ORACLE

ORACLE

Optimizing Performance with GraalVM

Alina Yurenko

GraalVM Developer Advocate

Oracle Labs

November 02, 2019

Safe harbor statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

GraalVM Native Image Early Adopter Status

GraalVM Native Image technology (including SubstrateVM) is early adopter technology. It is available only under an early adopter license and remains subject to potentially significant further changes, compatibility testing and certification

Agenda

1 Performance metrics

2 JIT performance

3 AOT performance

4 Demo: JIT and AOT modes

5 Tools

Performance metrics

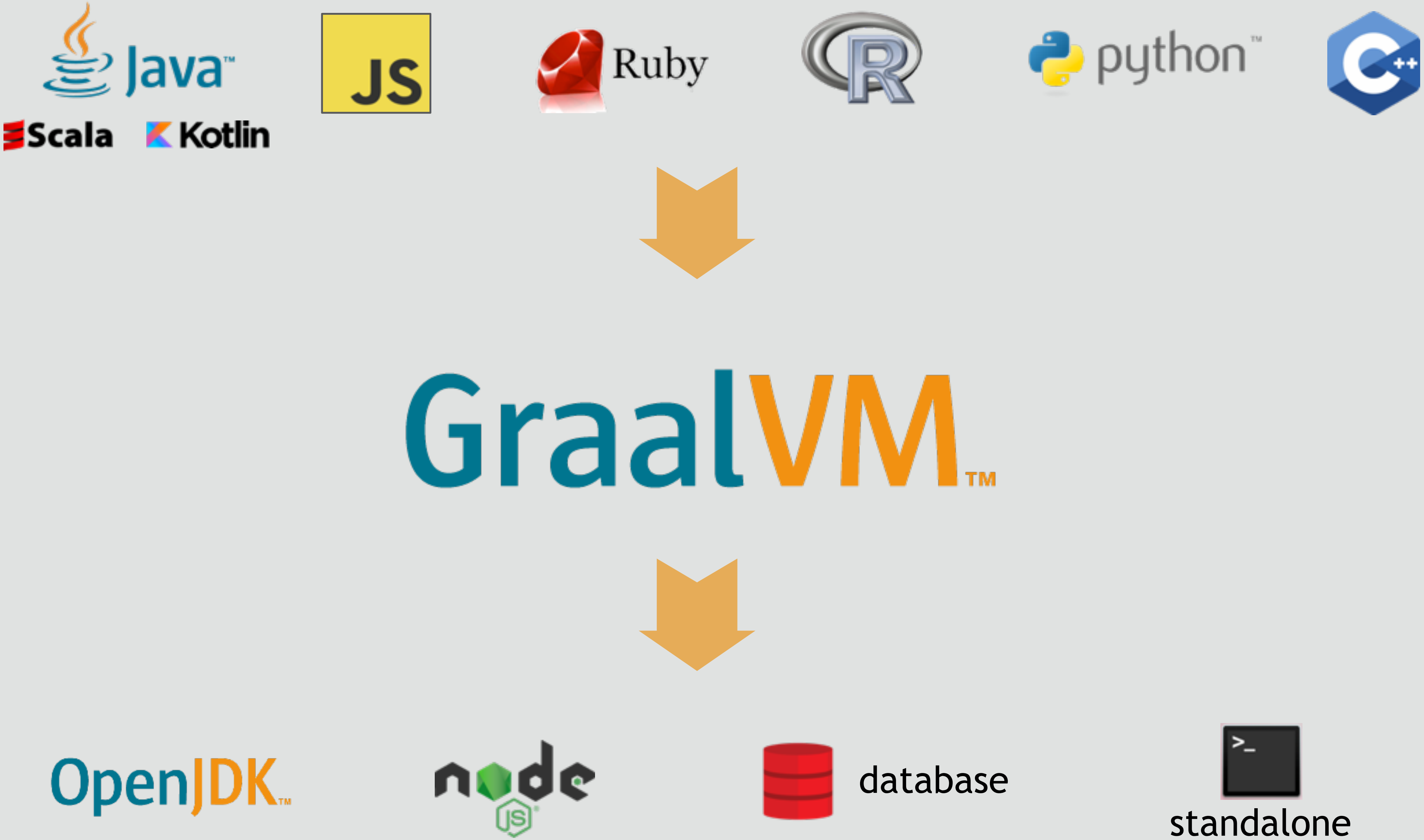
Performance metrics

- Throughput
- Latency
- Capacity
- Utilization
- Efficiency
- Scalability
- Degradation

Performance metrics

- Throughput
- Efficiency
- Scalability

JIT and AOT with GraalVM



Java™

JS

Ruby

R

python™

C++

Scala

Kotlin

GraalVM™

OpenJDK™

node JS

database

standalone

Get Started

GraalVM™

Run Programs Faster Anywhere

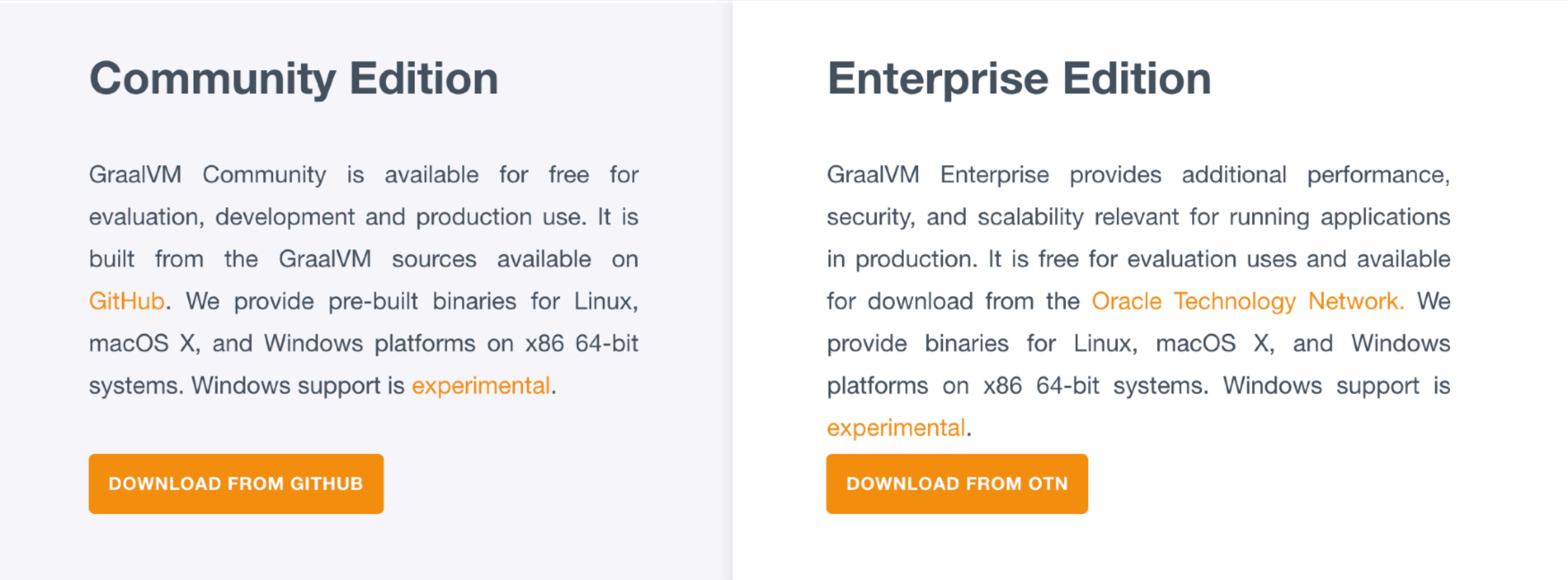
High-performance polyglot VM

GraalVM is a universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Kotlin, Clojure, and LLVM-based languages such as C and C++.

GraalVM removes the isolation between programming languages and enables interoperability in a shared runtime.

- Downloads
- Documentation
- Community support

GraalVM Versions

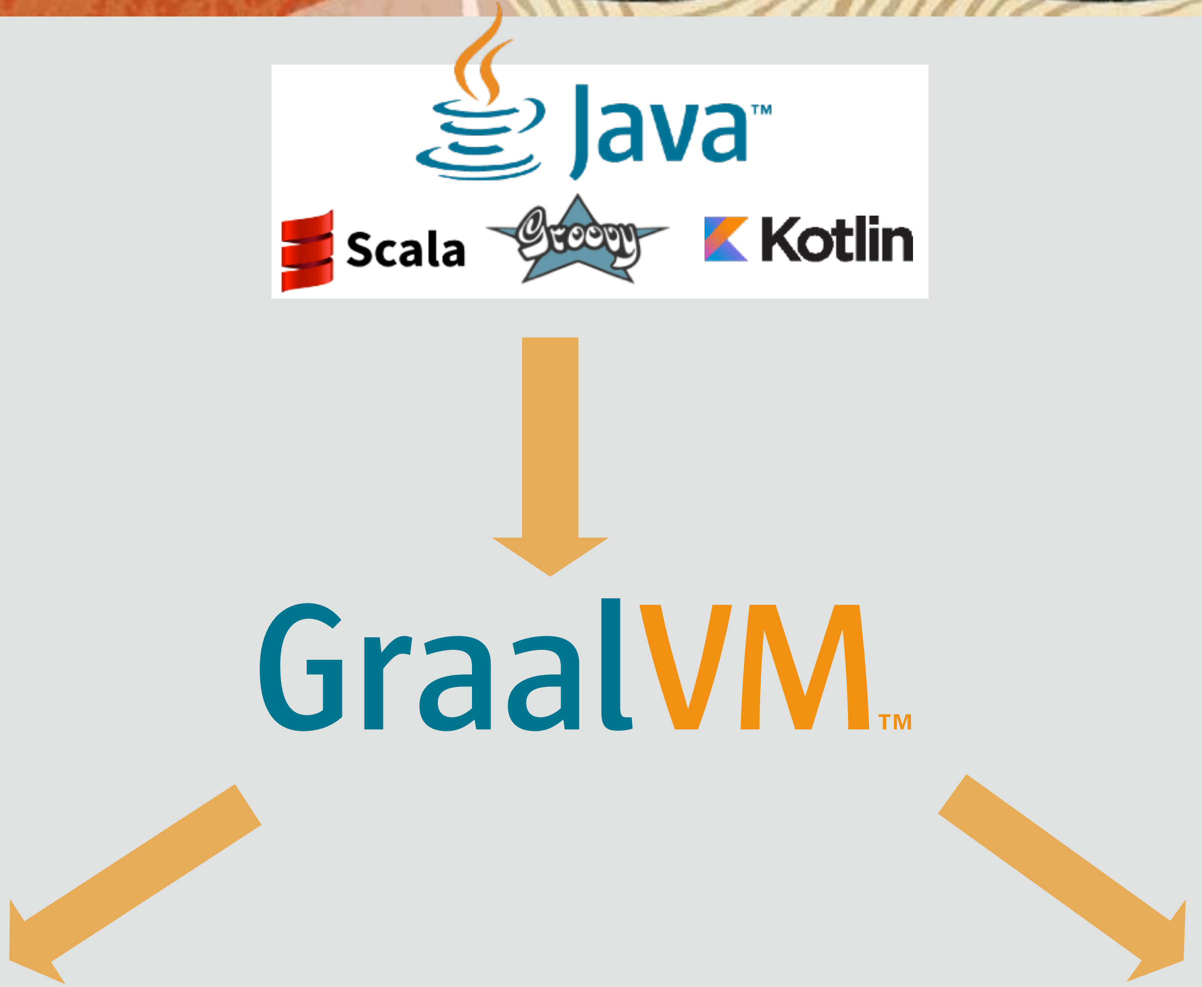


Community Edition

GraalVM Community is available for free for evaluation, development and production use. It is built from the GraalVM sources available on GitHub. We provide pre-built binaries for Linux, macOS X, and Windows platforms on x86 64-bit systems. Windows support is experimental.

Enterprise Edition

GraalVM Enterprise provides additional performance, security, and scalability relevant for running applications in production. It is free for evaluation uses and available for download from the Oracle Technology Network. We provide binaries for Linux, macOS X, and Windows platforms on x86 64-bit systems. Windows support is experimental.



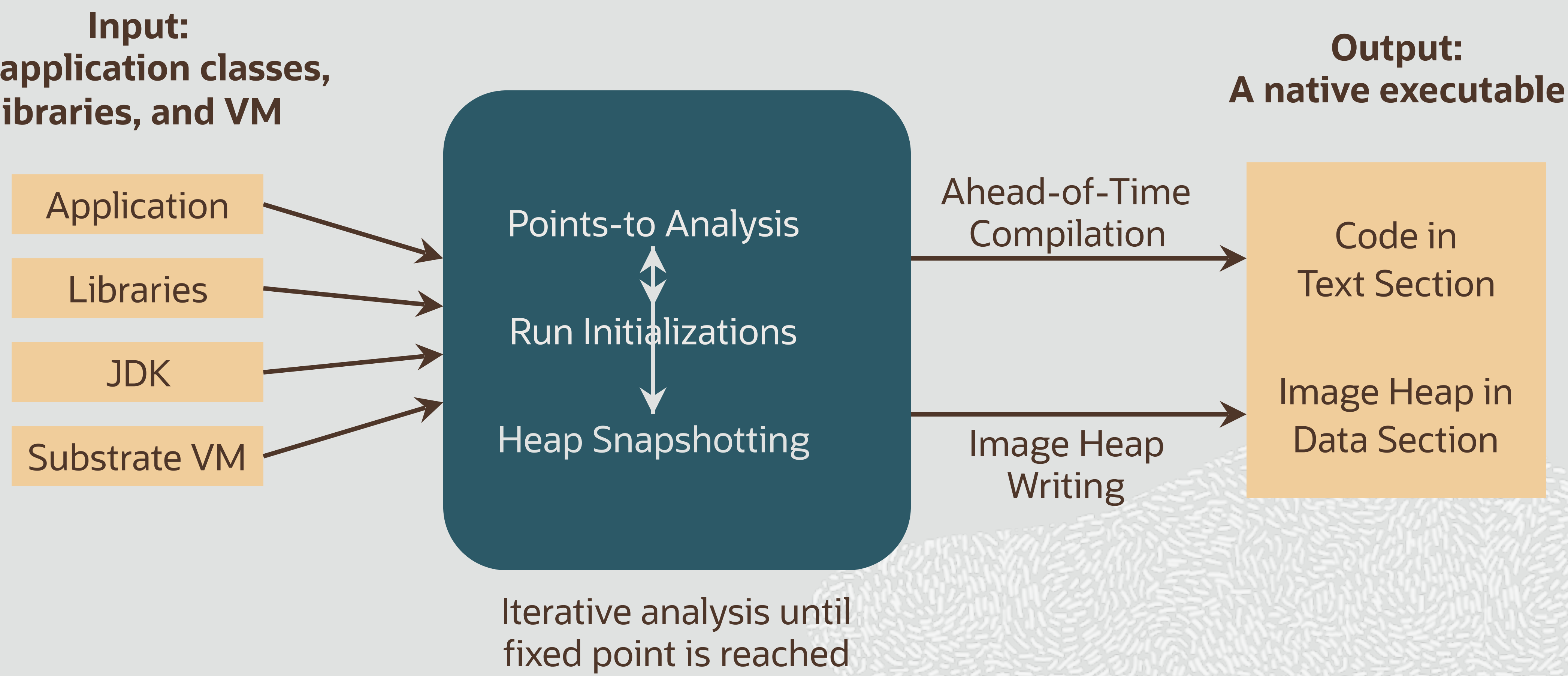
JIT

java MyMainClass

AOT

native-image MyMainClass
./mymainclass

How GraalVM native image works



Input:
All application classes, libraries, and VM

Application

Libraries

JDK

Substrate VM

Points-to Analysis

Run Initializations

Heap Snapshotting

Iterative analysis until fixed point is reached

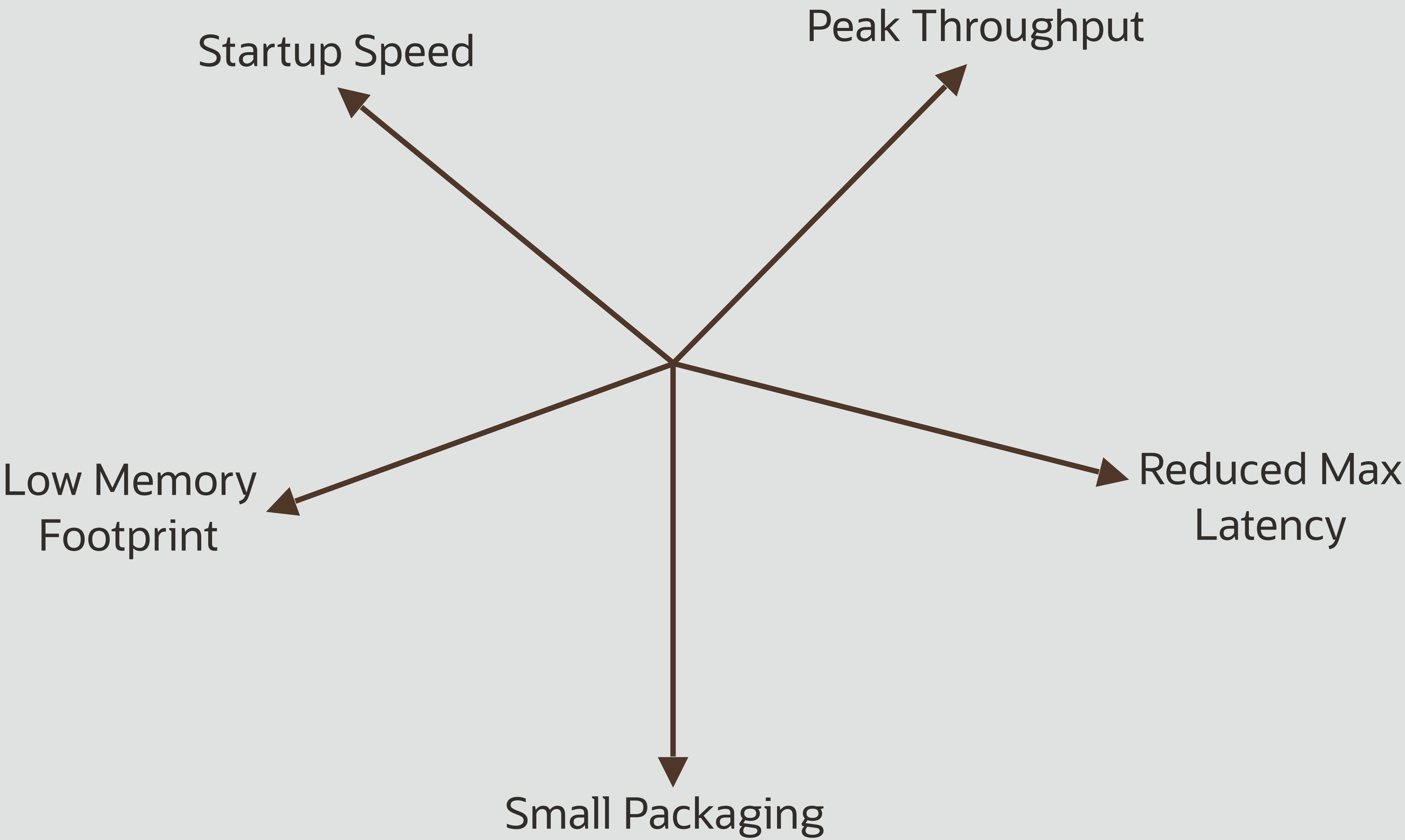
Ahead-of-Time Compilation

Image Heap Writing

Output:
A native executable

Code in Text Section

Image Heap in Data Section



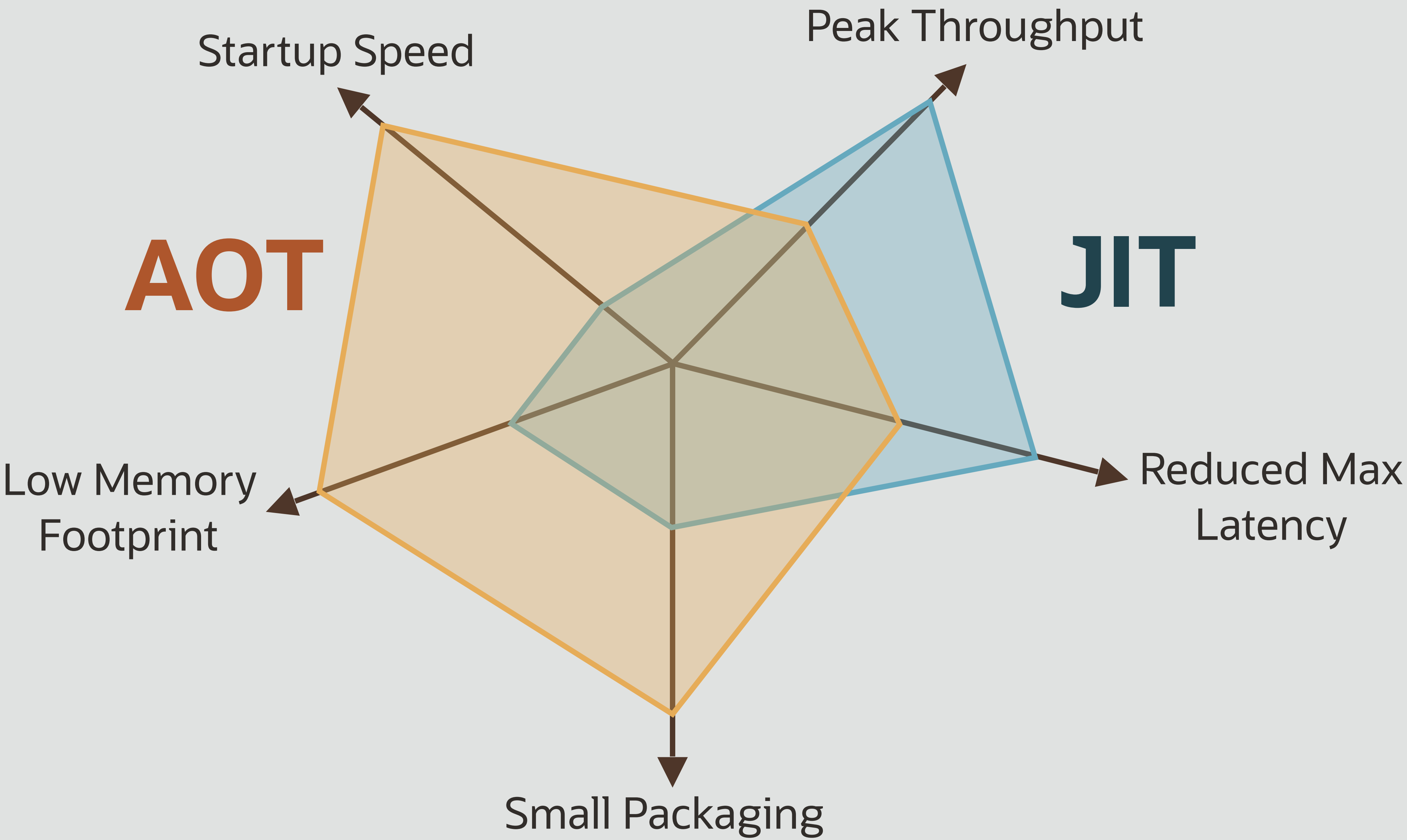
Startup Speed

Peak Throughput

Low Memory Footprint

Reduced Max Latency

Small Packaging


Startup Speed
Peak Throughput
AOT
JIT
Low Memory Footprint
Reduced Max Latency
Small Packaging

AOT vs JIT: Startup Time

JIT

- Load JVM executable
- Load classes from file system
- Verify bytecodes
- Start interpreting
- Run static initializers
- First tier compilation (C1)
- Gather profiling feedback
- Second tier compilation (GraalVM or C2)
- Finally run with best machine code

AOT

- Load executable with prepared heap
- Immediately start with best machine code

AOT vs JIT: Memory Footprint

JIT

- Loaded JVM executable
- Application data
- Loaded bytecodes
- Reflection meta-data
- Code cache
- Profiling data
- JIT compiler data structures

AOT

- Loaded application executable
- Application data

AOT vs JIT: Peak Throughput

JIT

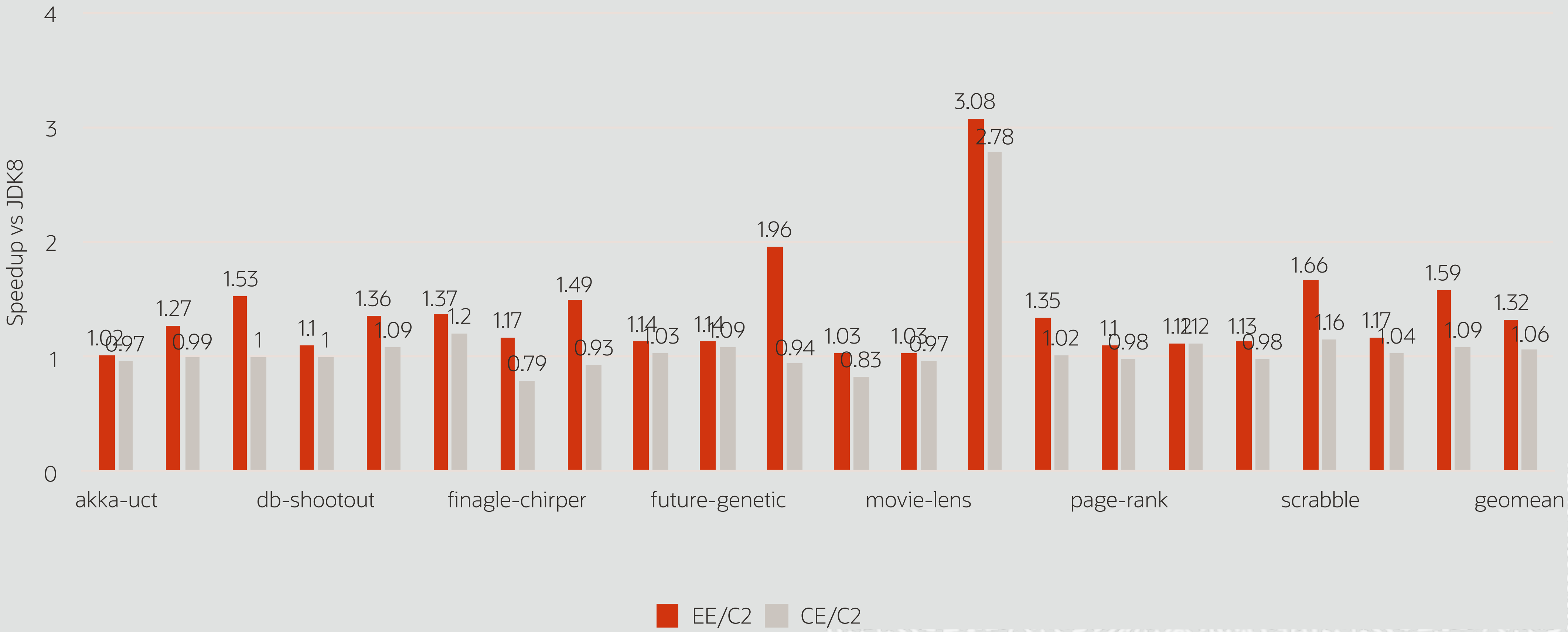
- Profiling at startup enabled better optimizations
- Can make optimistic assumptions about the profile and deoptimize

AOT

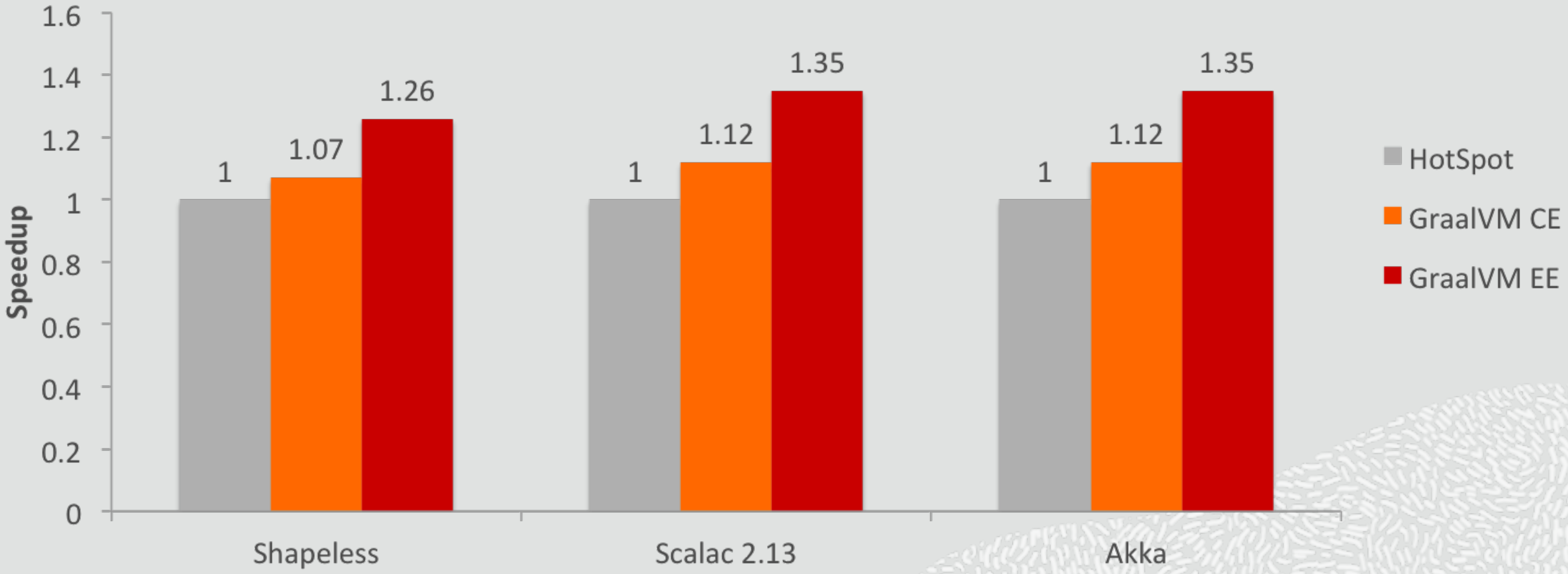
- Needs to handle all cases in machine code
- Profile-guided optimizations help
- Predictable performance

Demo time

GraalVM JIT Performance: Renaissance.dev

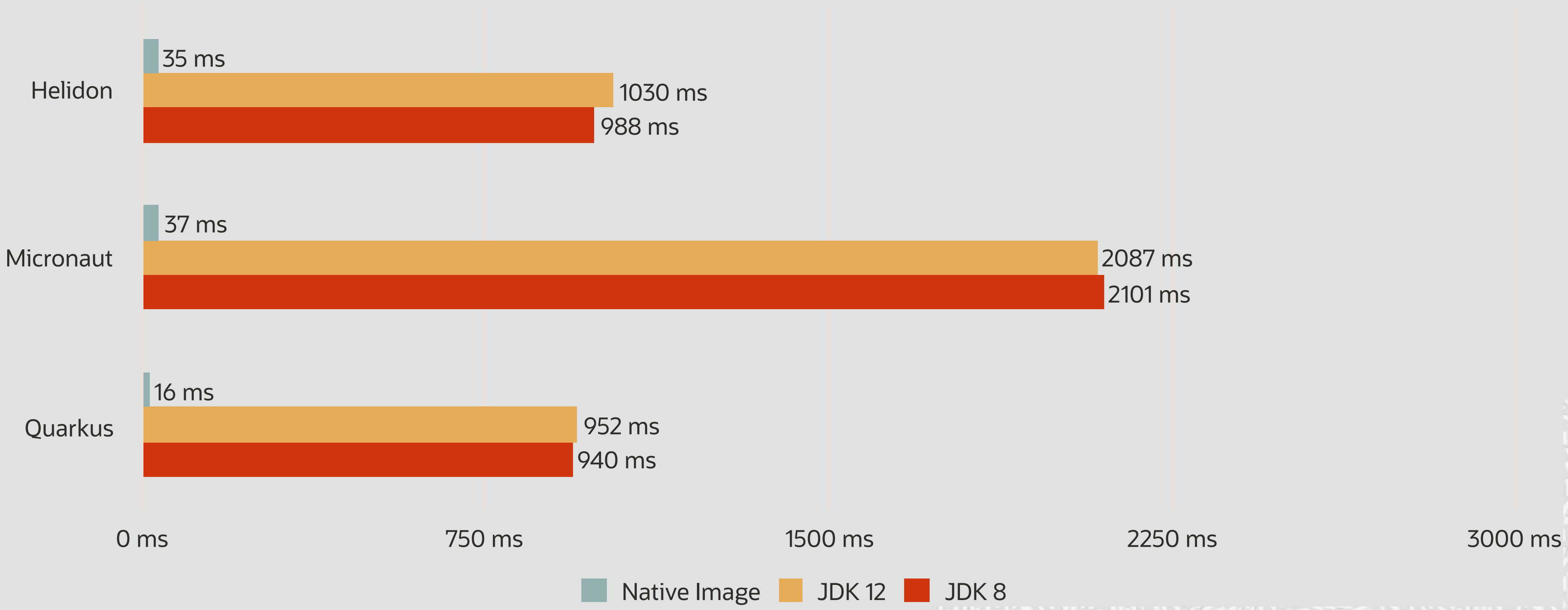


Scala Performance

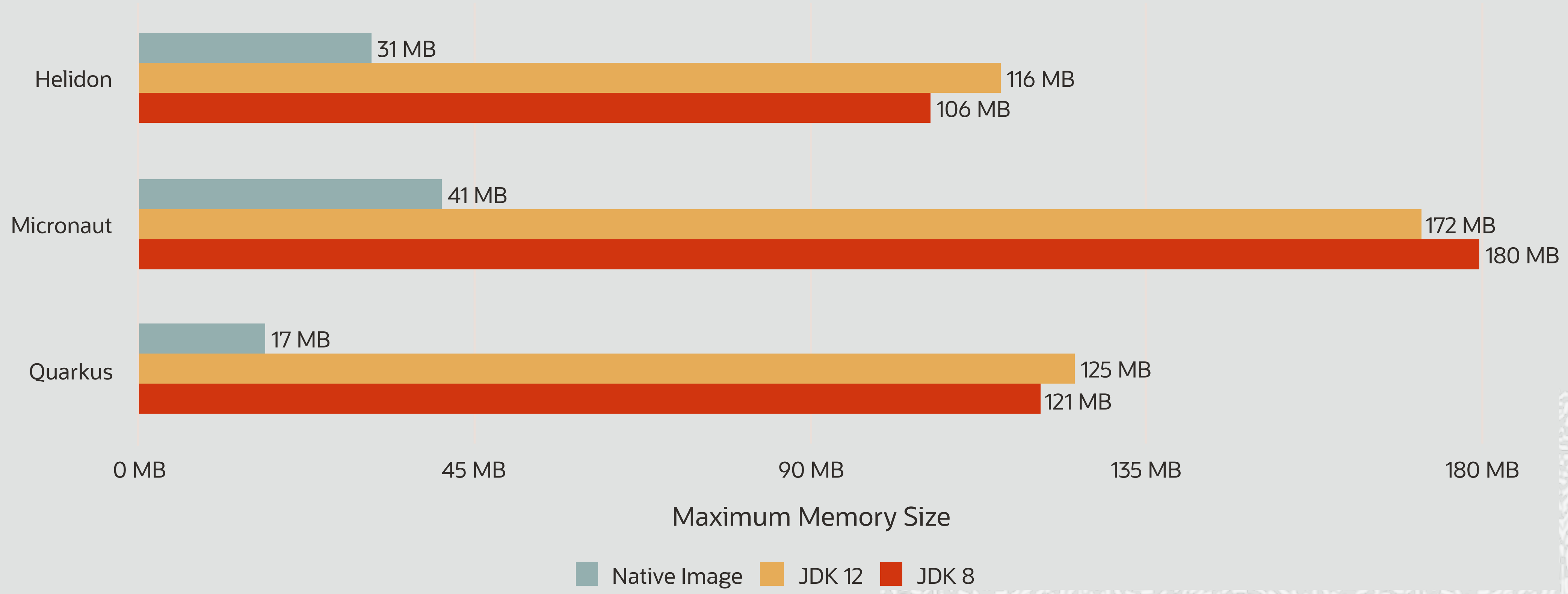


https://medium.com/graalvm/compiling-scala-faster-with-graalvm-86c5c0857fa3

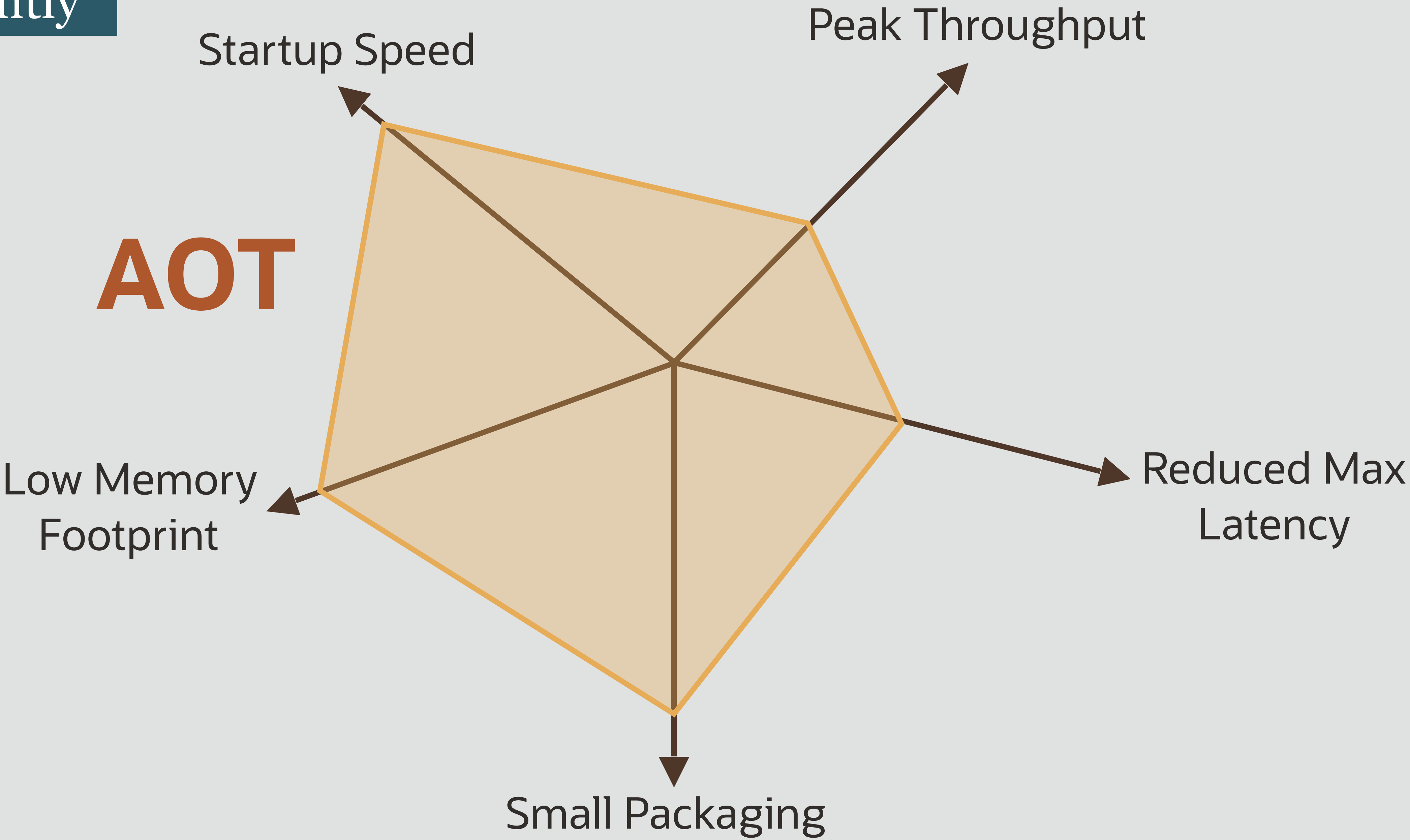
Microservice Frameworks: Startup Time



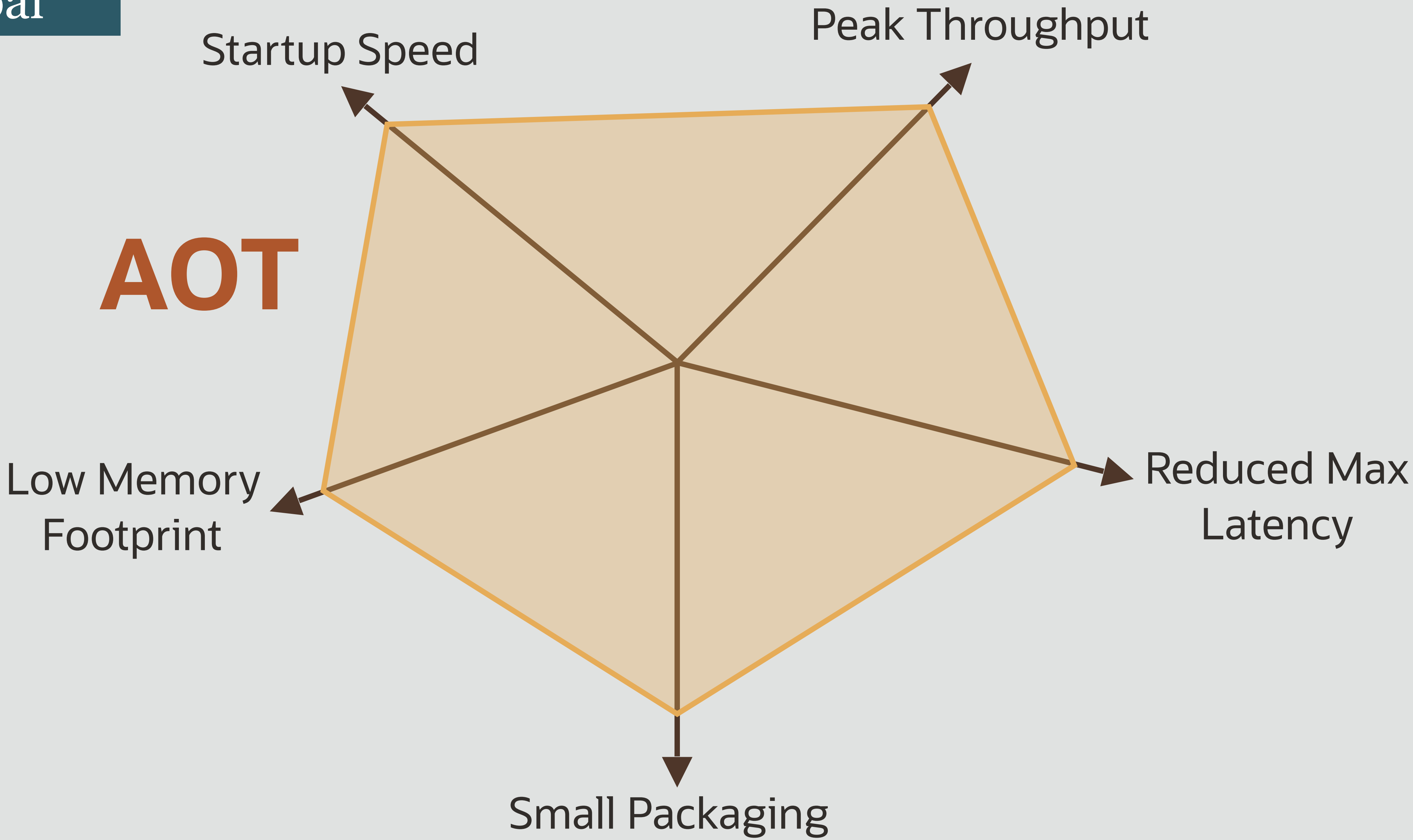
Microservice Frameworks: Memory Usage



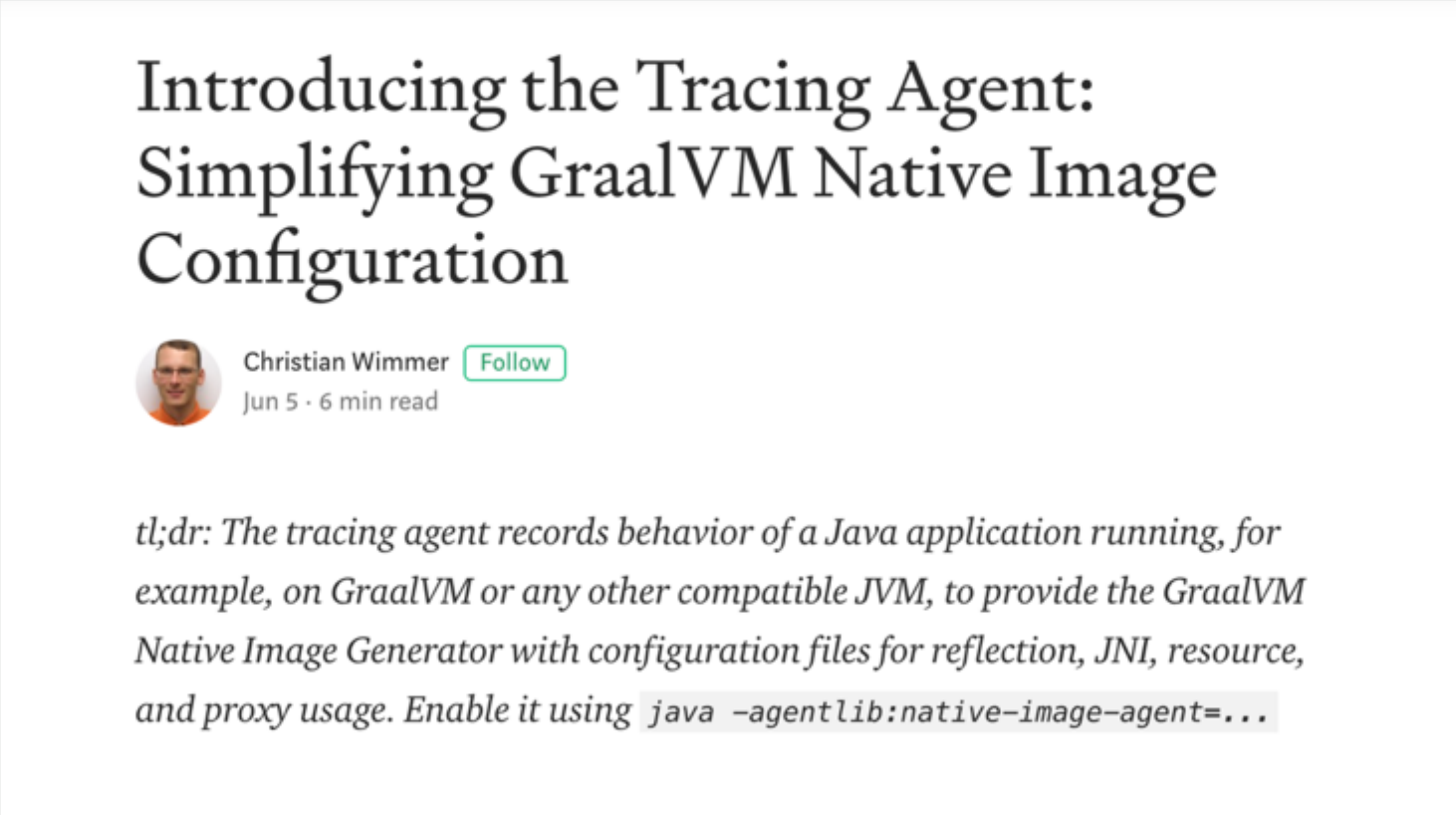
Currently


Startup Speed
Peak Throughput
AOT
Low Memory Footprint
Reduced Max Latency
Small Packaging

Goal


AOT
Startup Speed
Peak Throughput
Low Memory Footprint
Reduced Max Latency
Small Packaging

Simplifying the Native Image Configuration



Introducing the Tracing Agent: Simplifying GraalVM Native Image Configuration

Christian Wimmer Follow

Jun 5 · 6 min read

tl;dr: The tracing agent records behavior of a Java application running, for example, on GraalVM or any other compatible JVM, to provide the GraalVM Native Image Generator with configuration files for reflection, JNI, resource, and proxy usage. Enable it using `java -agentlib:native-image-agent=...`

Continue Learning About GraalVM Native Images

- Reference manual: graalvm.org/docs/reference-manual/aot-compilation/
- Improving performance of GraalVM native images with PGO: https://medium.com/graalvm/improving-performance-of-graalvm-native-images-with-profile-guided-optimizations-9c431a834edb
- GraalVM Native Images: The Best Startup Solution for Your Applications: https://www.youtube.com/watch?v=z0jedLjcWjI

Java Microservice Frameworks with GraalVM Native Image Support

https://micronaut.io

https://helidon.io

https://quarkus.io

(In progress) Spring Boot

How to achieve even more with native images: PGO

The GraalVM compiler is built ground-up with profiles in mind

Collecting profiles is essential for performance of native images

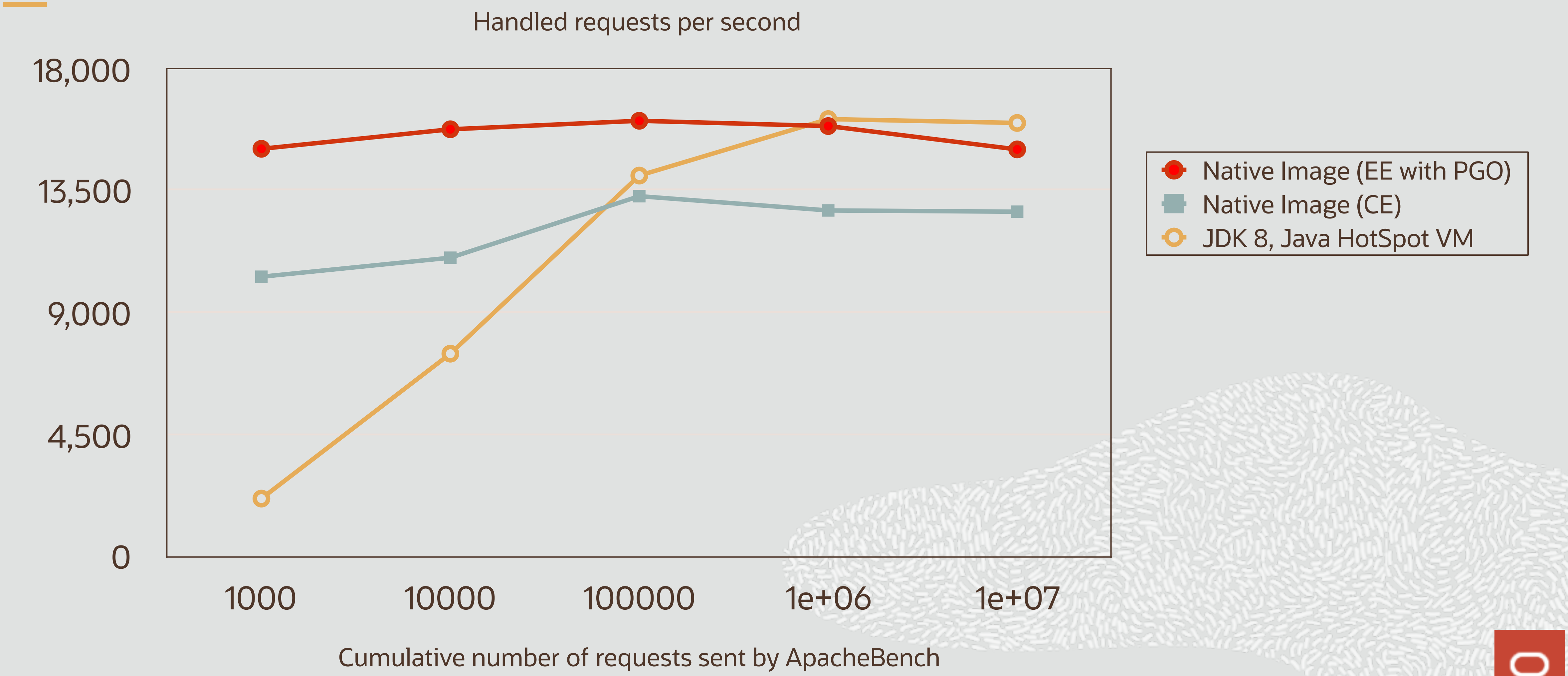
Profile guided optimizations requires running relevant workloads before building an image

```
$ java -Dgraal.PGOInstrument=myclass.iprof MyClass
```

```
$ native-image --pgo=myclass.iprof MyClass
```

```
$ ./myclass
```

Native Image: Profile-Guided Optimizations (PGO)



AOT vs JIT: Max Latency

JIT

- Many low latency GC options available
- G1
- CMS
- ZGC
- Shenandoah

AOT

- Only regular stop&copy collector
- Assumes small heap configuration
- Can quickly restart; could use load balancer instead of GC

Summary

GraalVM JIT

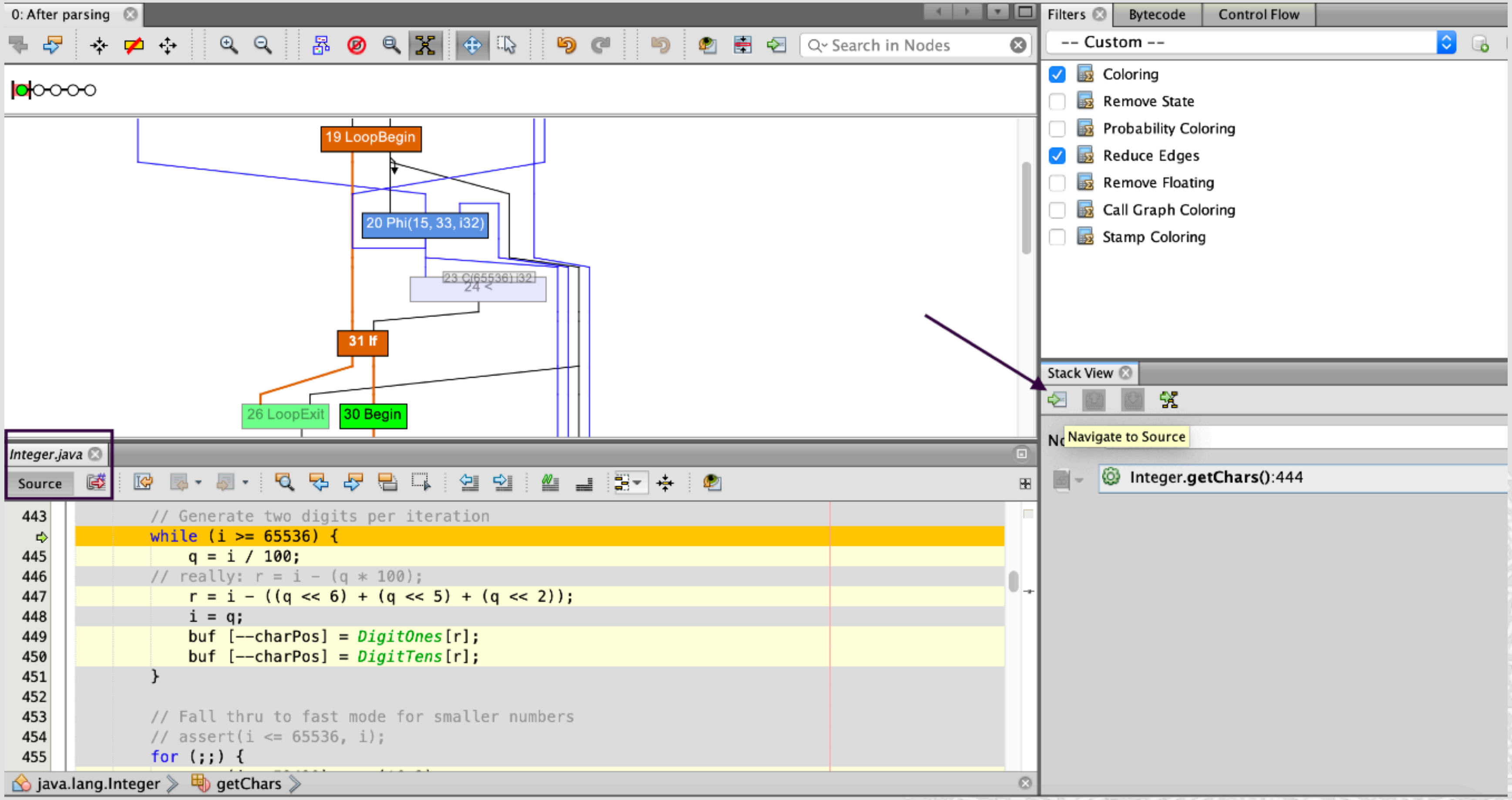
- Peak throughput
- Max Latency
- No configuration

GraalVM AOT

- Startup Time
- Memory footprint
- Packaging size

Tools

Ideal Graph Visualizer



-- Custom --

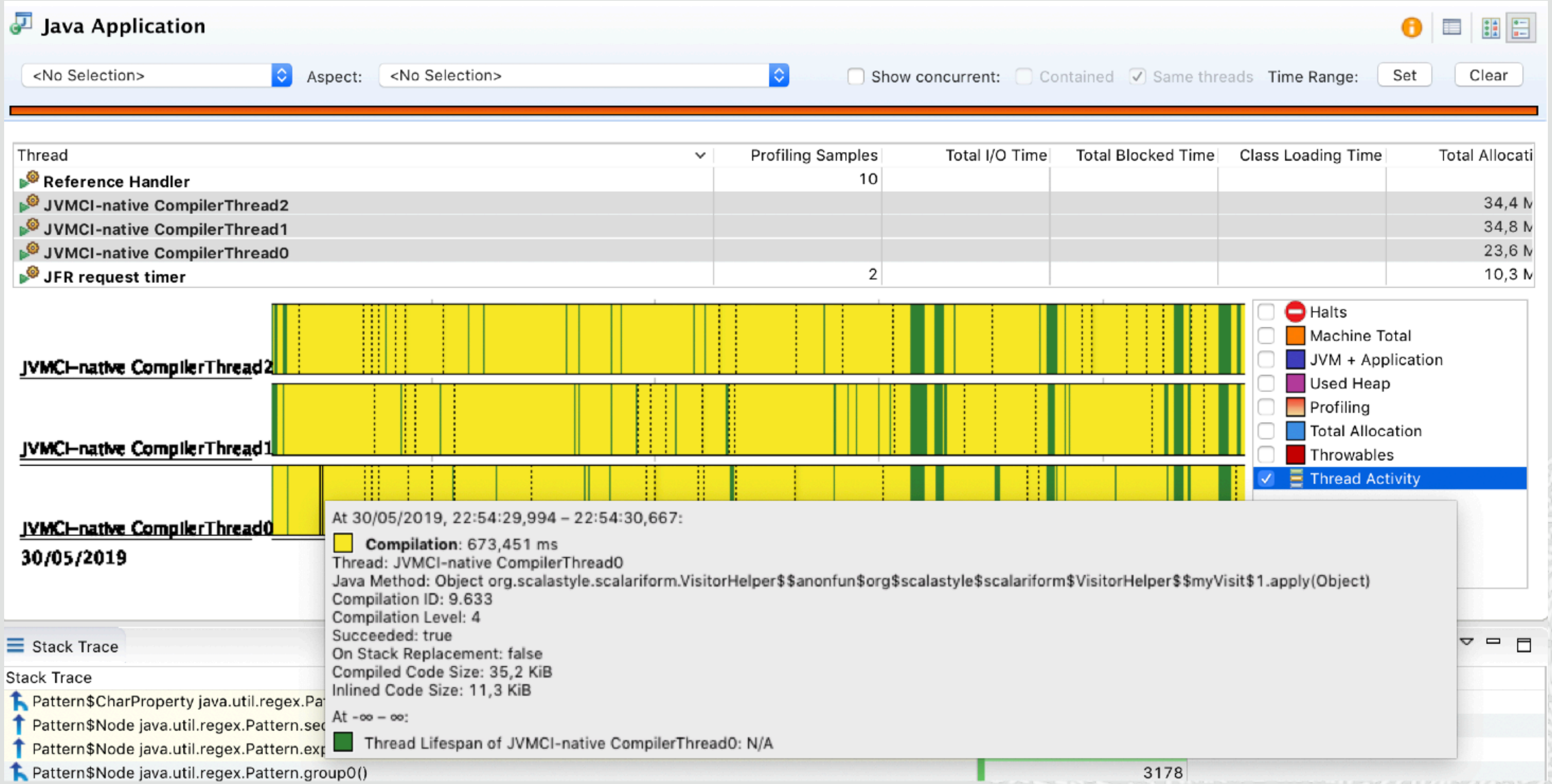
☑ Coloring
☐ Remove State
☐ Probability Coloring
☑ Reduce Edges
☐ Remove Floating
☐ Call Graph Coloring
☐ Stamp Coloring

```
        // Generate two digits per iteration
        while (i >= 65536) {
            q = i / 100;
        // really: r = i - (q * 100);
            r = i - ((q << 6) + (q << 5) + (q << 2));
            i = q;
            buf [--charPos] = DigitOnes[r];
            buf [--charPos] = DigitTens[r];
        }

        // Fall thru to fast mode for smaller numbers
        // assert(i <= 65536, i);
        for (;;) {
```

Integer.getChars():444

java.lang.Integer > getChars >

Java Flight Recorder Compilation Information



Java Application

<No Selection> Aspect: <No Selection> □ Show concurrent: □ Contained ☑ Same threads Time Range: Set Clear

Thread	Profiling Samples	Total I/O Time	Total Blocked Time	Class Loading Time	Total Allocati
Reference Handler	10				
JVMCI-native CompilerThread2					34,4 M
JVMCI-native CompilerThread1					34,8 M
JVMCI-native CompilerThread0					23,6 M
JFR request timer	2				10,3 M

JVMCI-native CompilerThread2
JVMCI-native CompilerThread1
JVMCI-native CompilerThread0
30/05/2019

- □ Halts
- □ Machine Total
- □ JVM + Application
- □ Used Heap
- □ Profiling
- □ Total Allocation
- □ Throwables
- ☑ Thread Activity

At 30/05/2019, 22:54:29,994 – 22:54:30,667:

🟨 **Compilation**: 673,451 ms
Thread: JVMCI-native CompilerThread0
Java Method: Object org.scalastyle.scalariform.VisitorHelper$$anonfunorgscalastyle$scalariform$VisitorHelper$$myVisit$1.apply(Object)
Compilation ID: 9.633
Compilation Level: 4
Succeeded: true
On Stack Replacement: false
Compiled Code Size: 35,2 KiB
Inlined Code Size: 11,3 KiB

At -∞ – ∞:

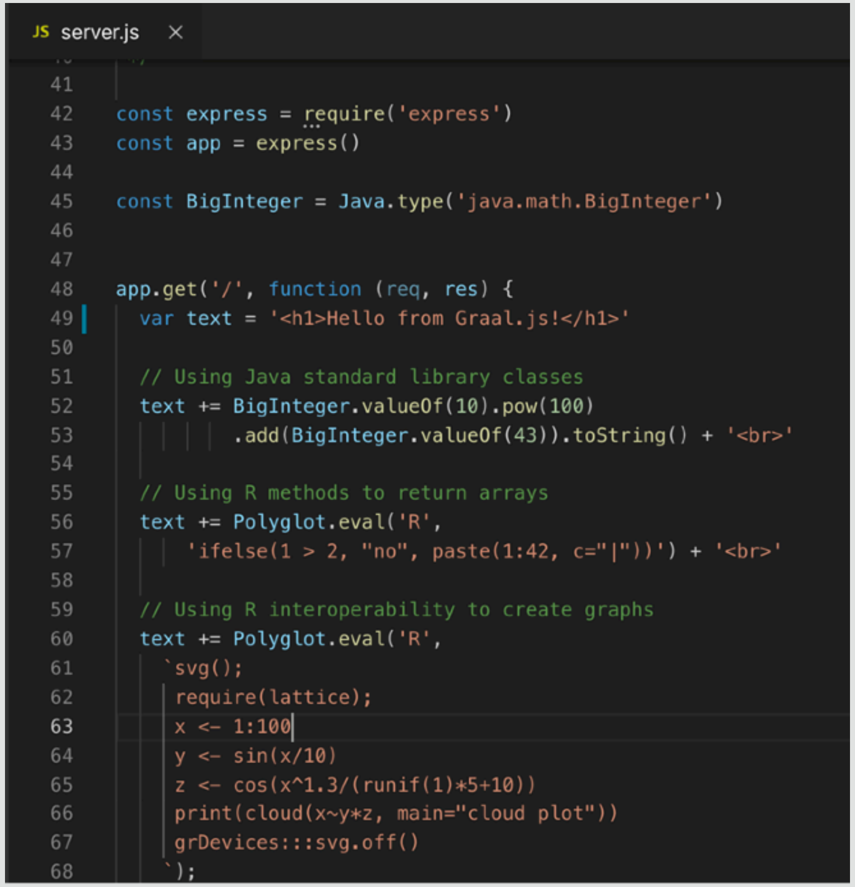
🟩 Thread Lifespan of JVMCI-native CompilerThread0: N/A

Stack Trace

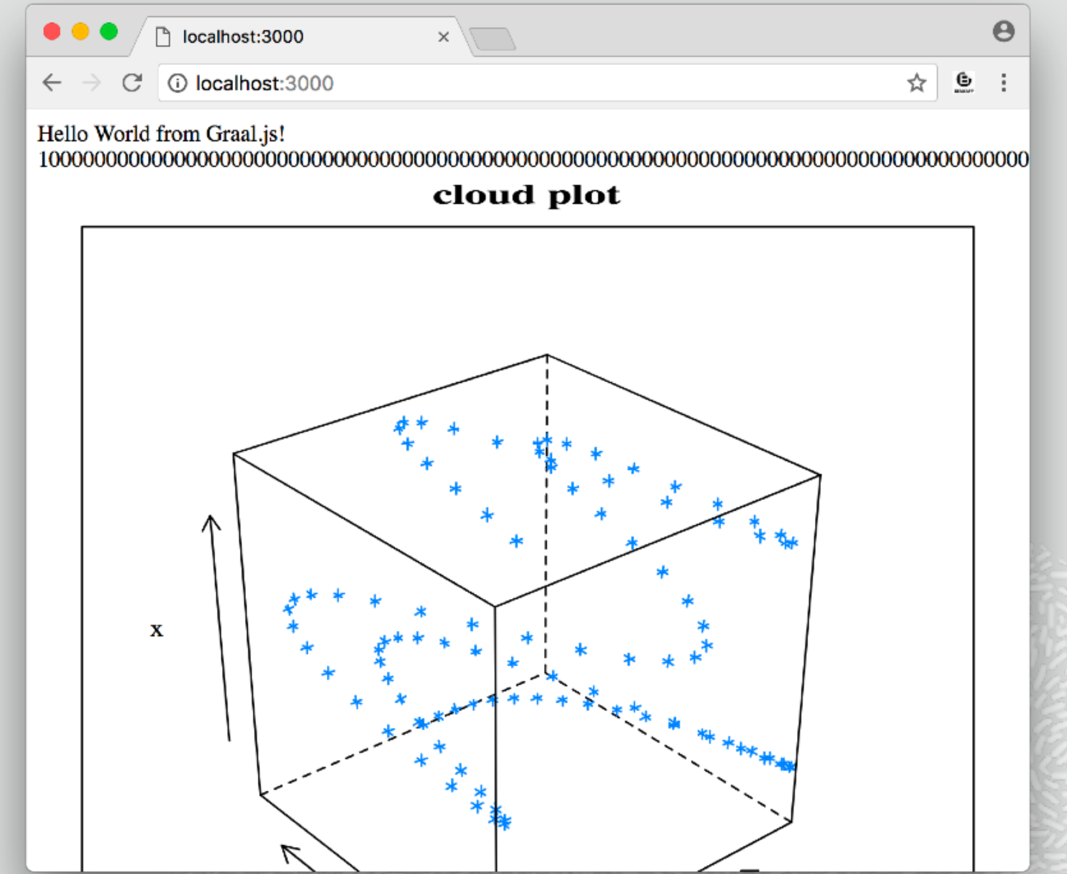
```
Stack Trace
Pattern$CharProperty java.util.regex.Pa
Pattern$Node java.util.regex.Pattern.se
Pattern$Node java.util.regex.Pattern.exp
Pattern$Node java.util.regex.Pattern.group0()                                3178
```

Do even more with GraalVM

JavaScript + Java + R



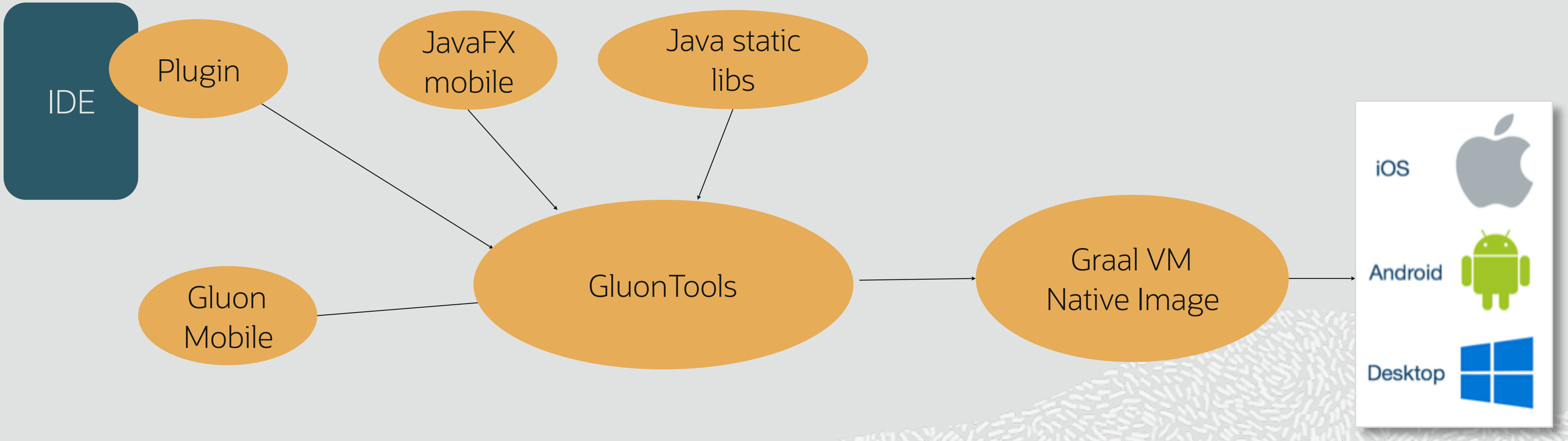
```
const express = require('express')
const app = express()

const BigInteger = Java.type('java.math.BigInteger')


app.get('/', function (req, res) {
  var text = '<h1>Hello from Graal.js!</h1>'

  // Using Java standard library classes
  text += BigInteger.valueOf(10).pow(100)
        .add(BigInteger.valueOf(43)).toString() + '<br>'

  // Using R methods to return arrays
  text += Polyglot.eval('R',
    'ifelse(1 > 2, "no", paste(1:42, c="|"))') + '<br>'

  // Using R interoperability to create graphs
  text += Polyglot.eval('R',
    `svg();
     require(lattice);
     x <- 1:100
     y <- sin(x/10)
     z <- cos(x^1.3/(runif(1)*5+10))
     print(cloud(x~y*z, main="cloud plot"))
     grDevices:::svg.off()
    `);
```



Hello World from Graal.js!
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000

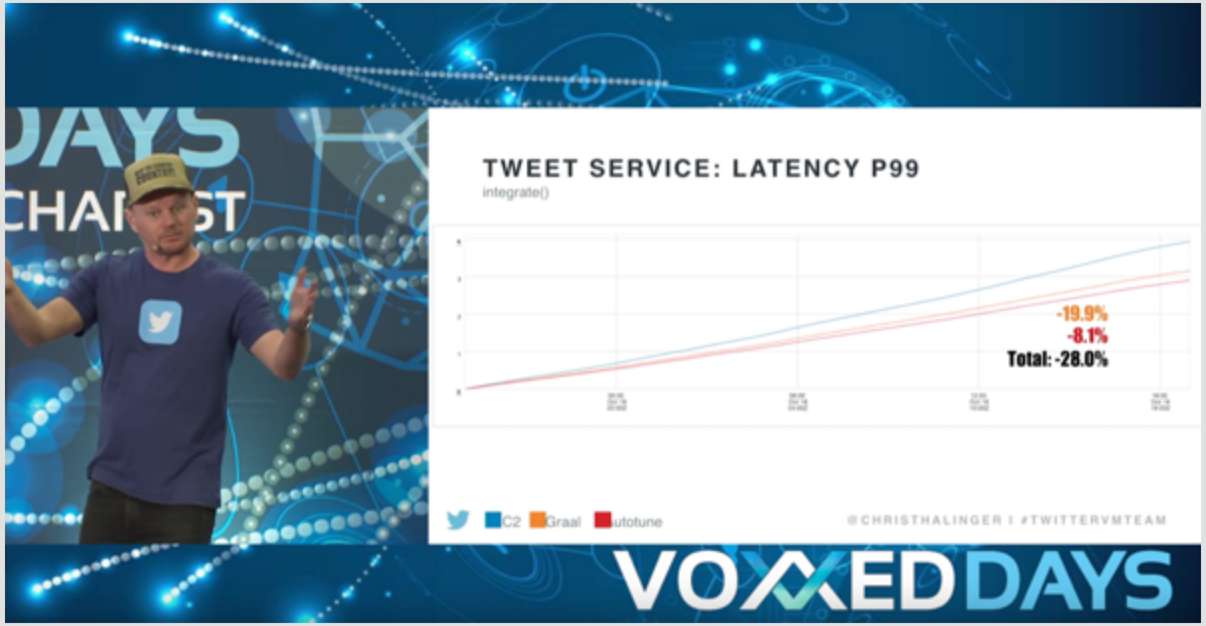
cloud plot

Do even more with GraalVM: Cross-Platform Development

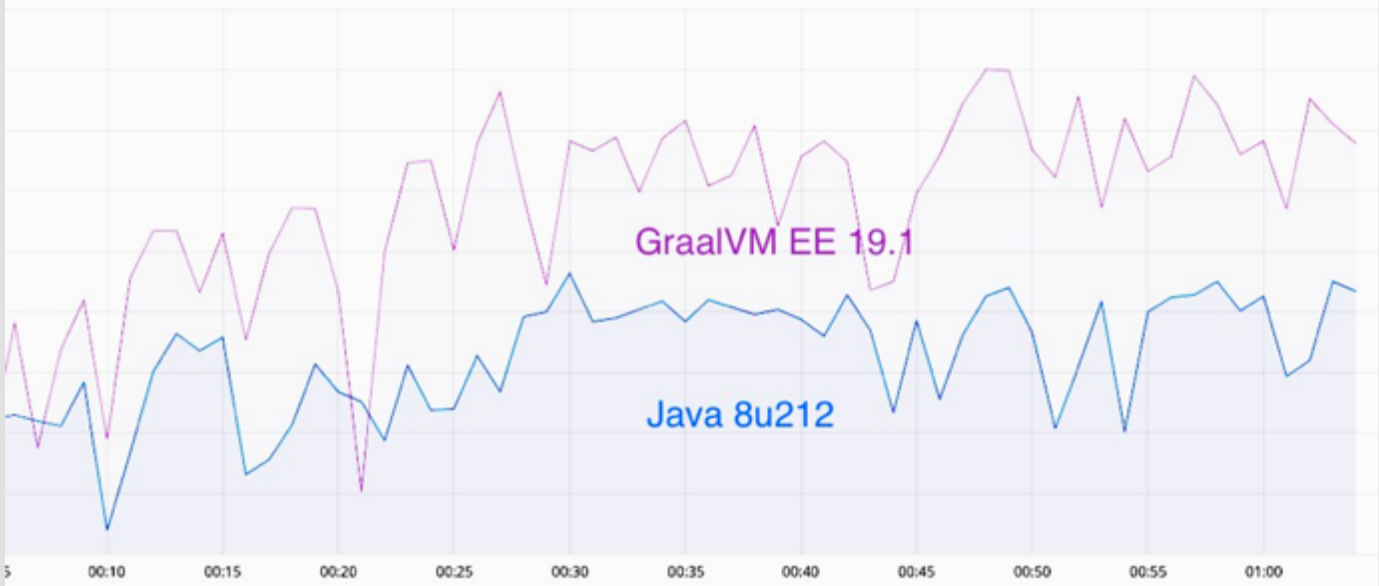


Industry Use Cases

Twitter uses GraalVM compiler in production to run their Scala microservices



- Peak performance: +10%
- Garbage collection time: -25%
- Seamless migration



ORACLE
Cloud Infrastructure

The rich ecosystem of CUDA-X libraries is now available for GraalVM applications.

GPU kernels can be directly launched from GraalVM languages such as R, JavaScript, Scala and other JVM-based languages.

NVIDIA®

What’s next for GraalVM

Recent Updates

- Updated profile-guided optimizations for native images;
- Support for JFR in Graal VisualVM;
- Throughput improvements in native images;
- LLVM toolchain;
- VS Code plugin preview;
- Class Initialization changes in native images.

What’s next for GraalVM

- JDK-11 based builds;
- ARM64 and Windows support;
- Low-latency, high-throughput, and parallel GC for native images;
- Work with the community to support important libraries;
- New languages and platforms;
- Your choice – contribute!

What’s next for you

- Download:

  graalvm.org/downloads

- Follow updates:

  @GraalVM / #GraalVM

- If you need help:
- gitter.im/graalvm
- graalvm-users @oss.oracle.com

Thank you!

Alina Yurenko / @alina_yurenko

GraalVM Developer Advocate
Oracle Labs