# Java with Ultra Fast Startup and Remote JIT Compilation

A workshop on acelerating Java application startup and memory use
With IBM Semeru Runtimes and the Eclipse OpenJ9 JVM

Mark Stoodley
Chief Architect for Java at IBM and an Eclipse OpenJ9 project lead
2024-05-13

# A little about myself…

- Ph.D. University of Toronto
  - Studied computer architecture and compiler optimizations
- 22 years experience developing, testing, building, servicing, leading, architecting Java runtimes for IBM products and customers
- My current roles:
  - STSM, Chief Architect for Java at IBM
  - A project lead for Eclipse OpenJ9
  - A project lead for Eclipse OMR

# Java with Ultra Fast Startup and Remote JIT Compilation

Java starting too slowly for you? Amazed that your program has to JIT compile the same old methods every time you run it, consuming memory and CPU cycles that you've got to pay for? It doesn't have to be that way!

With Eclipse OpenJ9 built into the Semeru Runtimes JDK, you can start your Linux-based Java servers and then save them to disk. When you need another instance, don't start it from scratch! Simply restore it from disk to avoid almost all that startup work to get your server handling requests in just a couple hundred milliseconds!

And that powerful JIT compiler can be easily configured to run as a remote service so that your applications aren't disrupted by those extra CPU cycles and memory use. In this workshop, we'll teach you a bit more about these amazing technologies and then let you try them out yourselves with some simple Liberty-based servers ~~running on the OpenShift Container Platform~~. You'll see firsthand why these technologies are being used in production today to provide incredibly elastic and cost-effective servers for Java workloads.

You know you're curious! Bring your questions and an open mind and let us show you how you too can overcome Java's traditional overheads without restricting the use of any Java language features!

# Notices and disclaimers



- **This document is distributed "as is" without any warranty, either express or implied. In no event shall IBM be liable for any damage arising from the use of this information, including but not limited to, loss of data, business interruption, loss of profit or loss of opportunity.**
- Customer examples are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual performance, cost, savings or other results in other operating environments may vary.
- Workshops, sessions and associated materials may have been prepared by independent session speakers, and do not necessarily reflect the views of IBM.
- Not all offerings are available in every country in which IBM operates.
- 
- Any statements regarding IBM's future direction, intent or product plans are subject to change or withdrawal without notice.
- IBM, the IBM logo, and ibm.com are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at: www.ibm.com/legal/copytrade.shtml.

- Certain comments made in this presentation may be characterized as forward looking under the Private Securities Litigation Reform Act of 1995.
- Forward-looking statements are based on the company's current assumptions regarding future business and financial performance. Those statements by their nature address matters that are uncertain to different degrees and involve a number of factors that could cause actual results to differ materially. Additional information concerning these factors is contained in the Company's filings with the SEC.
- Copies are available from the SEC, from the IBM website, or from IBM Investor Relations.
- Any forward-looking statement made during this presentation speaks only as of the date on which it is made. The company assumes no obligation to update or revise any forward-looking statements except as required by law; these charts and the associated remarks and comments are integrally related and are intended to be presented and understood together.

# Workshop Outline

| | |
|---|---|
| 1300 | Refresher on Semeru Runtimes and Eclipse OpenJ9 |
| 1350 | Break for 10 minutes |
| 1400 | Introduction to the workshop materials and how to get going |
| 1415 | 1. The benefits of the shared classes cache |
| 1500 | Break for 10 minutes |
| 1510 | 2. Liberty & Semeru InstantOn: ultra fast startup |
| 1600 | Break for 10 minutes |
| 1610 | 3. Semeru Cloud Compiler: shrink your containers |
| 1655 | Wrap up! |
| 1700 | |

# Before we begin

- The workshop materials assume you have git and podman/docker installed on your machine
  - Created and primarily tested on my Macbook Pro using podman
  - Tested some stuff using podman with Windows WSL and Ubuntu 22+
  - Should work really well if you're running Linux

- Please start now if you want to do the workshop yourself

```
$ git clone https://github.com/mstoodle/jcon2024_workshop
$ cd jcon2024_workshop
# See README.md which has some links to install podman

$ ./main.build.sh # builds a container called workshop/main
```

# Java on Cloud

# The Good

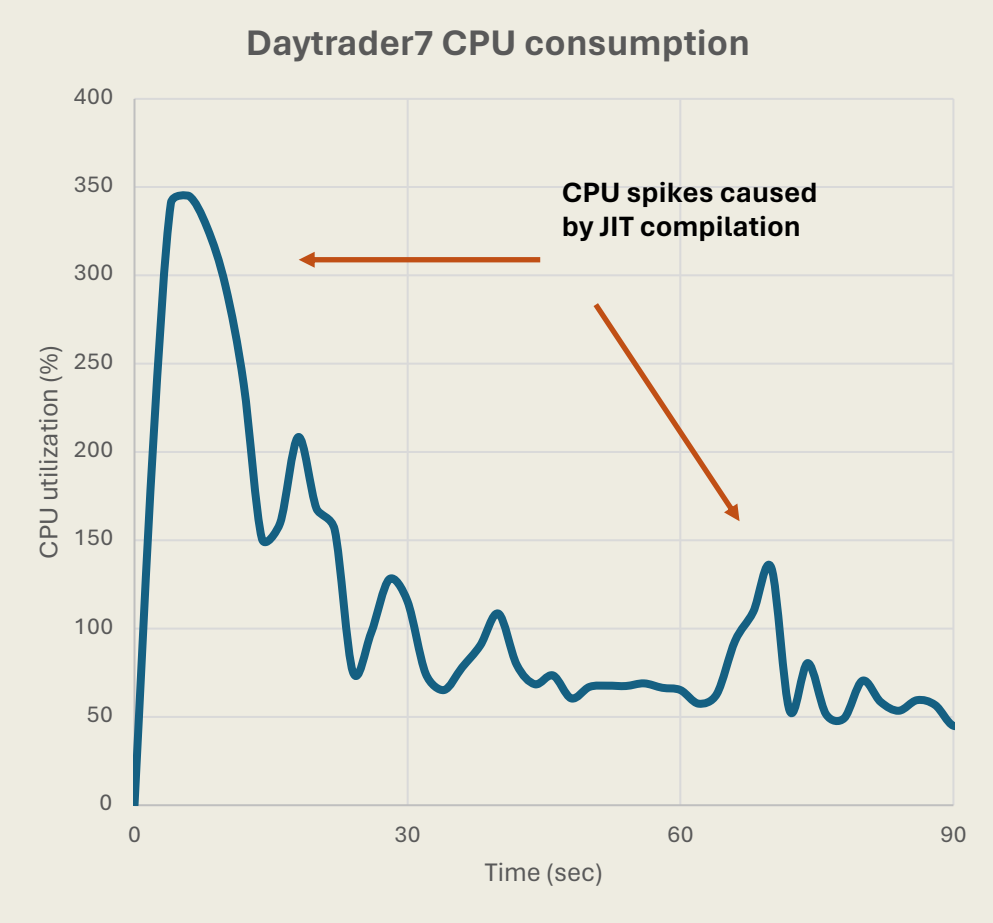
- Platform independent – write once, run anywhere
- > 25 years of improvements
- JIT produces efficient optimized machine code
- Efficient garbage collection
- Longer it runs, the better it runs (JVM collects more profile data, JIT compiles more methods)

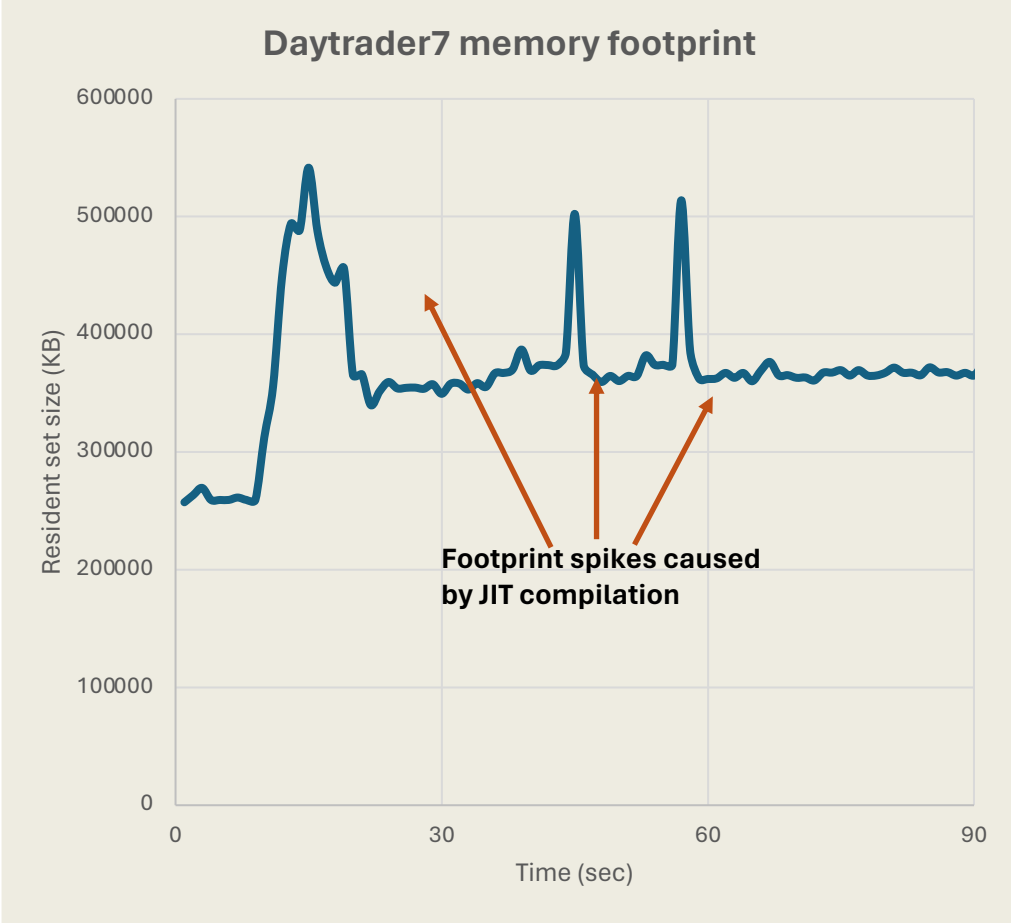
# The Not-So-Good

- Initial execution run is “interpreted”, which is relatively slow
- JIT compiling “Hot” methods brings CPU and memory spikes
  - Especially when the JVM is warming up

- CPU spikes can lower QoS
- Memory spikes can produce OOM issues, including crashes
- Slow start-up time (time to load and ready to accept requests)
- Slow ramp-up time (time to reach stead-state performance)

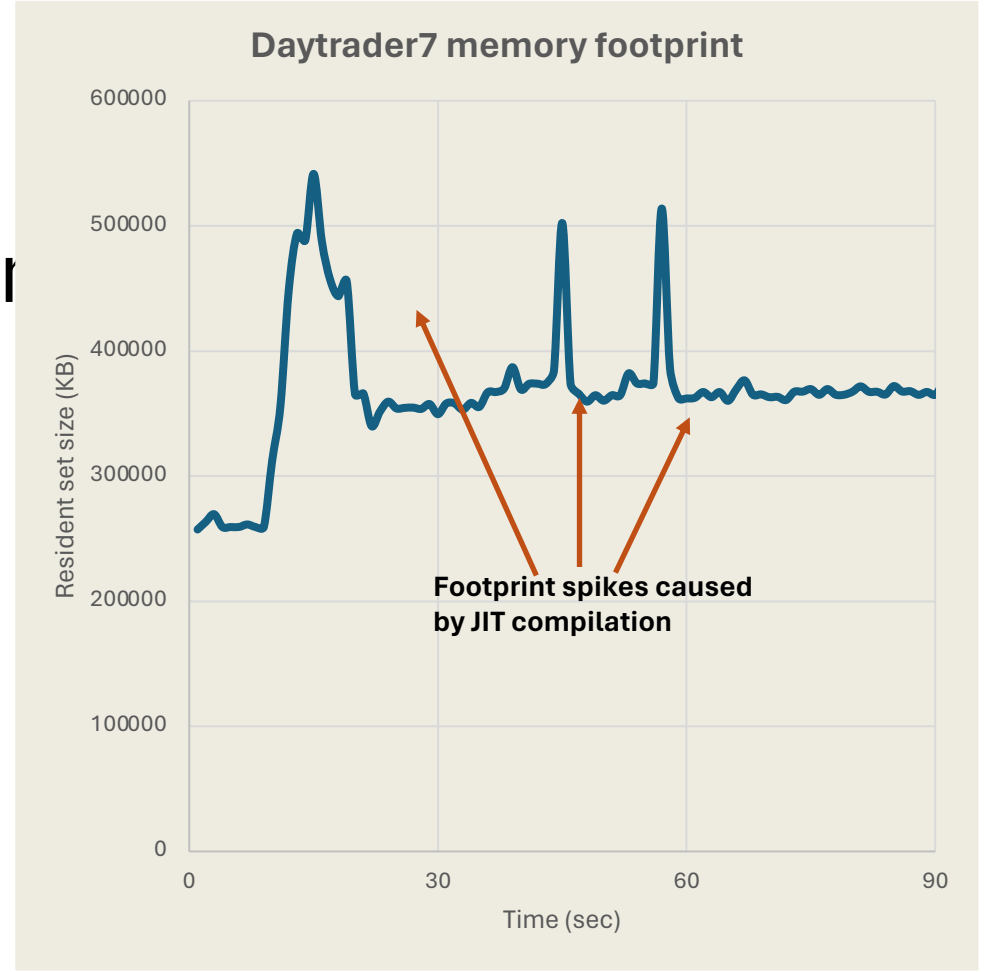
# CPU and Memory spikes



# Problem #1: Container Size

- Need to over-provision to avoid OOM issues/crashes and QOS
  - Very hard to do – JVMs have a non deterministic behavior
- Large containers = Low App Density
  - More cloud resources required
  - More dollars leave your account



# Problem #2: Responsiveness (Auto-Scaling)

- Slow start-up is non-starter for serverless
- CPU spikes during ramp-up can cause auto-scaler to **incorrectly** launch additional instances

# Sure would be nice to:

- Minimize/eliminate CPU and memory spikes
- Improve start-up and ramp-up times

- These are problems on which WebSphere Liberty, Semeru Runtimes, and Eclipse OpenJ9 focus

Get Started Guides Docs Support Blog

openliberty.io

# Open Liberty™

An IBM Open Source Project

**A lightweight open framework for building fast and efficient cloud-native Java microservices.**

Build cloud-native apps and microservices while running only what you need. Open Liberty™ is the most flexible server runtime available to Java™ developers in this solar system.

Get Open Liberty

# 6 reasons why Liberty


***Lightweight, highly-efficient runtime***

***CI/CD optimized operational experience***

***Simple true-to-production developer experience***

| | |
|---|---|
| Just enough runtime | 80% disk and 56% memory saving |
| Low operating cost | 4x increased density over Tomcat & Spring Boot |
| Continuous delivery | Zero-effort security fixing & zero technical debt |
| Zero migration | 100% v2v & fixpack migration saving |
| Kubernetes optimized | Self-tuned optimal perf, production-ready, kube-native |
| Developer experience | Container & kube-native experience, rapid inner loop |

# A little bit of history on the Liberty application server

- First shipped in WAS 8.5 in 2012
  - Servlet + JSP + JPA
- Web Profile 6 in 2014
- Java EE 7 in 2016 – first commercial product to certify
- Java EE 8 in 2018 – first to certify
- Jakarta EE 8 in 2019 – first to certify
- Eclipse MicroProfile – first to deliver 1.0-1.4, 2.0-2.1, 3.0




Liberty made available as open source “OpenLiberty” project in 2017 because success in today’s world requires Open Source



16

# Simple Config

```
<server>
  <featureManager>
    <feature>microProfile-4.0</feature>
  </featureManager>

  <webApplication location="myweb.war" contextRoot="/" />

</server>
```

server.xml

```
-Xmx1g
-Dsystem.prop=value
```

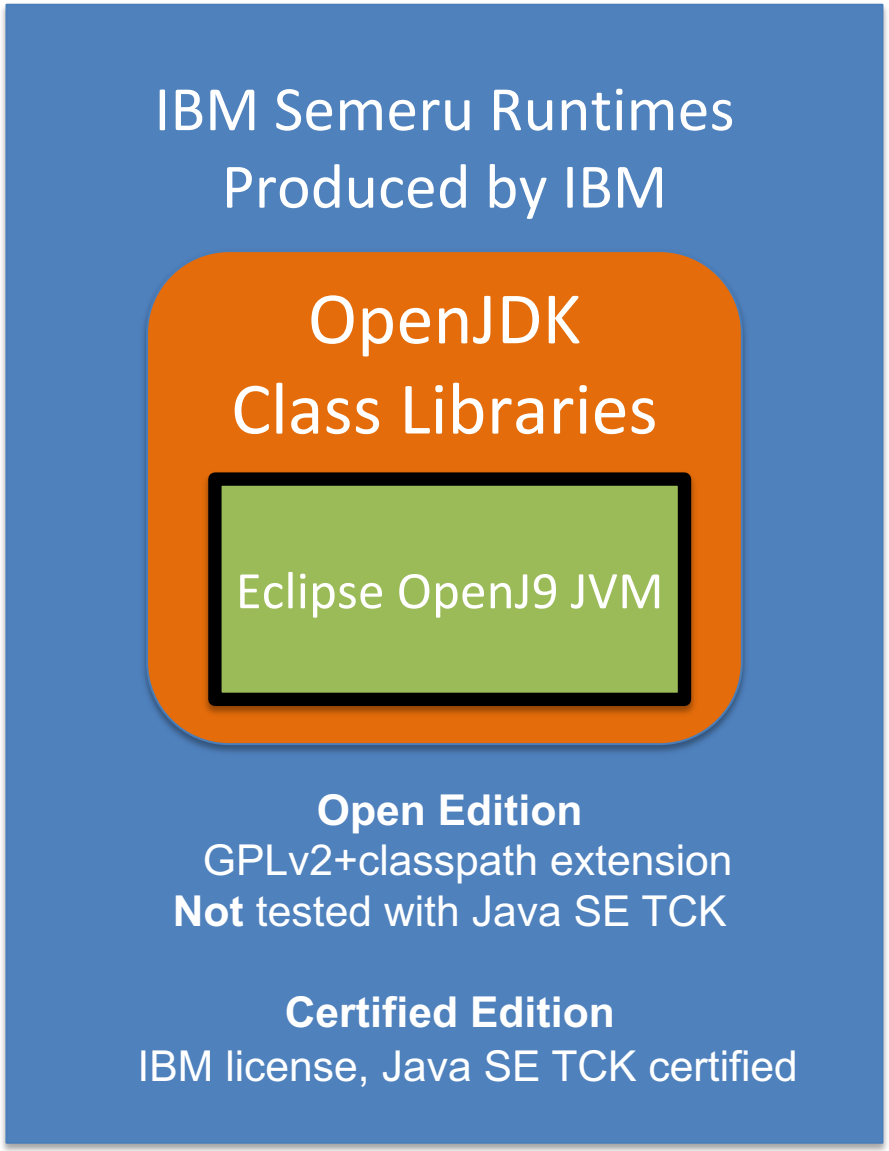
jvm.options

```
WLP_OUTPUT_DIR=/usr/wlp-out/
```

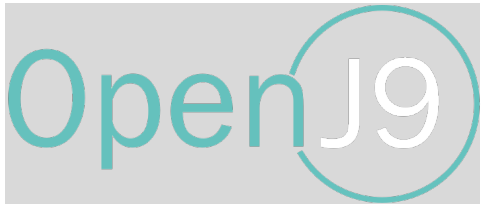
server.env

# IBM Semeru Runtimes

- Used in 1000s IBM Java-based products since 2006
- Available for use outside IBM products
- Two editions with different licenses but 100% functionally compatible
- All open source with no proprietary content except on z/OS platform
- Broad platform support
  - Definitely not “just” for IBM platforms!
  - IBM software has to run well on all platforms
- Also the only OpenJDK distribution that includes the Eclipse OpenJ9 JVM



# Eclipse OpenJ9



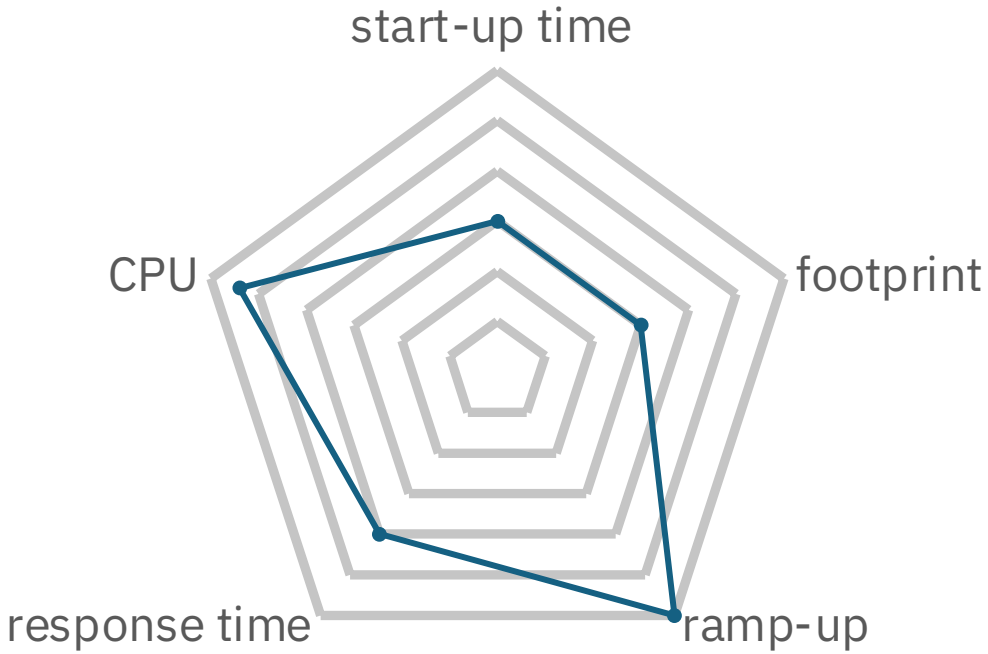

- Originally built in 1990s for use in embedded devices
  - Cell phones, oscilloscopes, handheld devices
  - Proprietary **independent JVM implementation** from when Java itself was proprietary
  - Contributed to Eclipse Foundation in 2016/2016 as Eclipse OpenJ9

- Several key technology investments and choices different than Hotspot
  - Architectural leaning to use memory efficiently due to embedded heritage
  - InstantOn leverages Linux CRIU project to start Java workloads in milliseconds (built in conjunction with OpenLiberty project but can be used by other projects)
  - Shared classes cache can transparently store loaded classes to reduce resident memory and accelerate future class loading for faster startup
  - Single JIT compiler technology (“Testarossa”) used for cold, warm, hot, veryHot, scorching optimization, compiled code can also be compiled to be cached in the shared classes cache
  - Same JIT compiler deployable as a separate server (same machine or remote) & many JVM clients
  - Single JVM source for all active Java releases from Java 8 to Java 21 (soon Java 22)

- These investments reflect a prolonged greater attention to fast startup time and low memory use in addition to raw speed, which can help in cloud deployments

# OpenJ9: A balanced approach to Java workload performance

- **Some key performance metrics :**
  - start-up time
  - ramp-up time
  - memory footprint
  - response time
  - CPU consumption
  - …




- Optimizing to improve one metric can negatively impact other metrics
  - Really important to measure all metrics that matter **at the same time**
- Different workloads and deployments may emphasize different metrics
- Myth: “OpenJ9 favours memory and startup over performance”
  - I prefer: “we aim for a more balanced approach that doesn’t unduly favour raw speed”
  - That last 10% of raw speed may be better spent improving other metrics

# Start up and Memory with Semeru Runtimes

**Java workloads start 50% faster**

**So?**

- Reduced down time
- Faster recovery from “events”
- Faster (re)deploy time
- Support elastic workloads to save costs
- Fully Java compliant

**Java workloads can use half the memory**

**So?**

- More memory means bigger VMs
- e.g. AWS: 2X memory -> 2X cost
- 2 smaller VMs cost the same but may handle more load than one big VM
- Fully Java compliant

**Liberty&Semeru InstantOn starts Linux containers another 5X-18X faster**

- Take a snapshot after startup, then restore from snapshot in milliseconds
- Fully Java compliant
- Available now with Liberty for Java EE applications*
- Works in Linux containers on Windows WSL & MacOS

**Semeru Cloud Compiler reduces peak memory consumption**

- Offloads JIT compiler overheads to service
- Containers can run in even less memory and with fewer (even < 1) cores
- Containers pack more densely onto nodes
- In the lab, seen cost savings as high as 33%
- Fully Java compliant

* Not all Liberty profiles supported yet

# OpenJ9 Shared Classes Cache

Transparently stores classes and JIT code
for faster startup on all platforms

Open Liberty

# OpenJ9 Shared Class Cache (SCC) Technology

- Memory mapped file used to cache
  - ROMClasses (pre-processed .class file contents, i.e. “read only” class data)
  - AOT (ahead-of-time) compiled code
  - Many kinds of profiling data collected by JVM

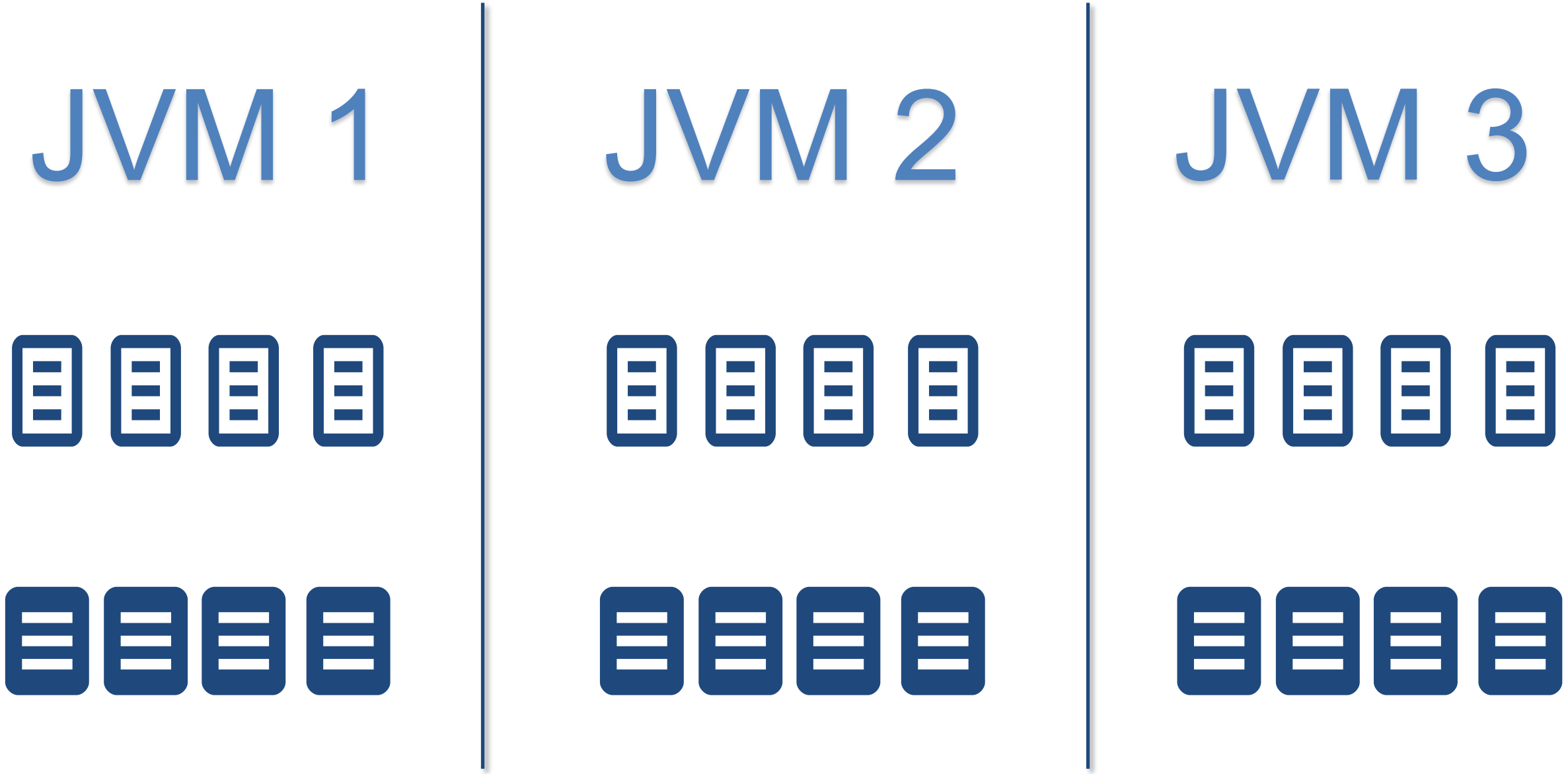
- Enabled by option : -Xshareclasses
  - “Cold run”: SCC populated with classes used by application
  - “Warm run”: application loads classes from SCC
    - Avoids file access; desired content likely present in RAM
    - Avoids some processing of classes

- SCC is persistent: survives machine reboot

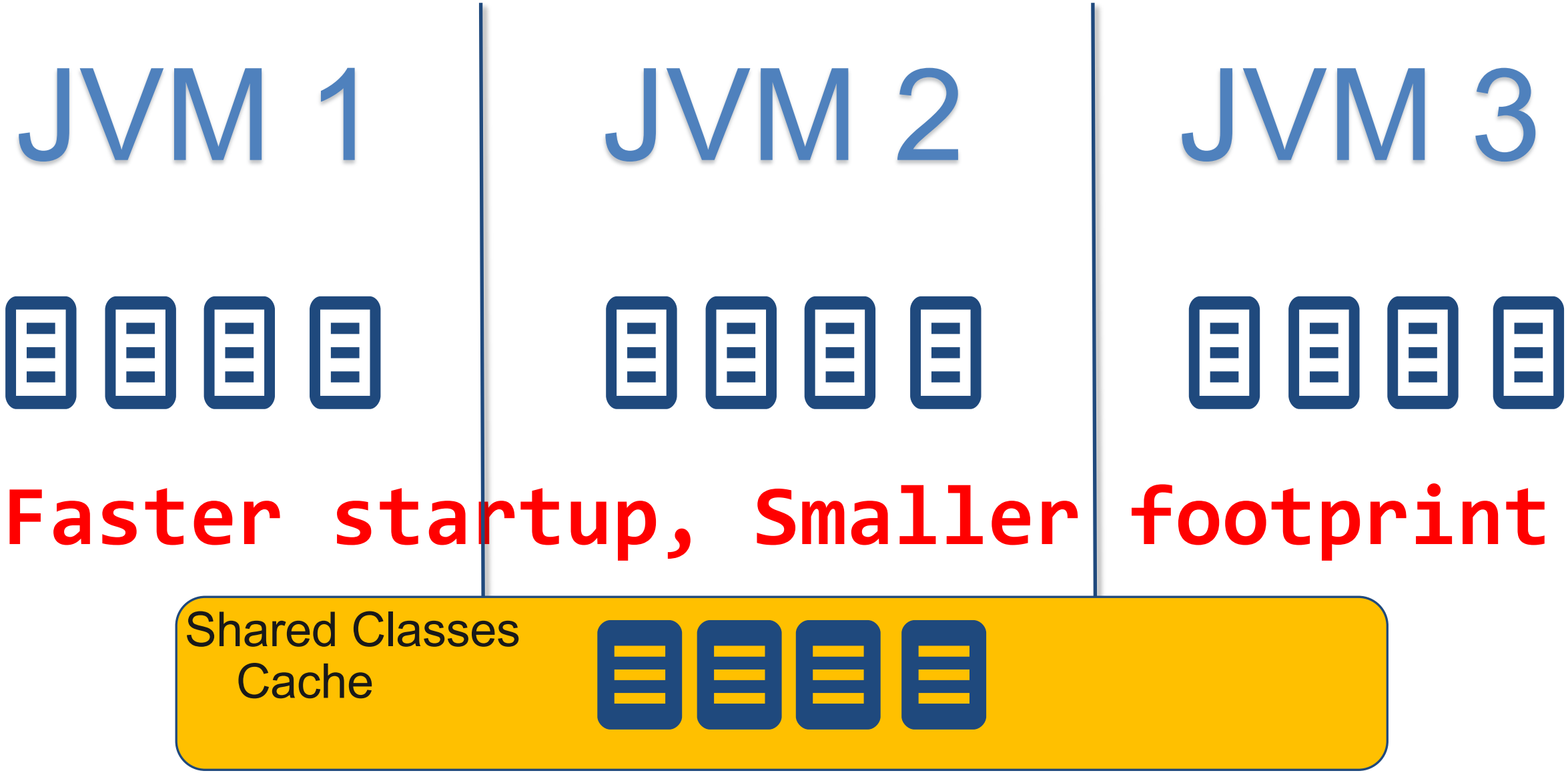
- Multiple JVMs can concurrently attach to the same SCC → footprint (RSS) savings due to ROMClass sharing in physical memory

# Class-related Metadata in OpenJ9

# SharedClasses Cache in OpenJ9



# SharedClasses Cache in OpenJ9



# Static vs. Dynamic Ahead of Time (AOT)

- Static AOT: Precompile all methods from a jar file
  - Generate off-line (ahead-of-time) from running application
  - GCJ, Excelsior JET approach, Graal native images

- Dynamic AOT: Run application and compile only methods important for performance
  - Significantly fewer methods compiled
  - Code is optimized based on application behavior
  - Used by Eclipse OpenJ9

# OpenJ9 Ahead of Time (AOT) Compilation

- AOT code stored in the shared classes cache (SCC)
  - Relocatable code generated during the cold run
  - Subsequent warm runs benefit from fast “AOT loads”

- Yields faster application start-up
  - JIT compilation overhead greatly reduced (AOT loads 100x faster than JIT compiles)
  - Compiled code is available sooner (AOT code 10x faster than interpreting)

- AOT-loaded methods may be recompiled for higher throughput

# Switch to the workshop

Some caveats:

If you're running on MacOS:

- I was unable to get a host browser to be able to access the servers in the inner containers
  - Please ignore lines like "you should be able to load the front page"
  - I know this was working at one point, but it isn't working for me now and I'm not sure why

If you're running on Windows:

- I was unable to get "podman stats" to work inside the main container
- Could be my configuration at home with an older Windows10 machine with Ubuntu 22 or 24

If you can make it work:

- Suggest opening a window with "podman stats" running all the time (will show you in a moment)
- Then you can ignore the steps inside the SharedCache workshop that ask you to start it

# What you learned

Liberty starts substantially faster using shared classes

Tomcat initially looked to start very slowly with Semeru Runtimes and even with class sharing enabled!

- But we realized that we were measuring “cold” runs
- Cold runs are even a lot slower than runs that do not share classes!
- When we added a prepopulation to the container build step, startup improved tremendously!
- Hopefully your results looked something like:

1 core results

| JDK | Start time | Memory |
|---|---|---|
| Temurin | 1152.47ms | 68MB |
| Semeru NOSCC | 1977.13ms | 42MB |
| Semeru SCC | 3234ms | 39MB |
| Semeru Prepop SCC | 593ms | 36MB |

2 core results

| JDK | Start time | Memory |
|---|---|---|
| Temurin | 633.684ms | 73MB |
| Semeru NOSCC | 1148.12ms | 42MB |
| Semeru SCC | 1624.24ms | 39MB |
| Semeru Prepop SCC | 448ms | 36MB |

# Liberty/Semeru InstantOn

For Serverless/Functions/Scale-to-zero on
Linux platforms without Java limitations

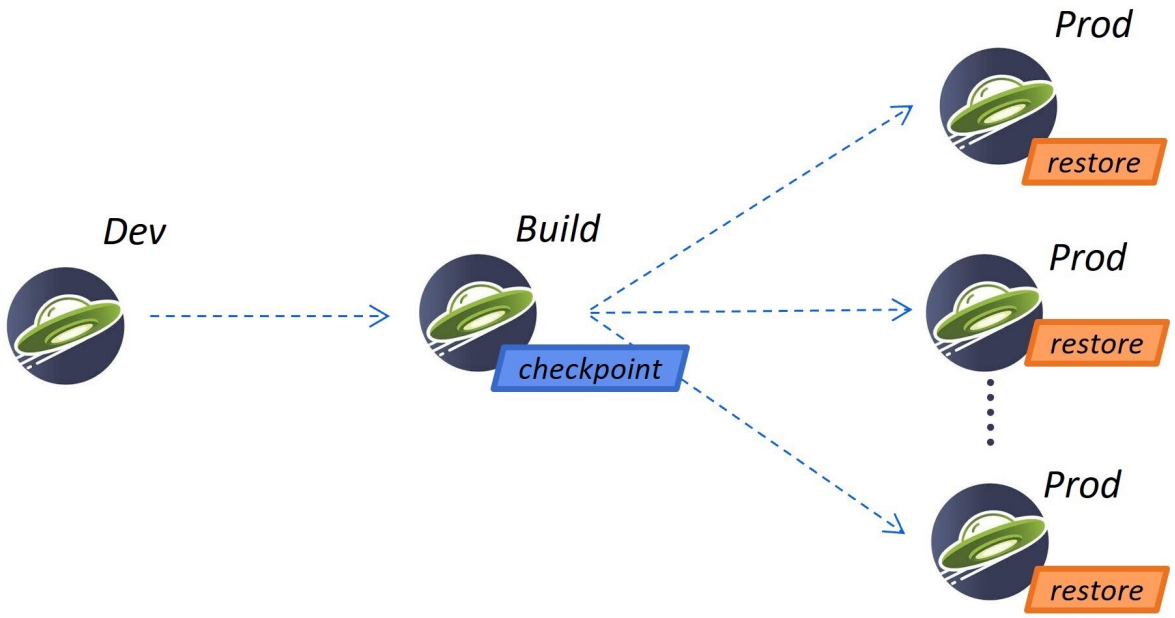
# Liberty and Semeru InstantOn : fast startup using Linux CRIU

Target Liberty application container deployments

Start application containers in milliseconds, ideal for serverless

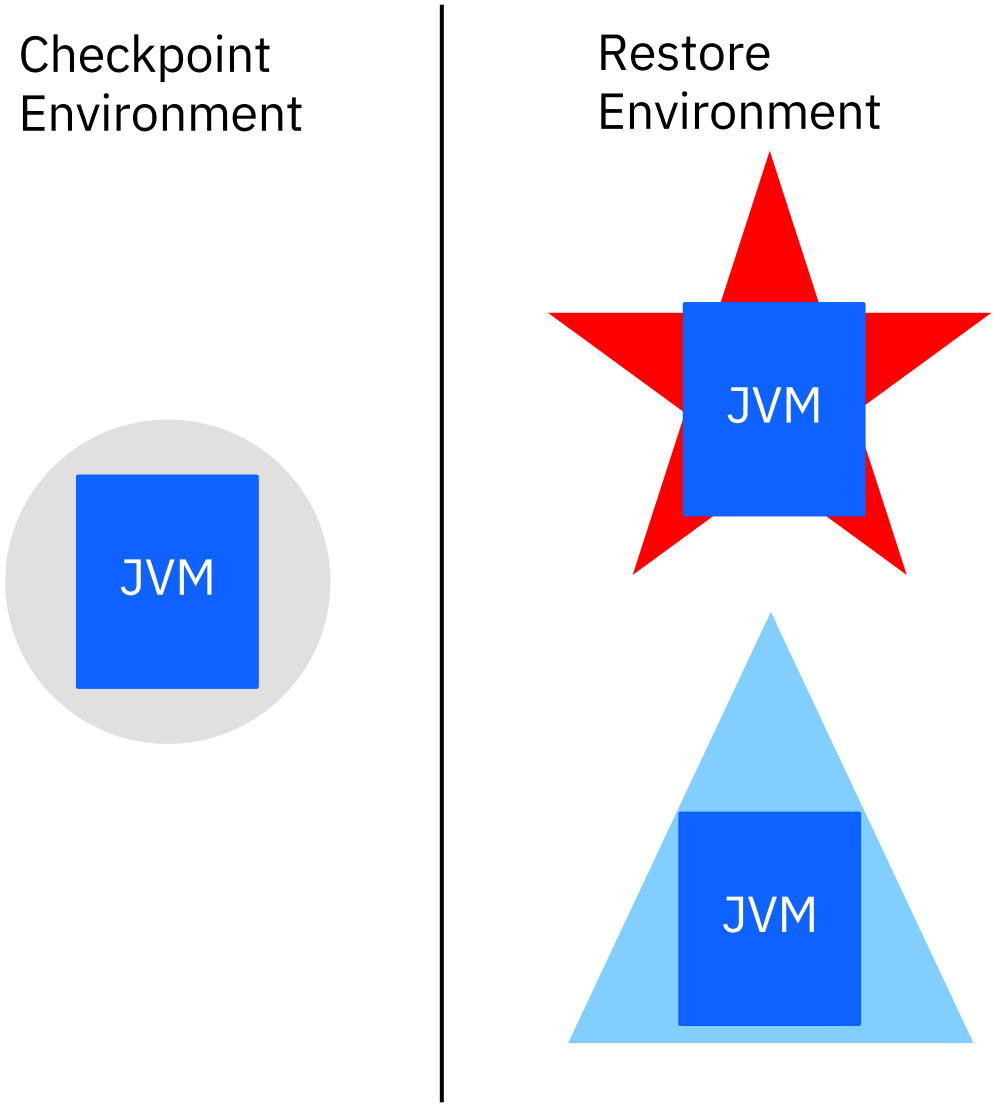
Leverages Linux CRIU to perform checkpoint / restore

Make it really easy to consume for a user of Liberty containers




| Characteristics | Liberty & Semeru InstantOn | Semeru JVM | Graal Native |
|---|---|---|---|
| Full Java support | Yes | Yes | No |
| 'Instant on' | Yes | No | Yes |
| High throughput | Yes | Yes | No |
| Low memory (under load) | Yes | Yes | Yes? |
| Dev-prod parity | Yes | Yes | No |

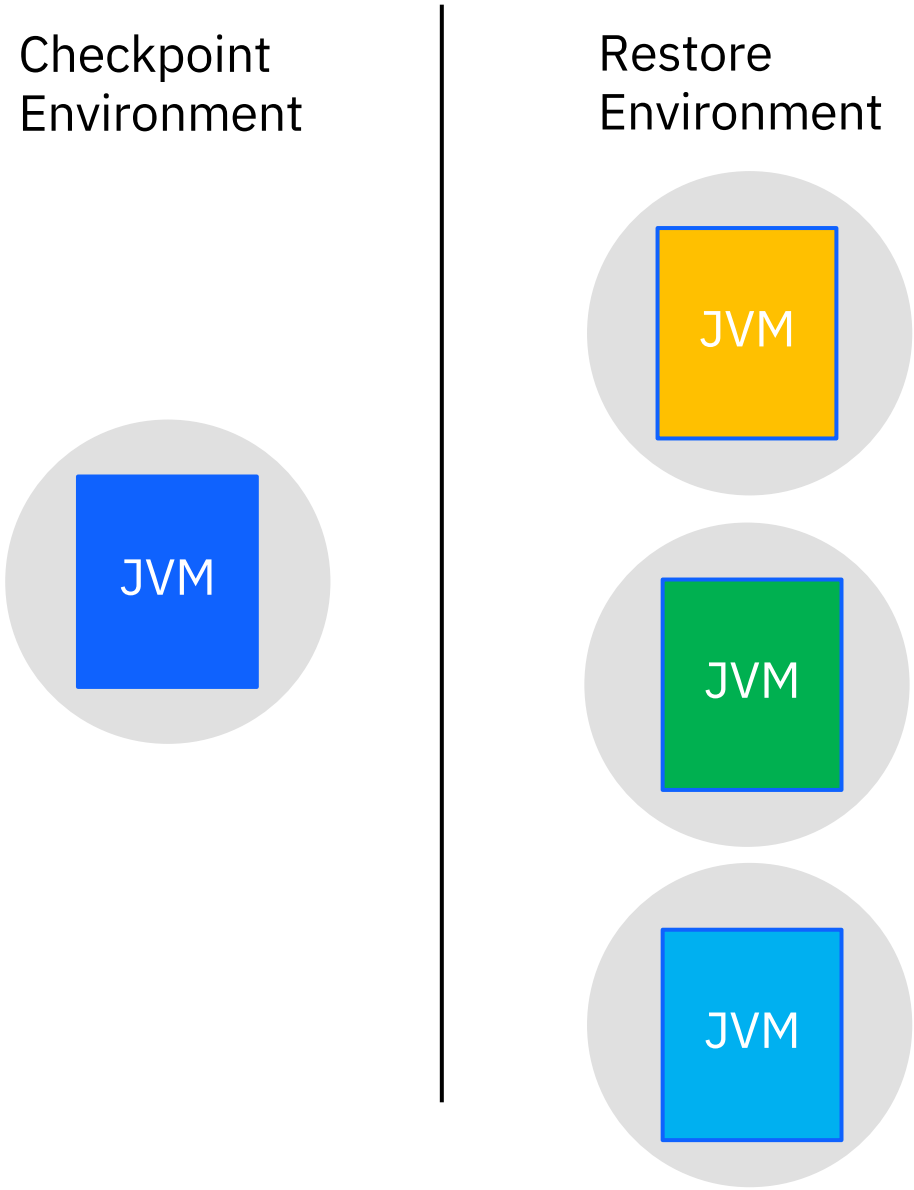
# Restoring Application state




**Environment Incompatibilities:** When restoring the JVM, there are several incompatibilities that may arise when restoring at a later time, on a different machine or both

1. Invalid time objects
   - LocalDateTime, Timer
2. Different machine configurations
   - # of CPUs, amount of RAM, resource limits
3. Different network or OS configuration
   - Different host name
   - Different environment variables
   - Different locales

# Restoring Application state




**Uniqueness:** When restoring the JVM there must be a way to make part of the JVM distinct from another JVM restored from the same image

1. Random seeds
   - Random
2. Security APIs
   - SecureRandom
   - SSL

# InstantOn:Where to checkpoint?

Liberty InstantOn leverages Semeru to provide a seamless checkpoint/restore solution for developers. With checkpoint restore, there is a tradeoff between startup time and the complexity of the restore.

Checkpoint phases

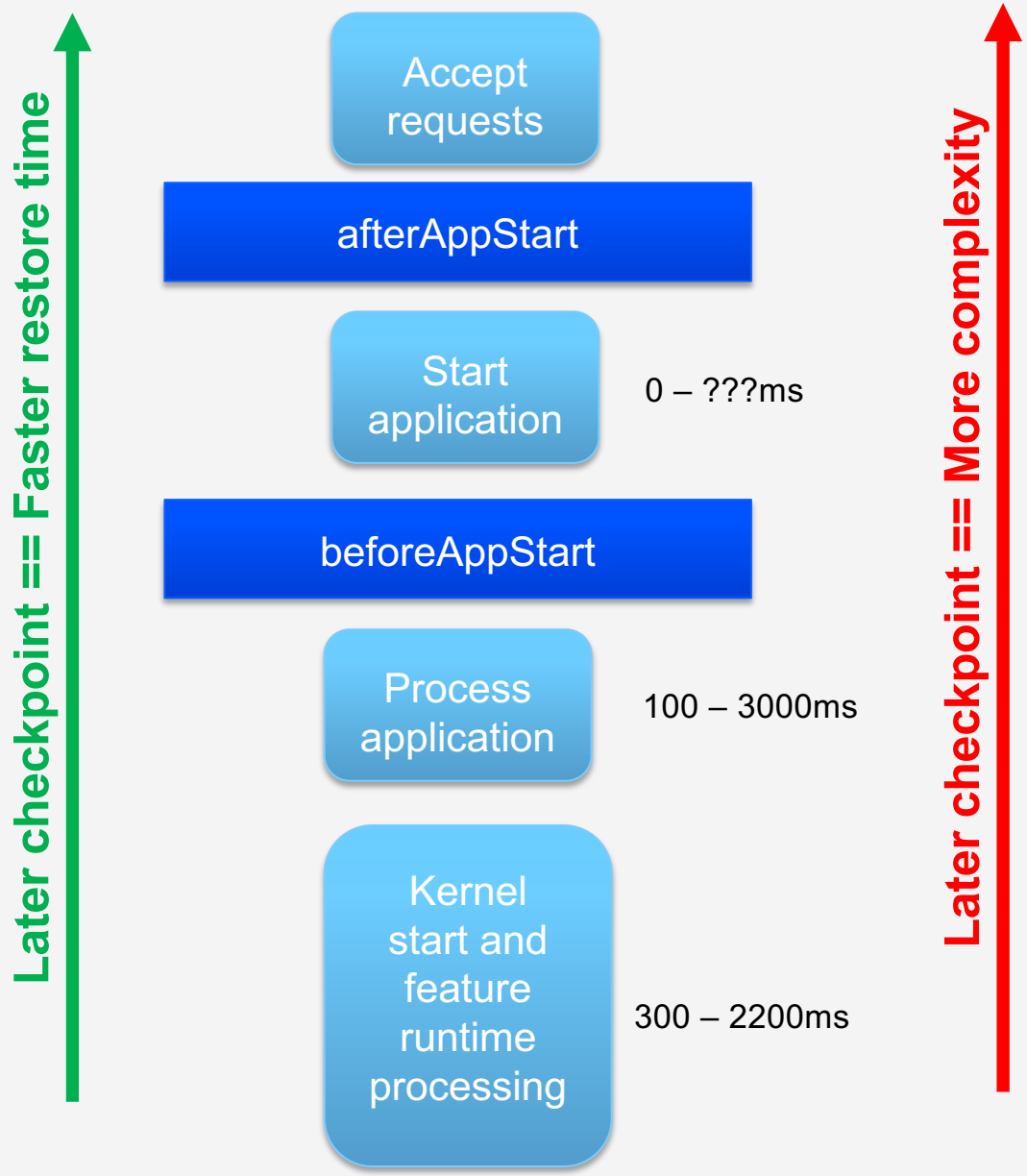
- `beforeAppStart`
- `afterAppStart`

Choice of where to checkpoint depends on what application start code does, e.g., any of the below would make `afterAppStart` unsuitable

Accessing a remote resource, such as a database

Reading configuration that is expected to change when the application is deployed

Starting a transaction



# Switch to the workshop

# What you learned

You started Liberty using Eclipse Temurin first just to get a baseline startup performance

- Temurin does not support InstantOn or comparable technology, so it is the slowest option

You started Liberty using Eclipse OpenJ9 and shared classes just to see how much startup improved

- With the default prepopulated shared classes cache, Libert starts in roughly half the time

You started Liberty using both the beforeAppStart and afterAppStart checkpoints

- beforeAppStart starts dramatically faster, afterAppStart starts in about half of even that time

Semeru Runtimes uses less than a third to less than a half the memory Temurin uses to start Liberty

| | 1 core results | | 2 core results | | 4 core results | |
|---|---|---|---|---|---|---|
| JDK | Start time | Memory | Start time | Memory | Start Time | Memory |
| Temurin | 5.524s | 191MB | 2.531s | 232MB | 1.913s | 272MB |
| Semeru | 2.971s | 95MB | 1.933s | 96MB | 1.718s | 97MB |
| Semeru beforeAppStart | 0.517 | 99MB | 0.425 | 101MB | 0.368 | 97MB |
| Semeru afterAppStart | 0.253 | 95MB | 0.228 | 95MB | 0.250 | 95MB |

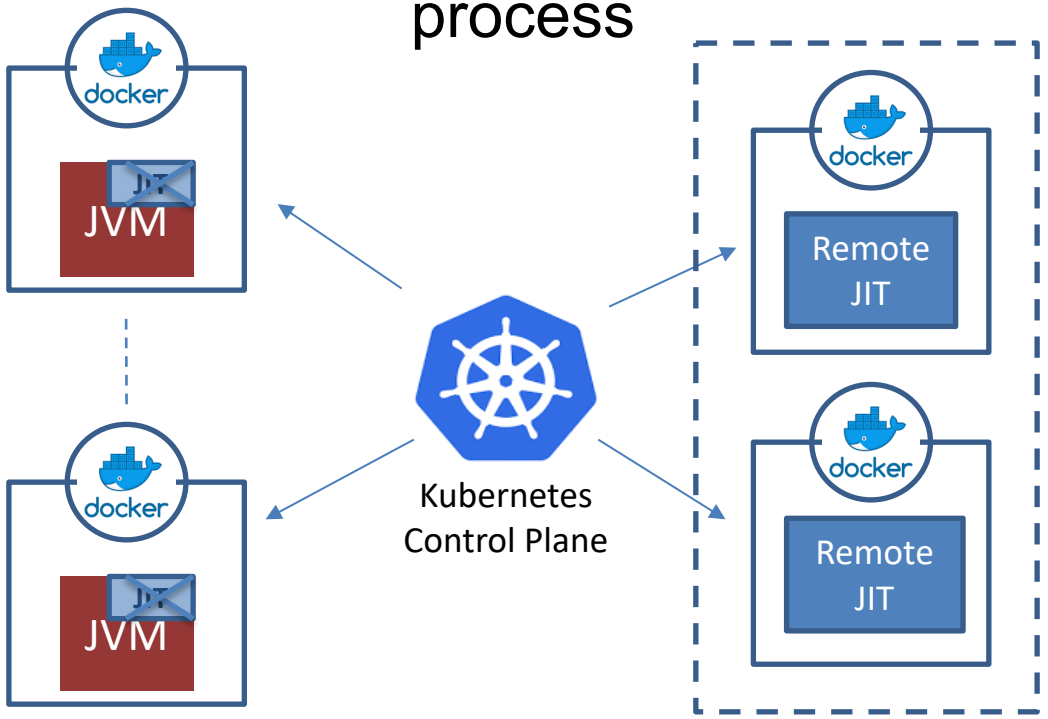
# Semeru Cloud Compiler

For Microservices


# JIT-as-a-Service

Decouple the JIT compiler from the JVM and let it run as an independent process

**Offload JIT compilation to remote process**




**Treat JIT compilation as a cloud service**

- Auto-managed by orchestrator
- A mono-to-micro solution
- Local JIT still available

# JITServer advantages for JVM Clients


## Provisioning

Easier to size;  only consider the needs of the application

## Performance

Improved ramp-up time due to JITServer supplying extra CPU power when the JVM needs it the most.

## Resiliency

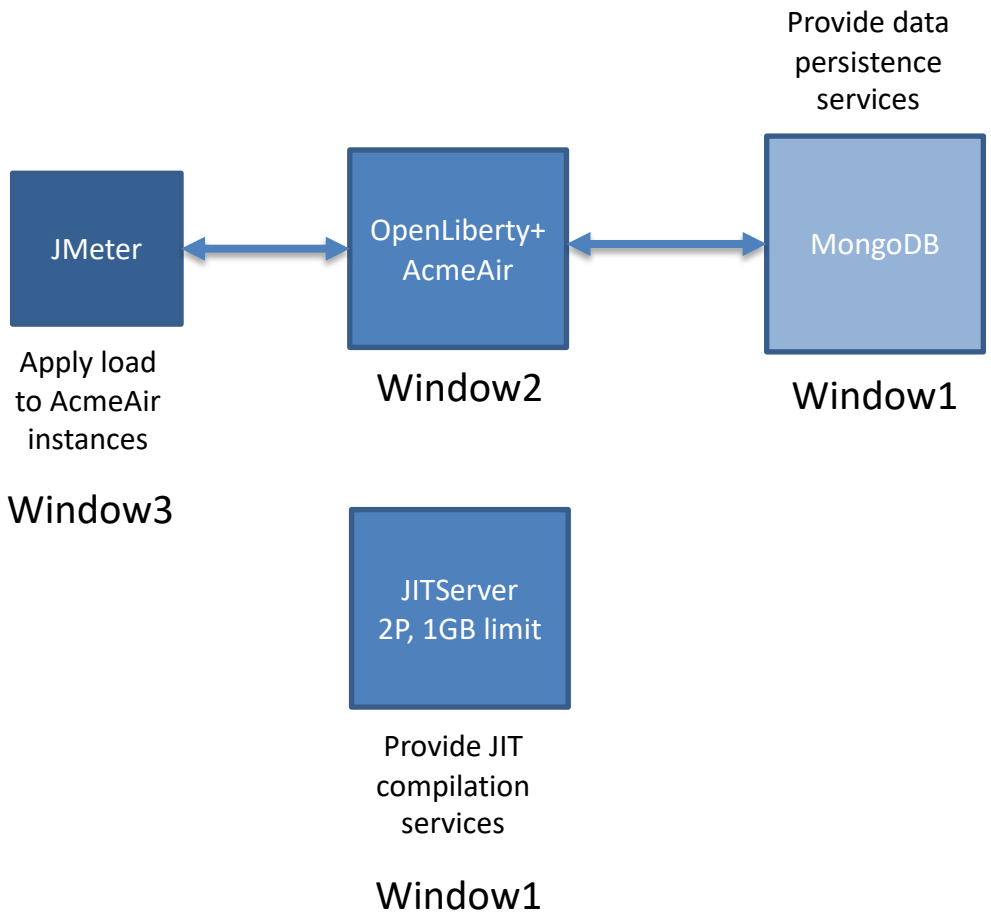
If the JITServer crashes, the JVM can continue to run and compile with its local JIT

## Cost

Reduced memory consumption means increased application density and reduced operational cost.

Efficient auto-scaling – only pay for what you need/use.

# Experimental setup with AcmeAir



# Switch to the workshop

## Slightly different than earlier examples

- This part of the workshop will be driving load at the Liberty server
- First step will be to build the containers used: run `all.build.sh` in the containers/ directory

## The sections in this lab are more complex

- You will need multiple windows open and you'll be typing commands into different windows
- The section README.sh provides the sequence you'll go through
- Each section has directories Window1, Window2, etc. with the scripts to run in those windows
- The scripts are numbered as stepN_start.<container>.sh with N=1,2,3,4
- Make sure you're running the scripts in numerical order and you should be fine

## Had hoped to but was unable to get Grafana able to show results graphically

- You will be watching textual performance results printed every 5 seconds

# What you learned

You saw Liberty ramp-up with 1 CPU core and 400MB limit using all that memory at times

You saw how easily it was to connect Liberty to a JIT server

- You saw Liberty with a JIT server start in the same environment ramp up more quickly to peak

You saw that Liberty could run with 1 CPU core and 200MB limit if using a JIT server

- Liberty even ramped up as quickly as with 400MB limit

# Wrap up

- We learned how shared class cache helps Semeru Runtimes to start Java applications almost 2X faster and consume half the memory
  - Looked at Liberty and Apache Tomcat servers
  - Looked at Eclipse OpenJ9 and Eclipse Temurin (using HotSpot)
- We learned how Liberty & Semeru InstantOn to start Liberty as much as 21.8X faster without growing memory use and with no Java language limitations
  - Looked at Liberty server with Eclipse OpenJ9 and Eclipse Temurin
- We learned how Semeru Cloud Compiler helps Java application reach peak performance even with limited CPU and memory
  - Looked at Liberty server, Eclipse OpenJ9, running AcmeAir application with one CPU core and 200mb memory
  - Saw the Java application being resilient even if the server goes down