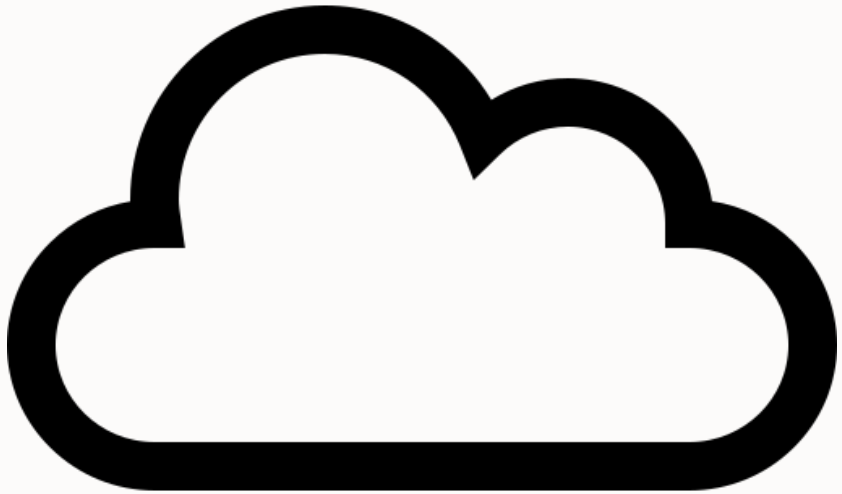
ORACLE

# Revolutionizing Java-Based Cloud Deployments with GraalVM

**David Leopoldseder**
GraalVM Compiler Developer
Principal Researcher
Oracle Labs
https://www.davidleopoldseder.com/

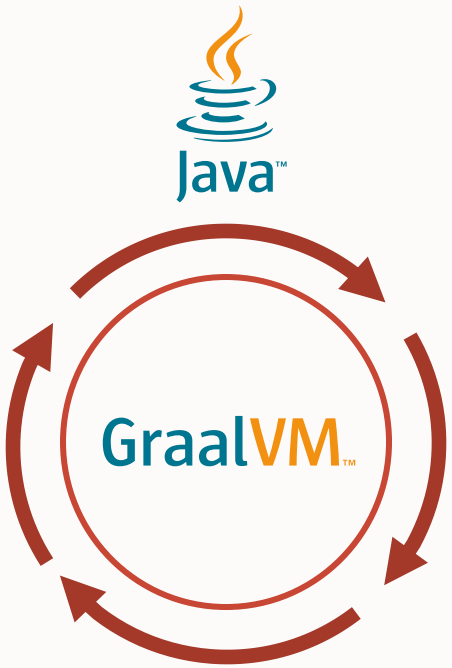




Native Image

# Raise your hands if you are familiar with GraalVM ?

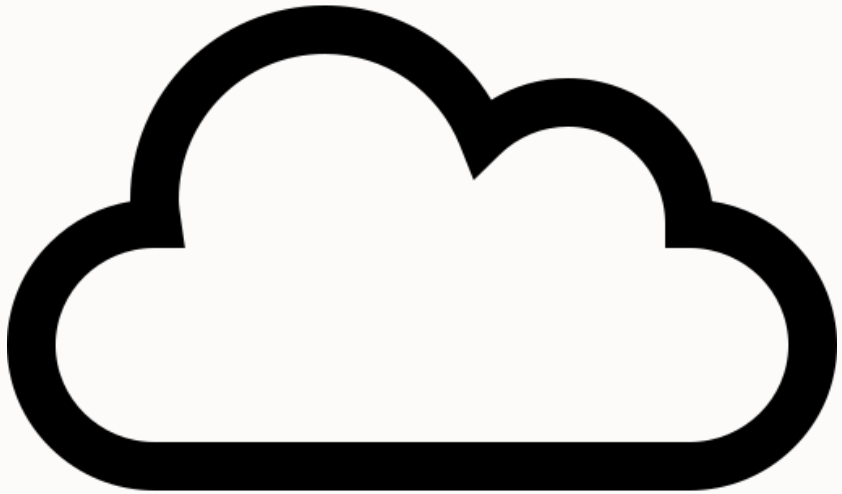
# GraalVM Magic in one tweet...




High-performance optimizing Just-in-Time (JIT) compiler


Multi-language support for the JVM


Ahead-of-Time (AOT) “Native Image” generator





Native Image


GraalVM™

## JIT

java MyMainClass

## AOT

native-image MyMainClass

./mymainclass

# Java in the Cloud - Goals

Start Fast

Low Resource Usage

Minimize Vulnerability

Compact Packaging

# Java in the Cloud - Goals

**Start Fast**



Low Resource Usage

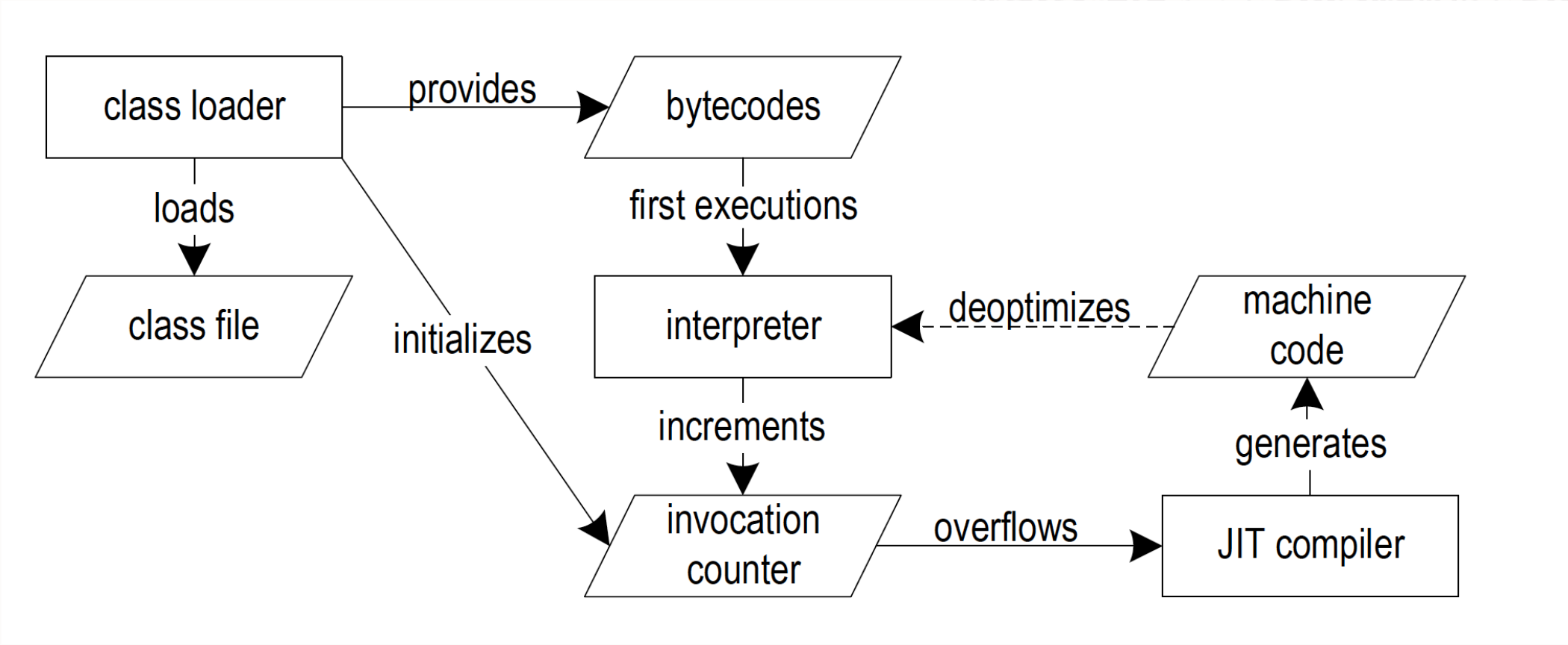


Minimize Vulnerability

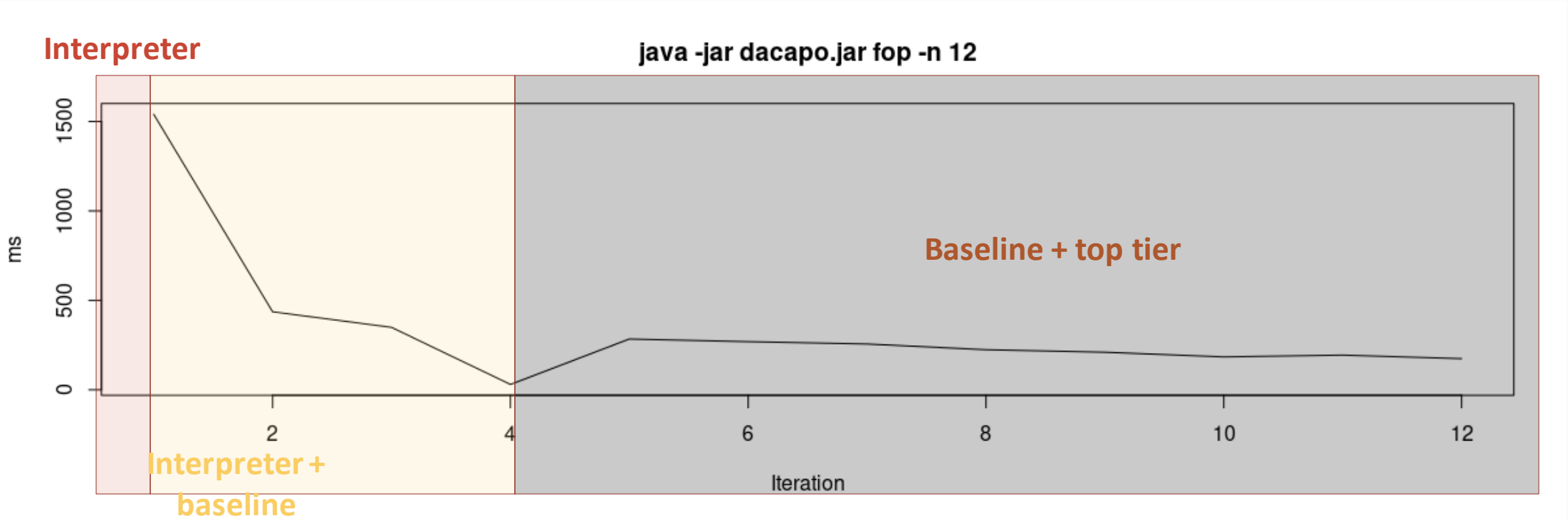


Compact Packaging

# Detour– Dynamic Compilation



# Excursus – Tiered Compilation



# JIT

Load JAR files from disk

⬇

Uncompress class files

⬇

Verify class definitions

⬇

Execute in interpreter (~20x slower)

⬇

Gather profiling feedback

⬇

Compile to machine code

⬇

Execute at peak performance

# AOT

Load executable from disk

⬇

Execute at peak performance



# Nice theory, but does it work in practice?

**Cédric Champeau**
@CedricChampeau

🤯 #Micronaut

```
> Task :nativeRun

 __  __ _                                  _
|  \/  (_) ___ _ __ ___  _ __   __ _ _   _| |_
| |\/| | |/ __| '__/ _ \| '_ \ / _` | | | | __|
| |  | | | (__| | | (_) | | | | (_| | |_| | |_
|_|  |_|_|\___|_|  \___/|_| |_|\__,_|\__,_|\__|
  Micronaut (v3.0.2-SNAPSHOT)

14:54:13.311 [main] INFO  io.micronaut.runtime.Micronaut - Startup completed in 6ms. Server Running: http://localhost:8080
<============--> 85% EXECUTING [3s]
```

# Java in the Cloud - Goals



Start Fast

**Low Resource Usage**



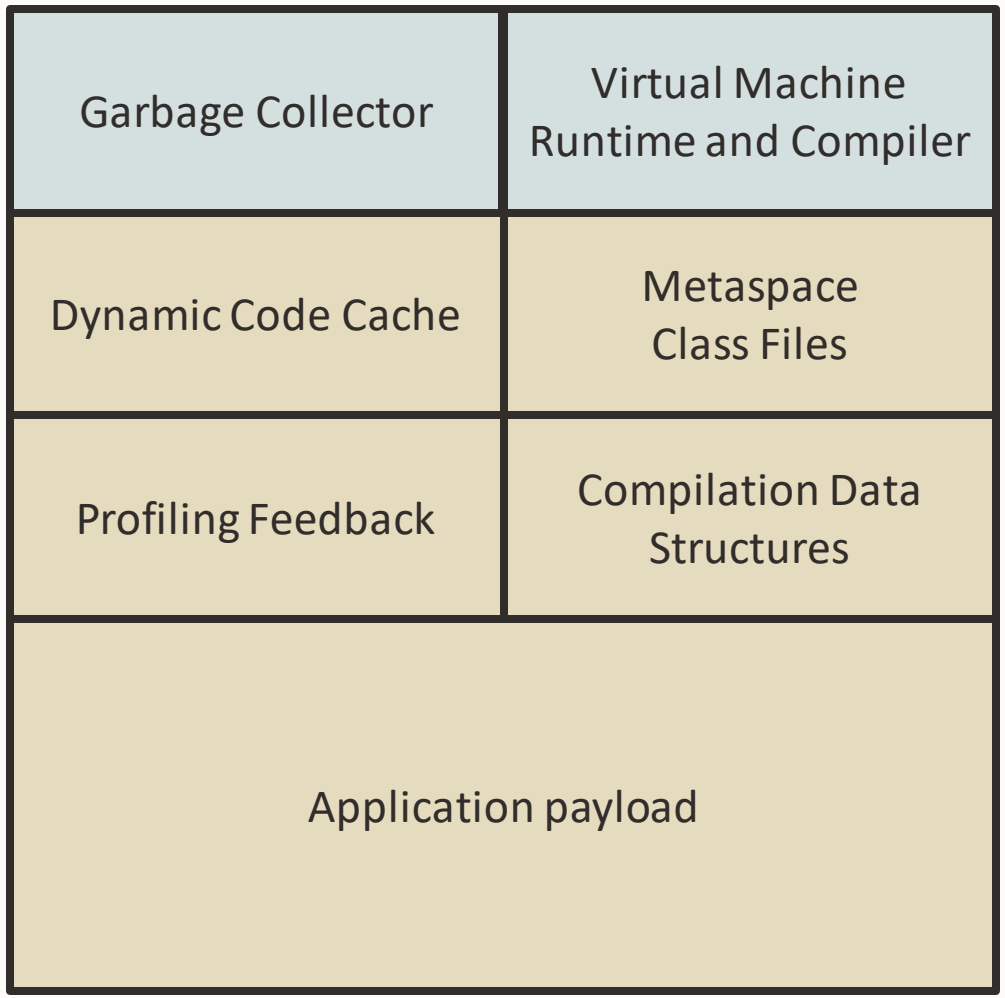
Minimize Vulnerability



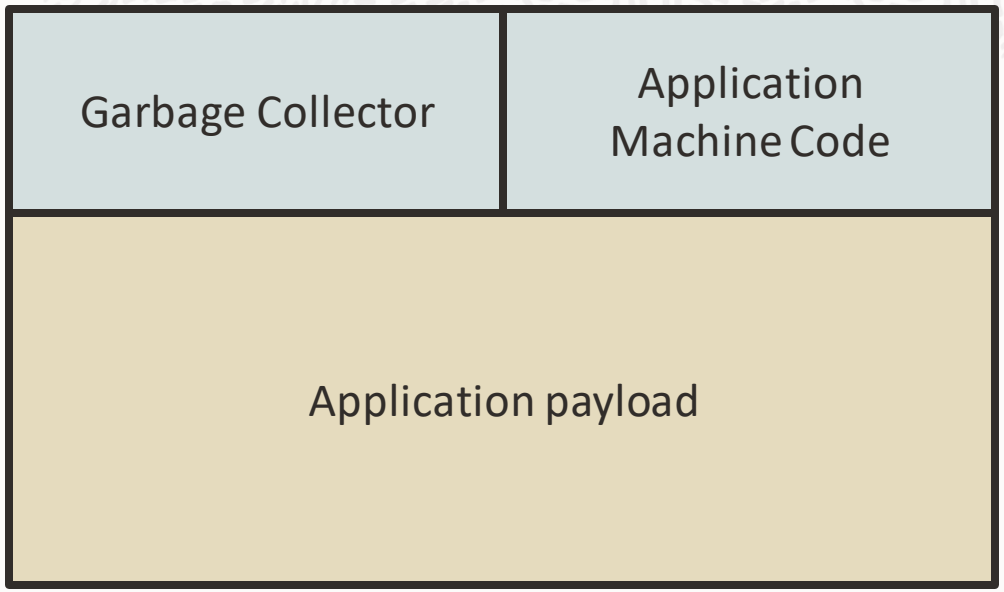
Compact Packaging

# Memory

## JIT

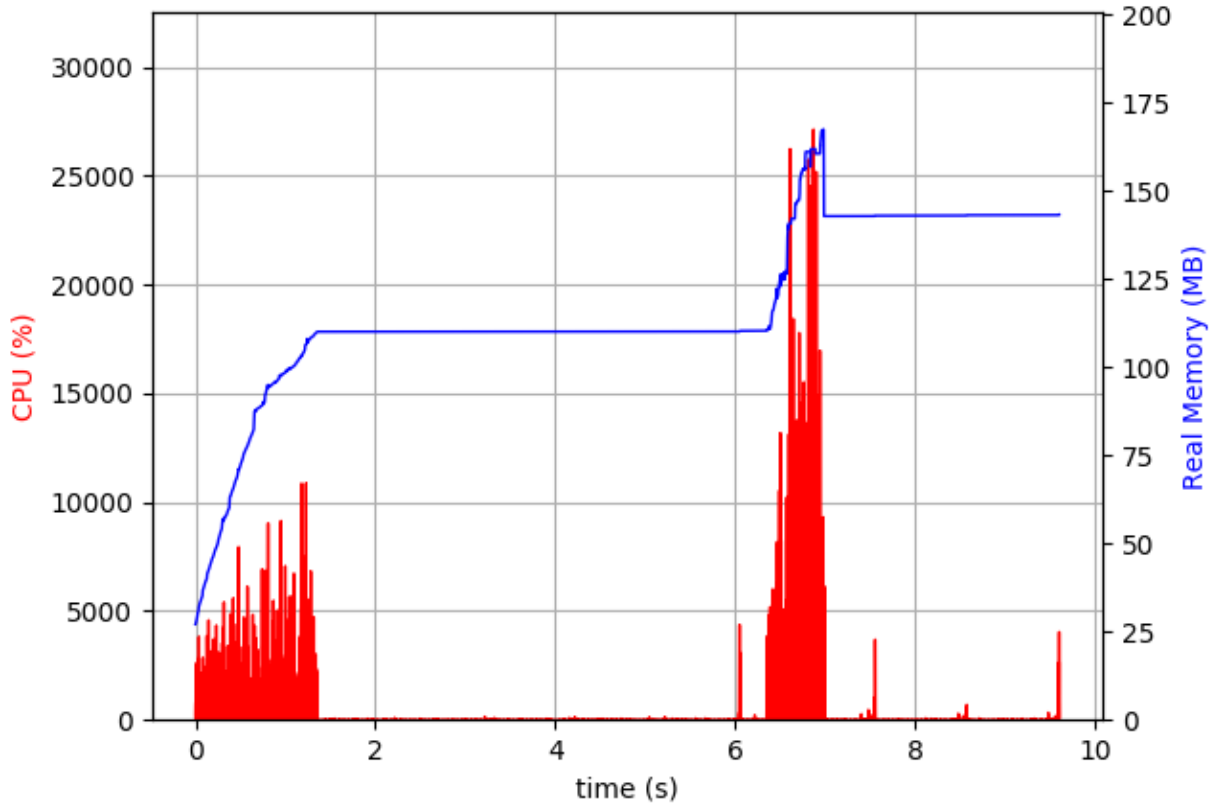



## AOT

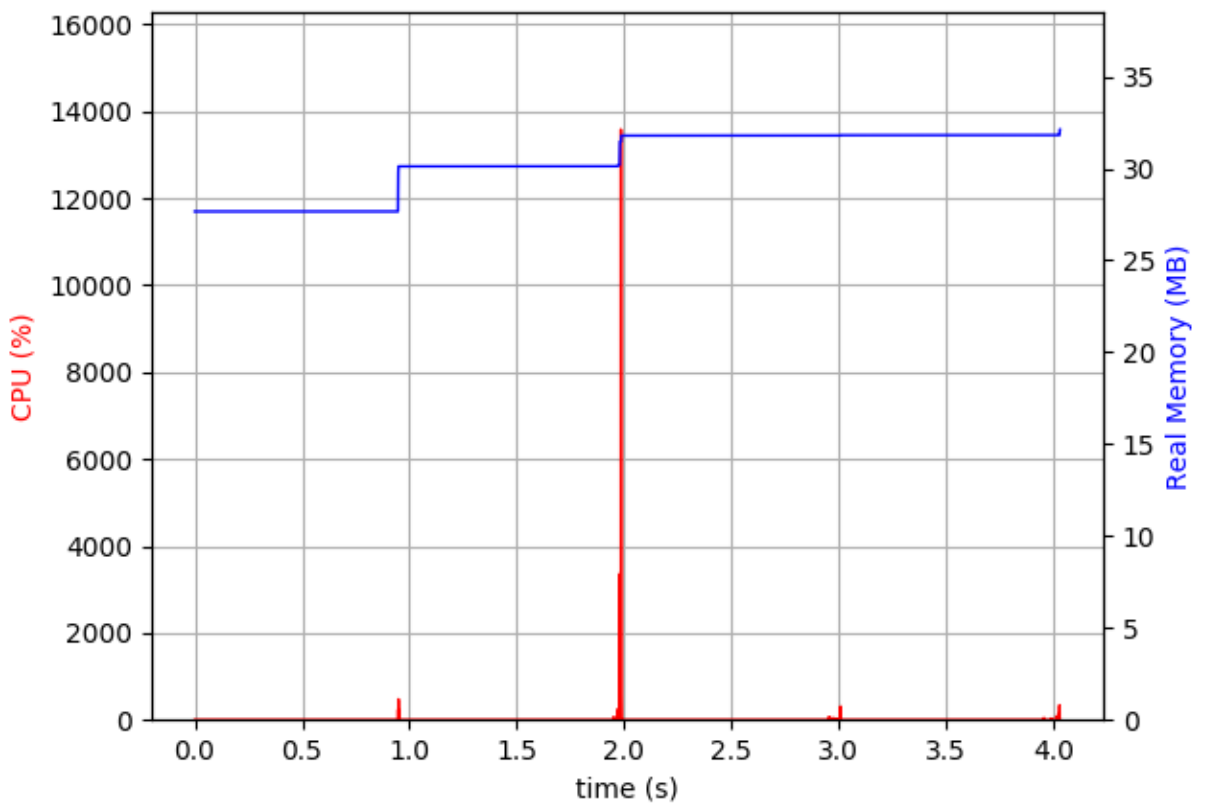


# Example: CPU and memory usage of web service

## JIT

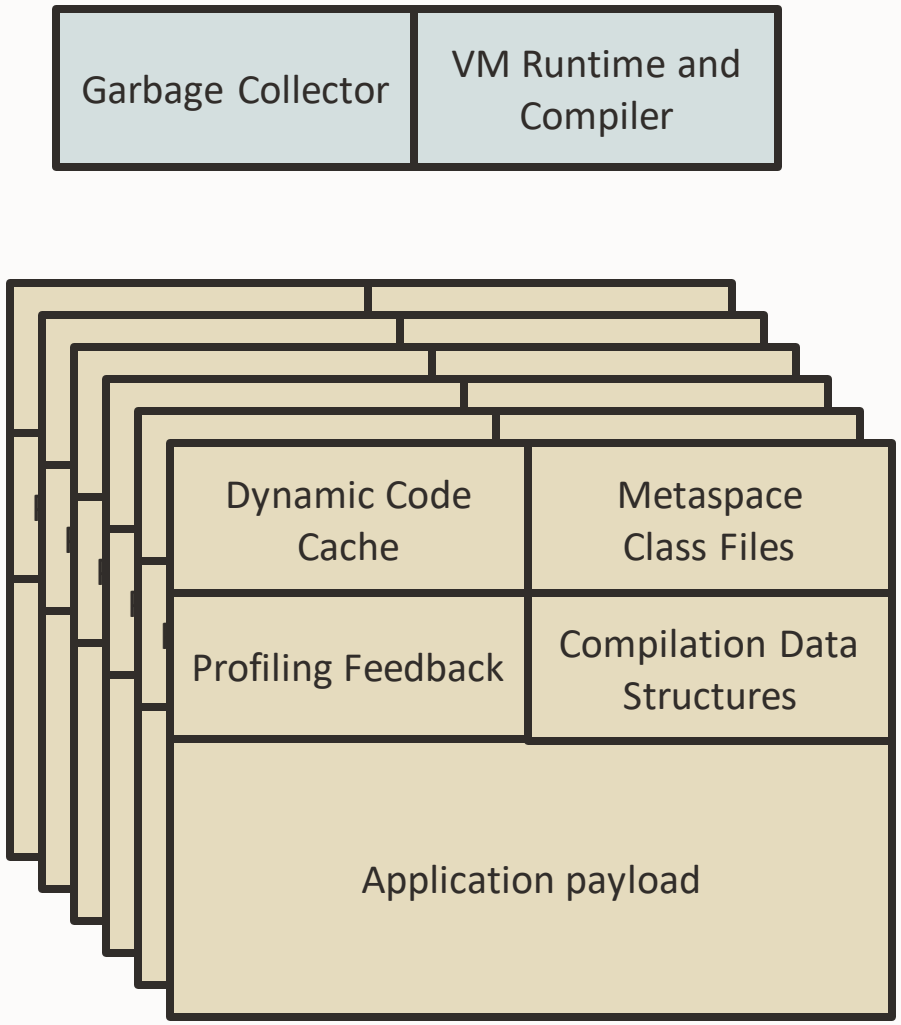



## AOT



# Memory Scalability

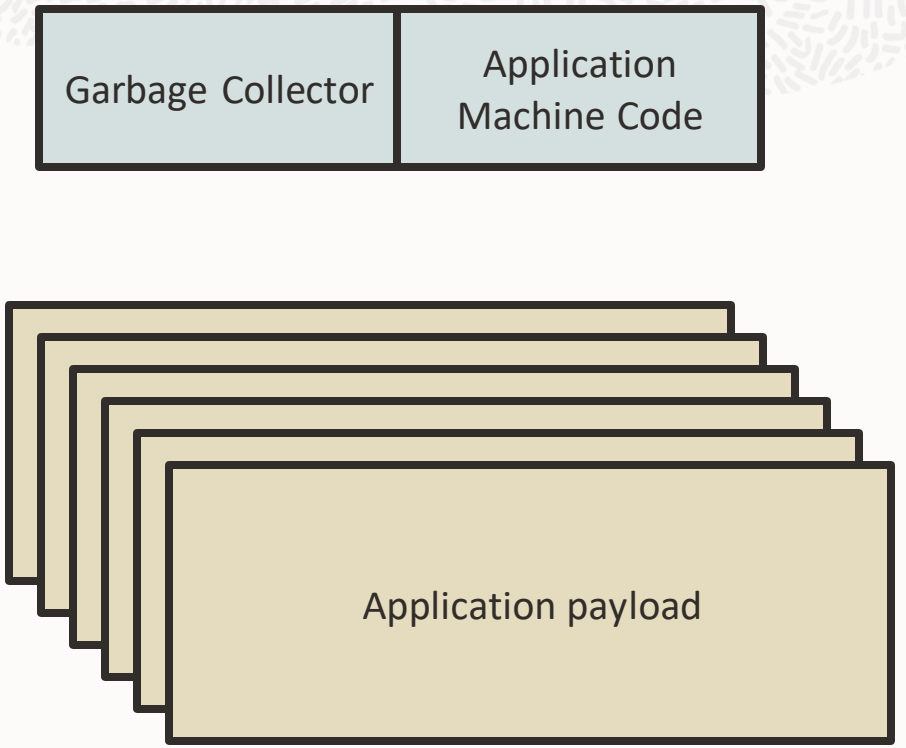
## JIT




**shared**

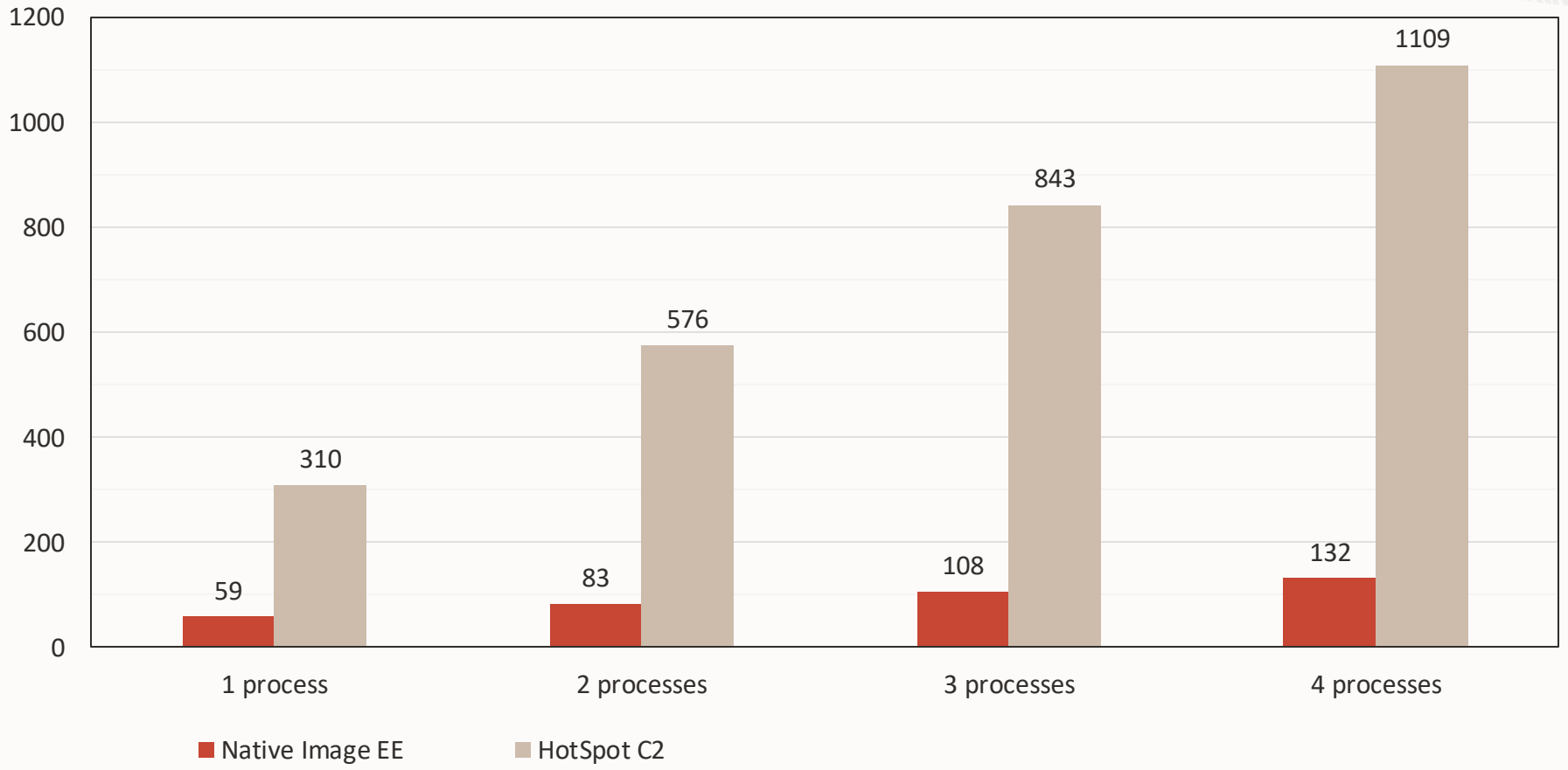
**duplicated per process**

## AOT



# Example: horizontal scaling of microservices

**Memory Usage in MByte**




Quarkus Apache Tika ODT in a “tiny” configuration and with the serial GC (1 CPU core per process, -Xms32m -Xmx128m) – JDK 11

# Java in the Cloud - Goals

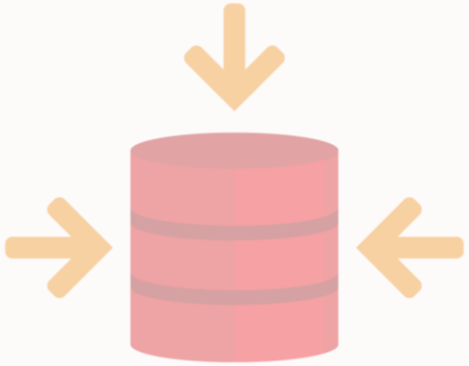


Start Fast



Low Resource Usage

**Minimize Vulnerability**



Compact Packaging

# What constitutes a Java program ?

- A set of types with
  - Fields
  - Methods
  - Native implemented methods
- A set of configuration files
- A start method (main method)
- A class-path
- A set of command line options ?
- A set of constants ?

**Basically code & data**

# What constitutes a Java program ?

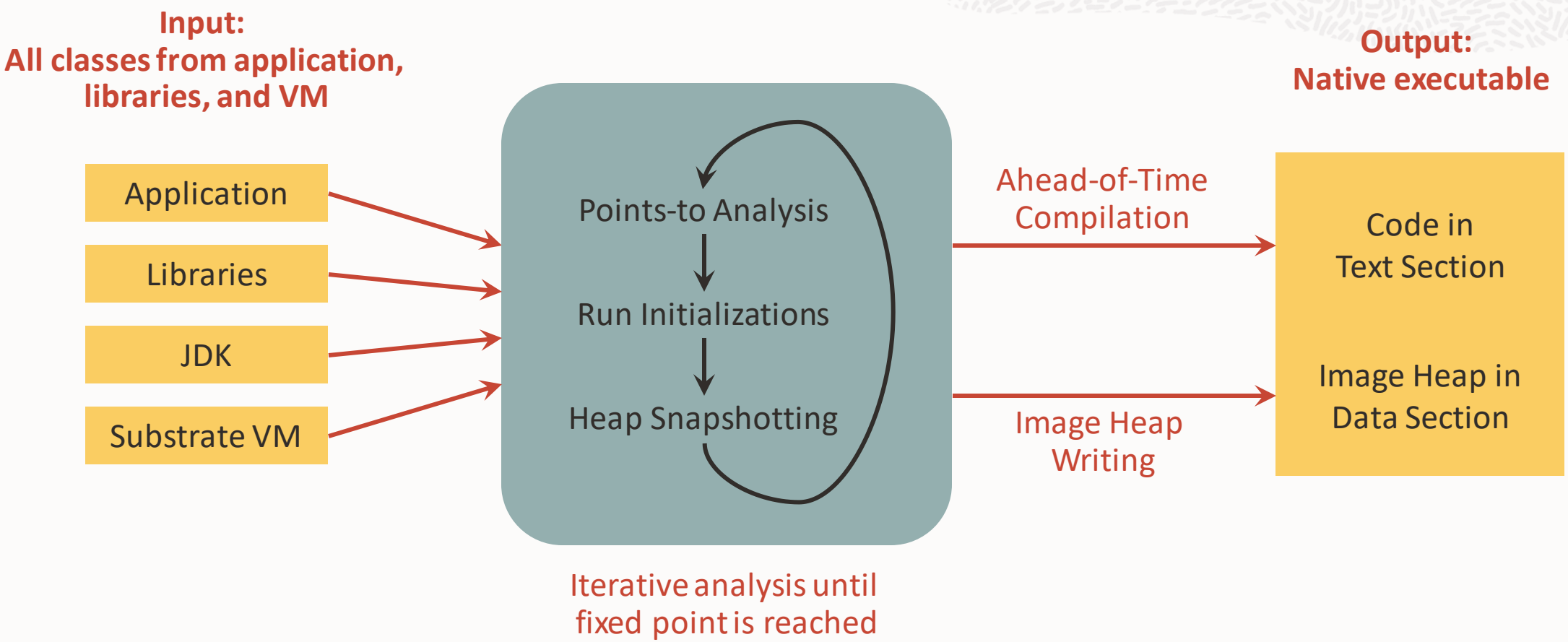
- A set of types with
  - Fields
  - Methods
  - Native implemented methods
- A set of configuration files
- A start method (main method)
- A class-path
- A set of command line options ?
- A set of constants ?

**Basically code & data**

- Runtime System
- Native libraries
- JDK Code
- Garbage Collector

**The “REST”**

# Detour – Native Image



# Native Image – Closed World Assumption

1 Entry Point

```
public class ClosedWorld {
 static class A {
  public void foo(){
   Date date = Calendar.getInstance().getTime();
   DateFormat dateFormat = new SimpleDateFormat("yyyy-mm-dd hh:mm:ss");
   String strDate = dateFormat.format(date);
   System.out.println("Hello world it is: " + strDate);
  }
 }
 public static void main(String[] args) {
  new A().foo();
 }
}
```

2 A used

3 Used

- Date (+ super classes)
- Calendar (+ super classes)
- DateFormat
- String
- System
- PrintStream
- ...[Date]

# Reduced Attack Surface



- No new unknown code can be loaded at run time
- Only paths proven reachable by the application are included in the image
- Reflection is disabled by default and needs an explicit include list
- Deserialization only enabled for specified list of classes
- Just-in-time compiler crashes, wrong compilations, or “JIT spraying” to create machine code gadgets are impossible

# Java in the Cloud - Goals



Start Fast



Low Resource Usage



Minimize Vulnerability

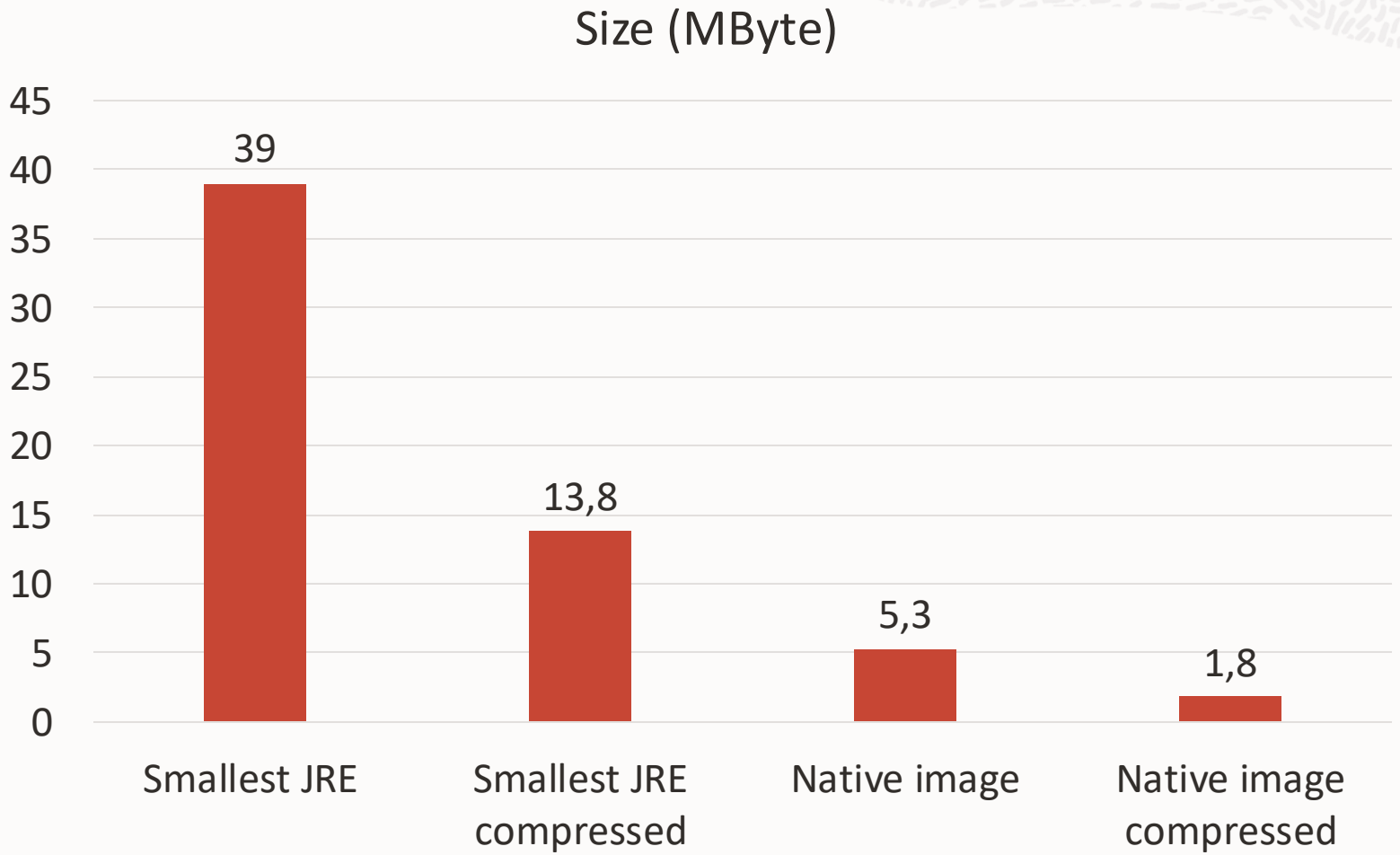
**Compact Packaging**

# Single Executable Native Image



- All relevant JVM runtime and JDK library code is included
- Unreachable paths (i.e., dead code) in the application and its dependencies eliminated
- Disadvantage that Java runtime installation cannot be shared, but also advantage that applications can be patched/updated individually

# Example: List directory command line utility




GraalVM EE 22.1, MacOS, JDK 11, native image compressed with upx

# Java in the Cloud with GraalVM™ Native Image

Start Fast

Low Resource Usage

Minimize Vulnerability

Compact Packaging

# What’s the catch?

# Reachability Meta-Data

- Need to assist static analysis with reachability information
- No unknown dynamic class loading allowed
- Use a framework with support for native image

https://helidon.io

https://quarkus.io

Experimental Spring support
https://github.com/spring-projects-experimental/spring-graal-native

# Required Build Time Step

- Computational effort necessary at build time
- Need a powerful machine with the same target architecture & OS
- Use it with GitHub Actions: github.com/marketplace/actions/github-action-for-graalvm
- Develop in JIT mode for fast development, only use AOT for final deployment
  - Or use Quick Build mode https://www.infoworld.com/article/3658988/graalvm-speeds-up-native-image-builds.html
    - Dev Only
    - Little Optimization
    - Reduced peak
- Performance

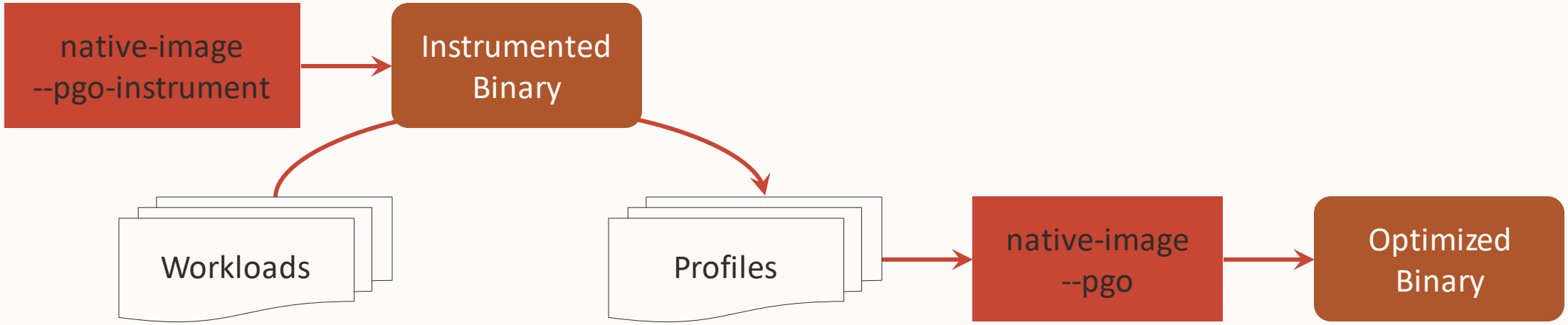
# AOT vs JIT Performance Considerations

- JIT
  - Run once code not compiled
  - Code size not important
  - Profiles & Deoptimization for high performance
  - Large deployment footprint
  - Slow Warmup
  - High-Performance for dynamic applications
- AOT
  - Everything is compiled
  - Code Size Relevant (image size)
  - One try to reach performance
  - Small image sizes (only necessary code)
  - Fast Warmup (no warmup ☺ )
  - Harder to optimize for dynamic use cases



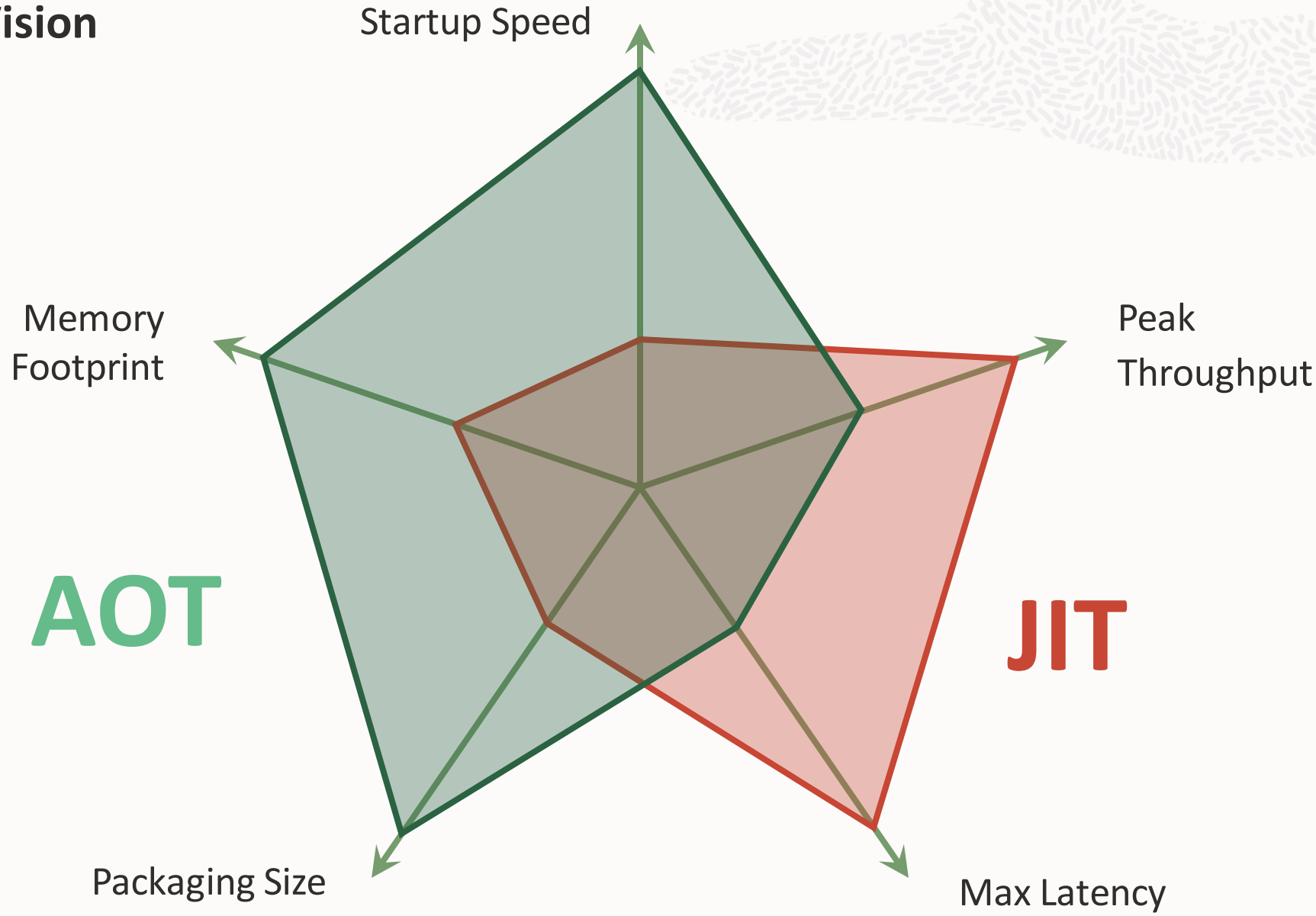
# PGO – Profile-Guided Optimizations

- AOT compiled code cannot optimize itself at run time
  - No dynamic “hot spot” compilation
- PGO requires relevant workloads at build time
- Optimized code runs immediately at startup, no “warmup” curve



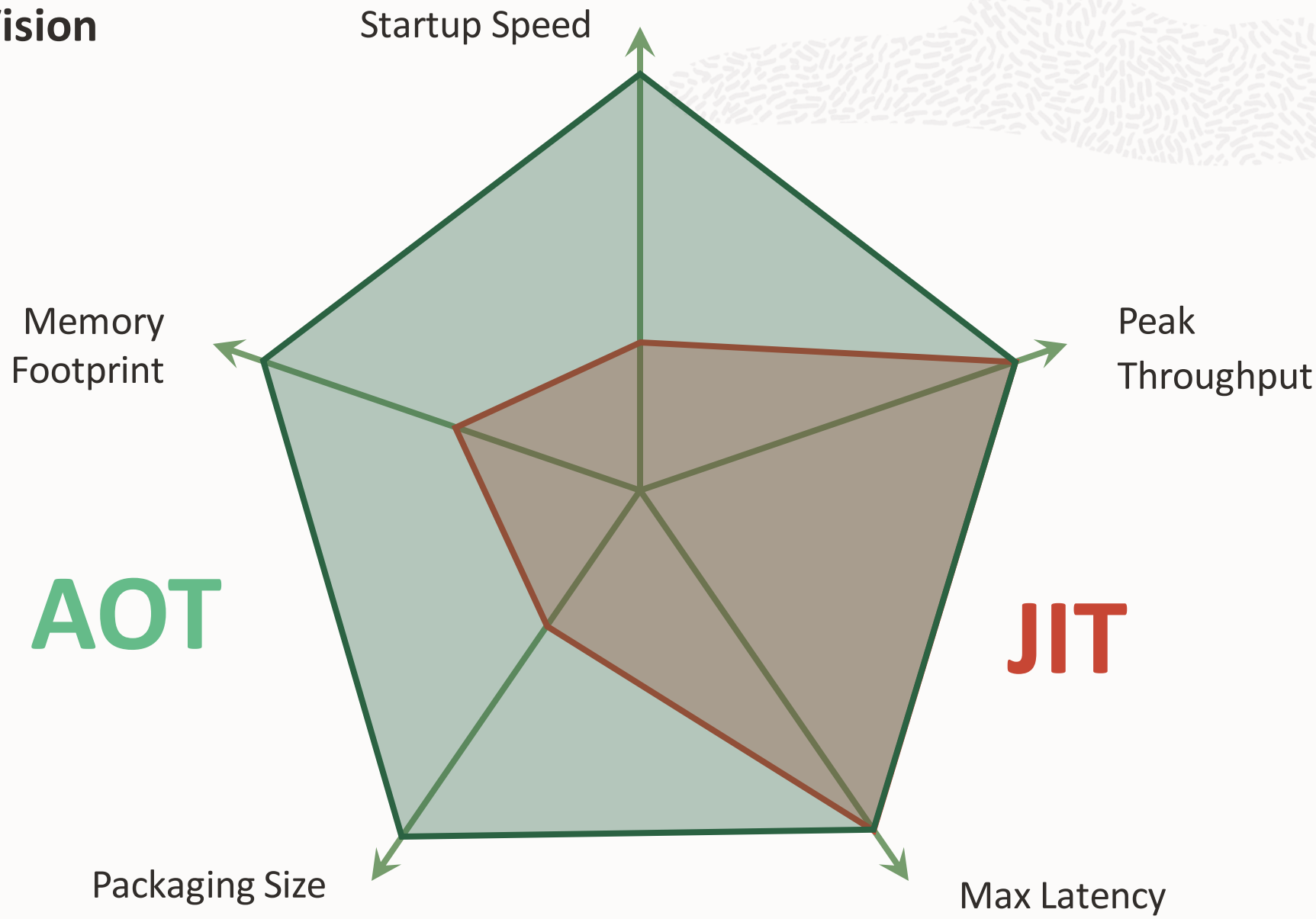
## Performance Vision

- Currently



## Performance Vision

- Goal



# Open source on GitHub

https://github.com/oracle/graal

oracle / graal Public

GraalVM: Run Programs Faster Anywhere 🚀

www.graalvm.org

View license

16.9k stars 1.4k forks

Contributors 242

+ 231 contributors

Languages

Java 90.8% C 4.4%

Python 1.6% WebAssembly 1.4%

C++ 0.5% JavaScript 0.4%

Other 0.9%

~ 4 million open source LOC maintained by the GraalVM team at Oracle!

# Get started with GraalVM!

**Download, Docs & Community: graalvm.org**


**or get started on Oracle Cloud: oracle.com/cloud**

bit.ly/free-resources-jax22

# Thank you!

**David Leopoldseder**

https://www.davidleopoldseder.com/