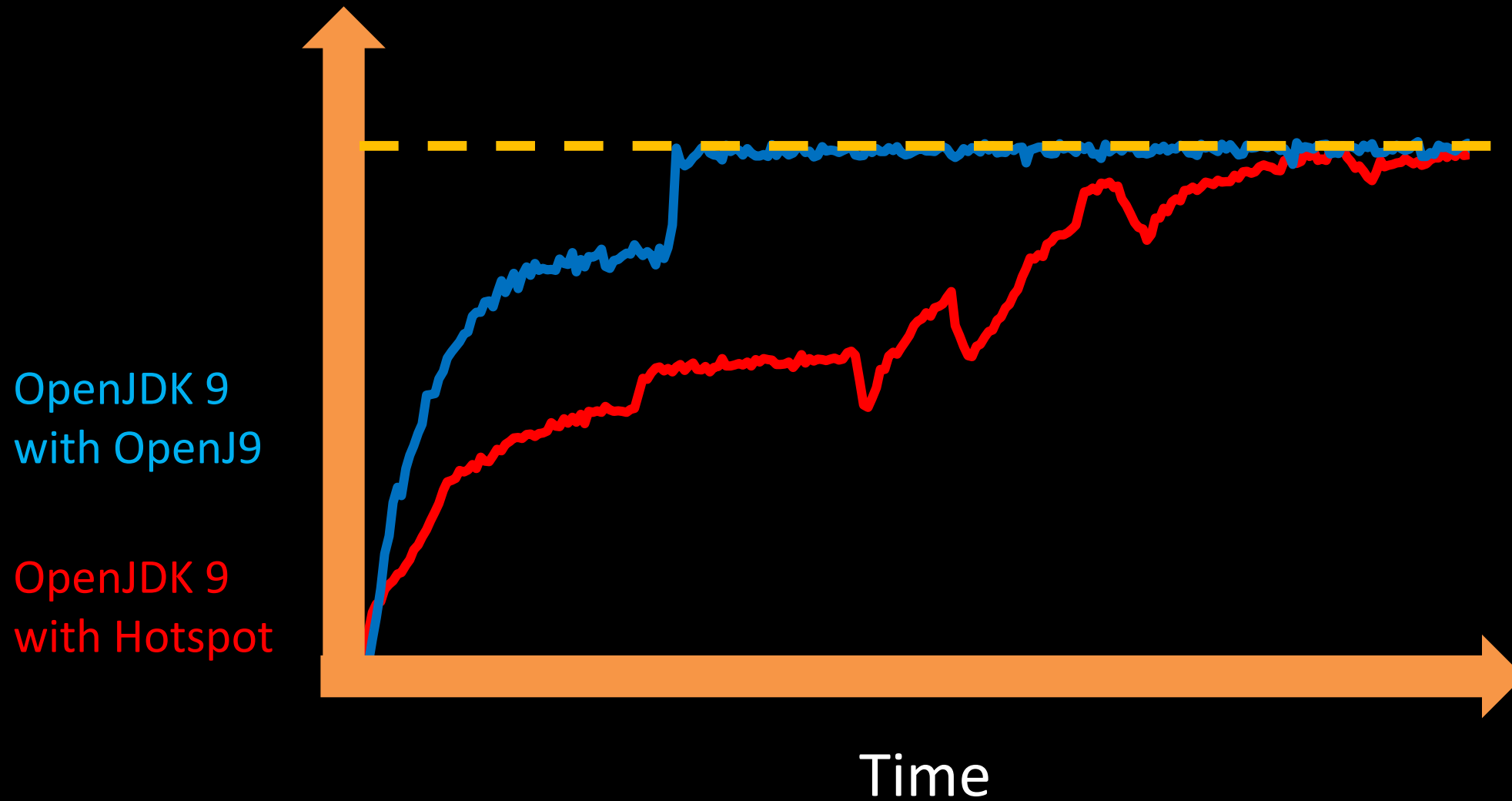
More like this please



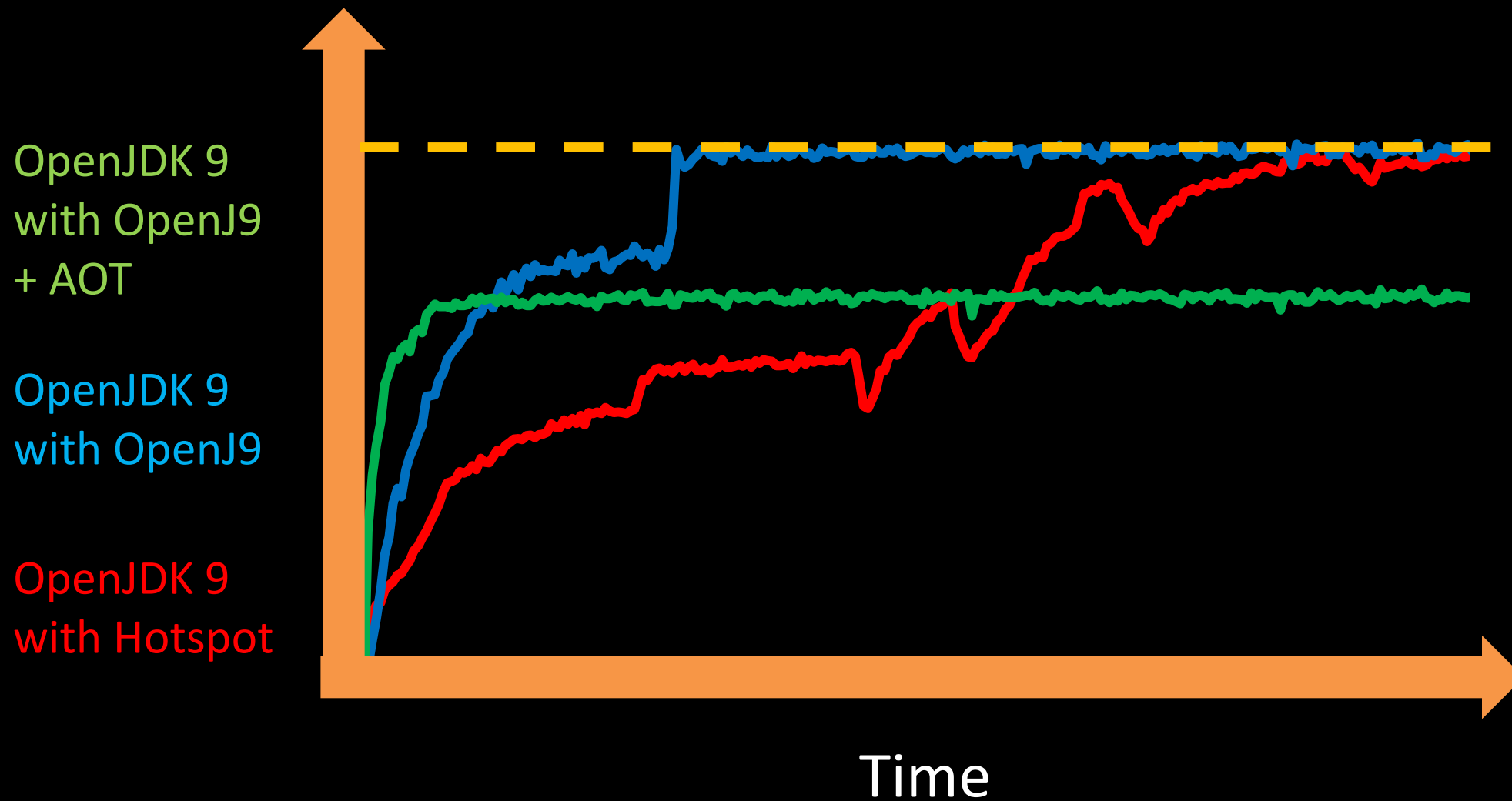
Real data



Real data



Real data



Open J9

- IBM has been developing J9 since Java began
- J9 was born into a restless world
- not static like the manatee , not focused on speed like the dolphin or stamina like the mustang.
- This jvm needed to be able to do everything – be all these creatures.



Open J9

Designed from the start to span all the operating systems needed by IBM products

This JVM can go from small to large

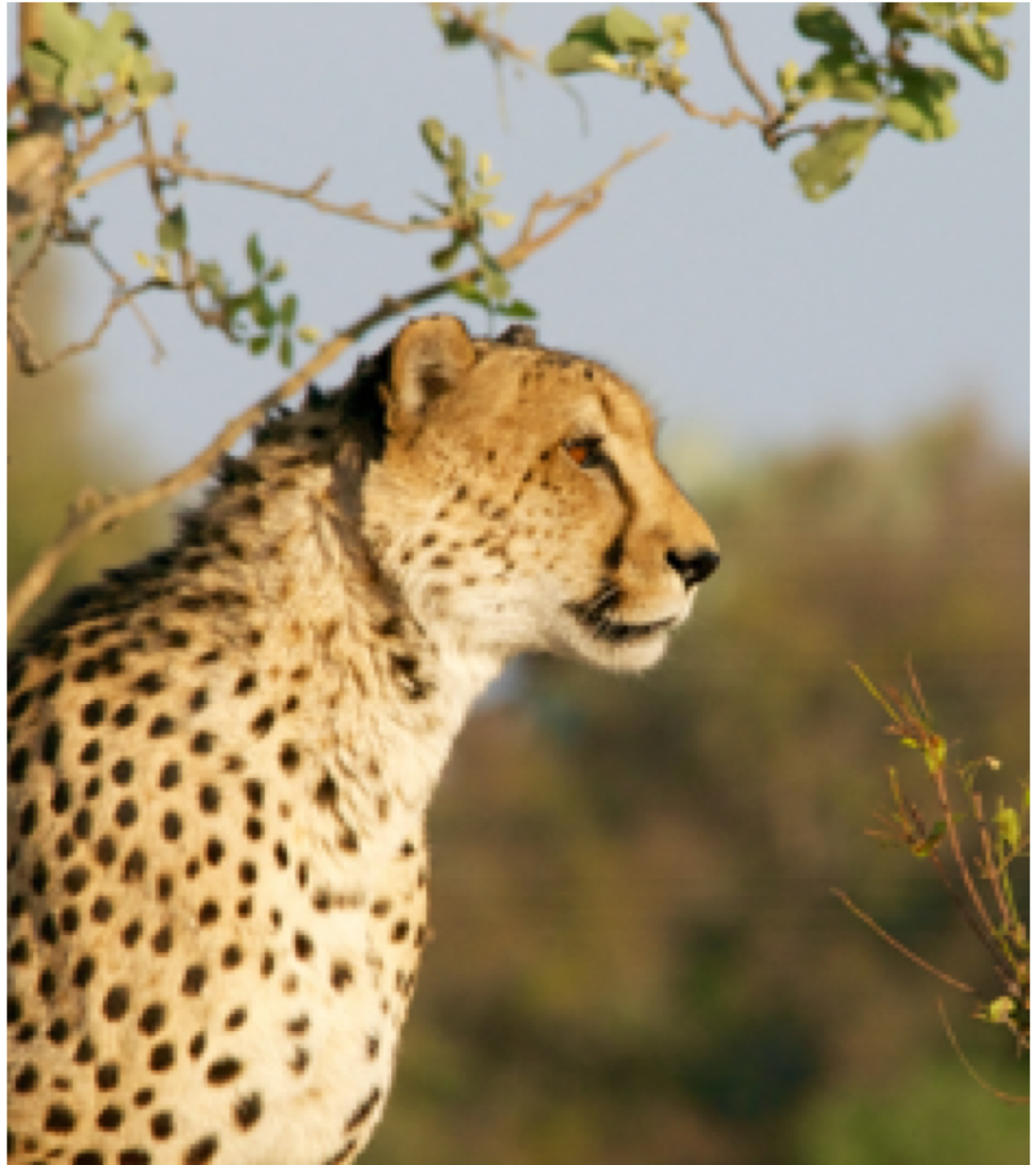
Can handle constrained environments or memory rich ones

Is used by the largest enterprises on the planet

If any JVM can be said to be at the heart of the enterprise – its this one.

Open J9 is a
new animal

@spoole167





Using OpenJ9 is easy

Prebuilt OpenJDK Binaries

Java™ is the world's leading programming language and platform. The code for Java is [open source](#) and available at [OpenJDK™](#). AdoptOpenJDK provides prebuilt OpenJDK binaries from a fully open source set of [build scripts](#) and infrastructure. Looking for docker images? Pull them from [our repository on dockerhub](#)

Downloads

OpenJDK 8 with Eclipse OpenJ9 ▼

Latest build ➔

jdk8u152-b16

Archive 📁

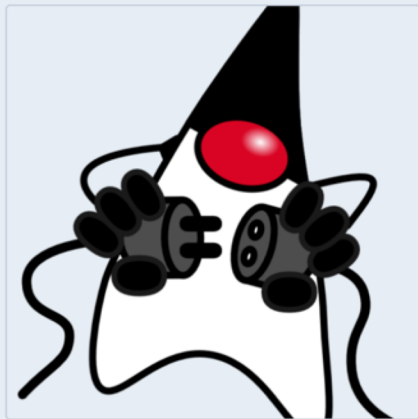
Installation ➔

Get involved ➔

[Blog](#) | [Support](#) | [Sponsors](#) | [About](#) | [API](#)



Repos



adoptopenjdk
AdoptOpenJDK
Community
Project

<https://adoptopenjdk.net>

Joined June 2017



[adoptopenjdk/openjdk8](#)
public

2
STARS

5.3K
PULLS

>
DETAILS



[adoptopenjdk/openjdk8-openj9](#)
public

2
STARS

4.9K
PULLS

>
DETAILS



[adoptopenjdk/openjdk9](#)
public

1
STARS

4.1K
PULLS

>
DETAILS



[adoptopenjdk/openjdk9-openj9](#)
public

5
STARS

3.7K
PULLS

>
DETAILS



[adoptopenjdk/openjdk10](#)
public

2
STARS

1.9K
PULLS

>
DETAILS



[adoptopenjdk/openjdk10-openj9](#)
public

1
STARS

11
PULLS

>
DETAILS



IBM donated J9 to Eclipse
because we believe it's the best
way to move Java forward

- It offers a new place to start
- As the future emerges we can see that Java needs to handle new technologies, new hardware.
- Whether GPUs or Neuromorphic Processors or even ultimate prize of Quantum computers: Java must adapt.
- We can't do it on our own. We have to do it together



Having a choice allows
different roads to be
followed

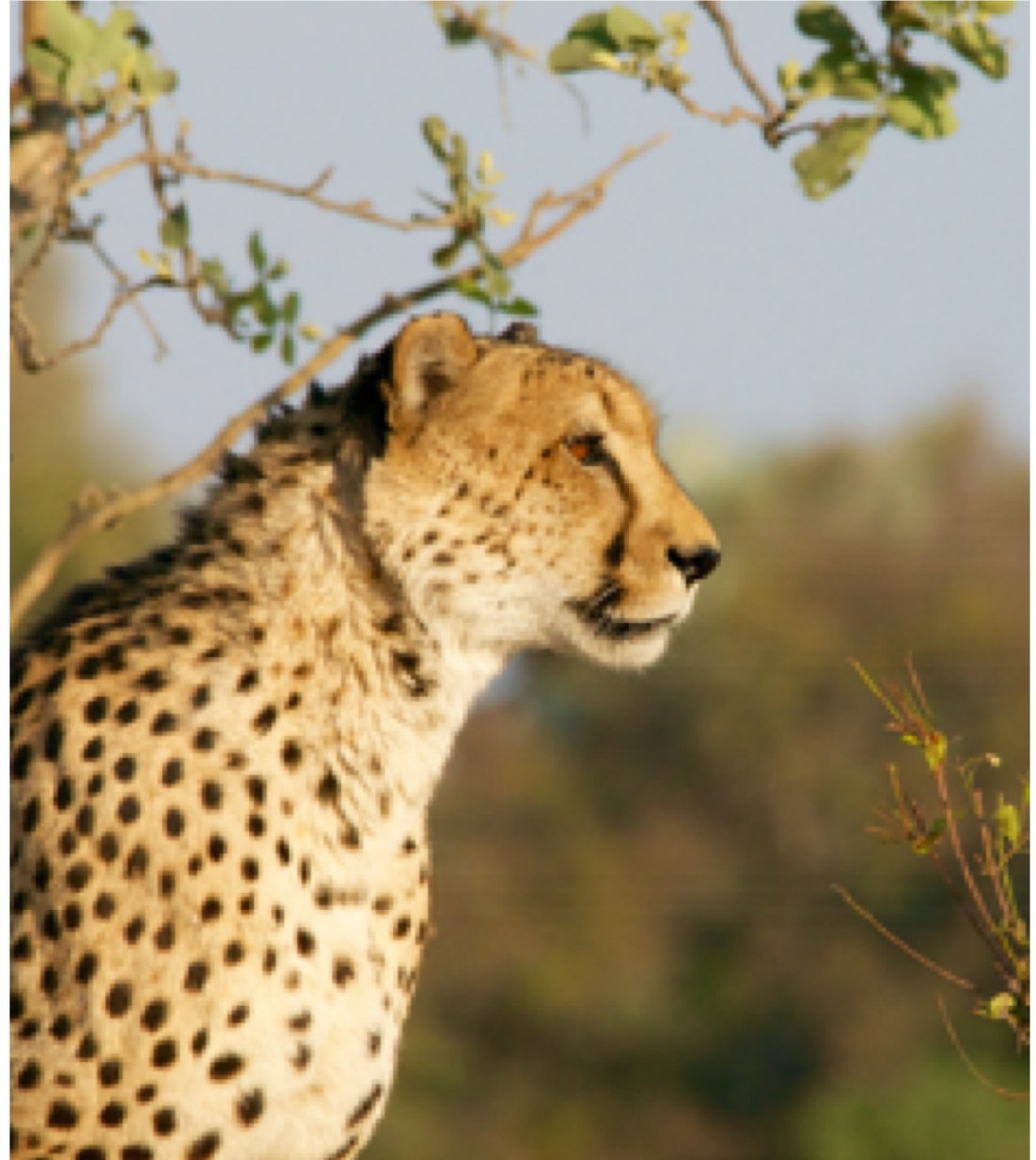
- Different places
found
- Different friends
made



Open J9: a different beast



@spoole167



adoptopenjdk.net

Don't just
take my word
for it

OpenJ9

@spoole167



Eclipse @openj9 is “mindblowingly good”
good for Docker and microservice use cases
with Java. Check it out!

Follow



Roy van Rijn
@royvanrijn

Oh wow, switching from OpenJDK 9 to
#Eclipse OpenJ9 is impressive, everything
worked out of the box (just two changes in
my Dockerfile) and memory usage went from
300MiB to 130MiB, the Spring Boot service
is now almost 'micro' 🥰👍

Follow

Java and Docker, the limitations
Mismatch in virtualizationThe combination of Java and Docker
isn't a match made in heaven, initially it was far from it. For
th...

adoptopenjdk.net

Thank you



@spoole167

Prebuilt OpenJDK Binaries

Java™ is the world's leading programming language and platform. The code for Java is [open source](#) and available at [OpenJDK™](#). AdoptOpenJDK provides prebuilt OpenJDK binaries from a fully open source set of [build scripts](#) and infrastructure. Looking for docker images? Pull them from [our repository on dockerhub](#)

Downloads

OpenJDK 8 with Eclipse OpenJ9 ▼

Latest build ➞

jdk8u152-b16

Archive 📦

Installation ➞ Get involved ➞

[Blog](#) | [Support](#) | [Sponsors](#) | [About](#) | [API](#)

adoptopenjdk.net