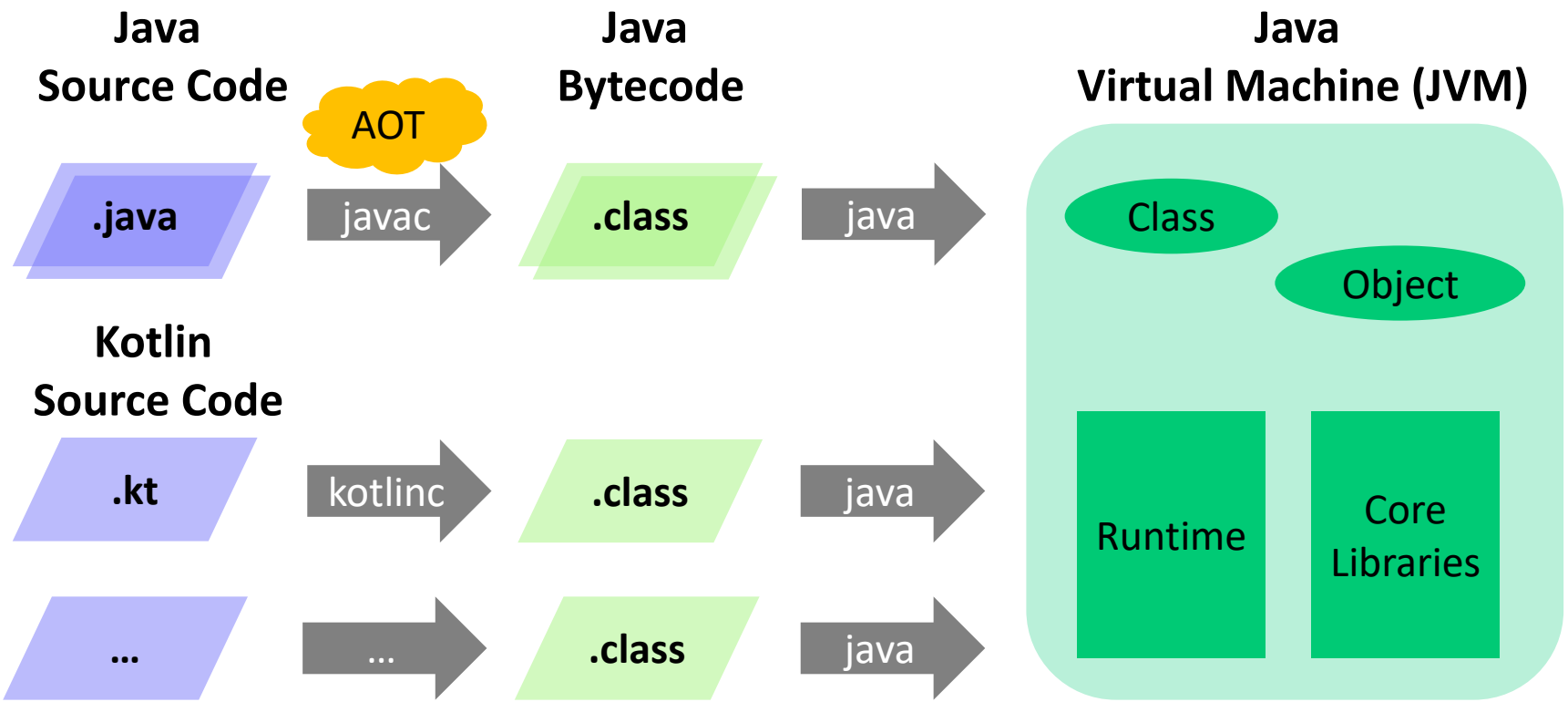
# GraalVM™ and Native Images

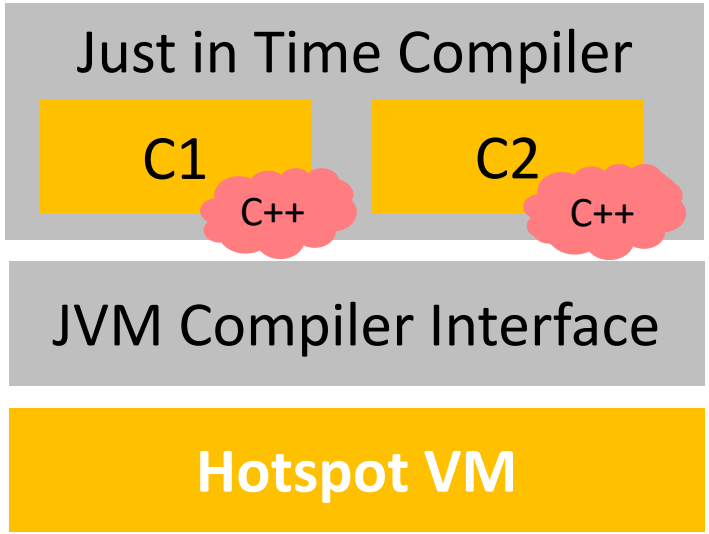
## AHEAD OF TIME COMPILATION FOR JAVA

ACCSO
ACCELERATED SOLUTIONS

# Java is a Language as well as an Execution Environment



# Just in Time Compiler: Runtime-Compilation of Bytecode to Machine Code



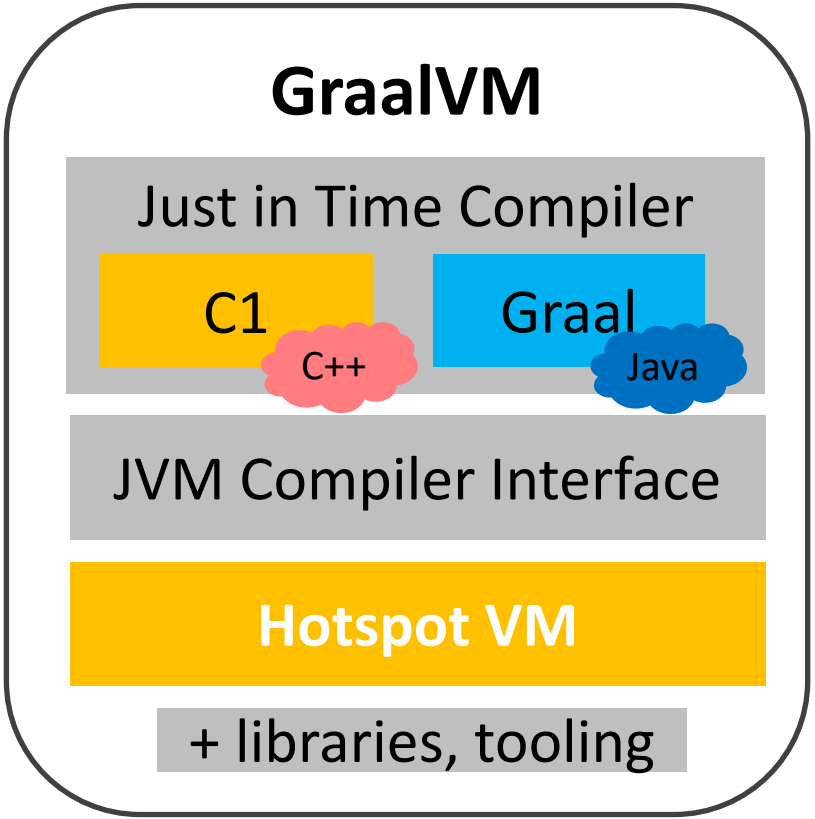

**C1**

- Fast, lightweight
- Simple optimizations, e.g. Inlining

**C2**

- Slower, resource intensive
- Highly optimizing, e.g. loop unrolling, array range check elimination, …

# Just in Time Compiler: Runtime-Compilation of Bytecode to Machine Code




**Graal**

- Modern JIT-Compiler written in Java
- Optimizations for Streams, Lambda
- Advanced vectorization support
- Polyglott support

# Drawbacks of Virtual Machines and JIT Compilation

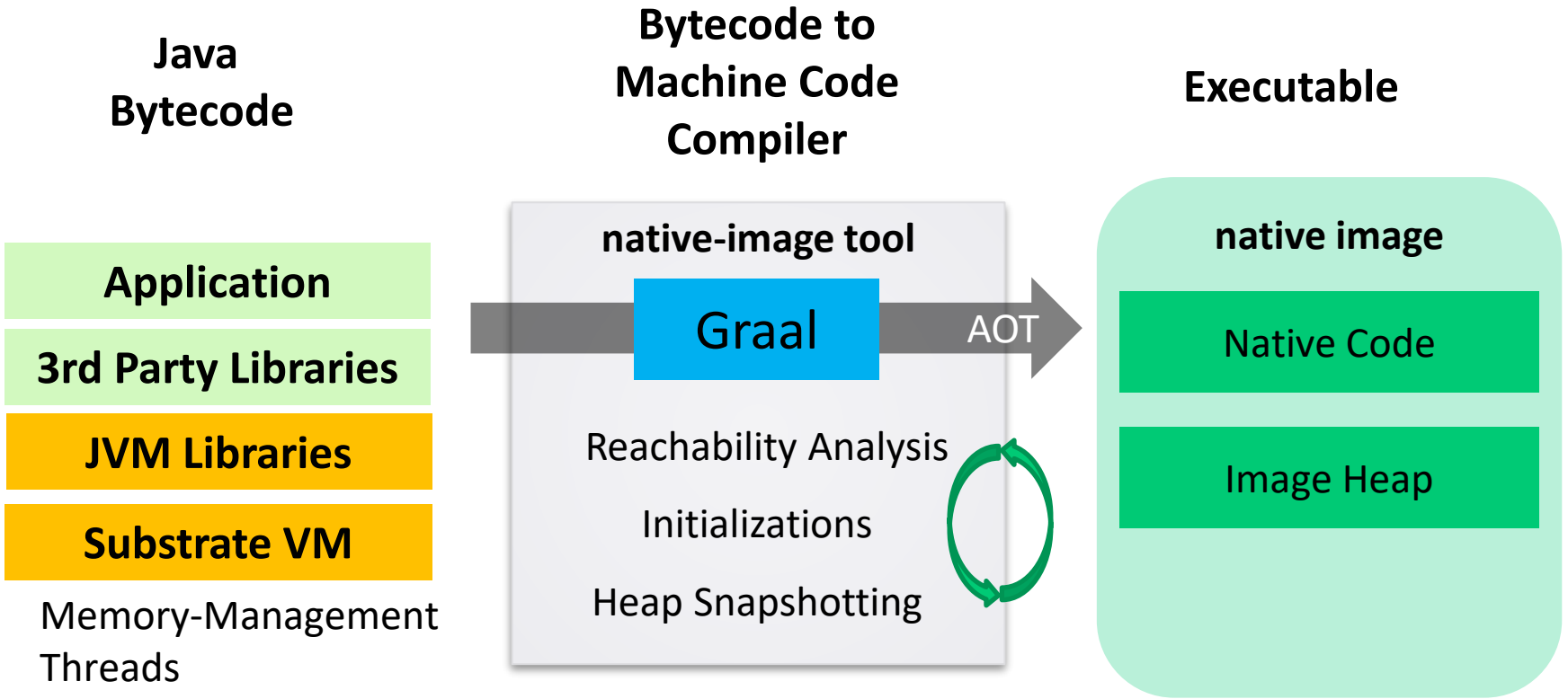
- Increased startup time
- Performance gains only after warmup phase
- Memory and CPU consumption at runtime (profiling, compilation)
- Harder to predict

For some workloads, this may be undesirable

Example: Azure Functions for Java disable C2 compiler by default:

```
-XX:+TieredCompilation
-XX:TieredStopAtLevel=1
```

# GraalVM Native Image: AOT Compilation of Java Bytecode




- Memory-Management
- Threads
- …

# Performance of Native Images vs JVM (v22.2.0 CE)

Example: Simple Spring Boot Webservice

## Startup Time

**JVM:** 1.60sec
**native:** 0.06sec

## Throughput

**JVM:** 24k r/s (+/- 1k)
**native:** 16k r/s (+/- 1.5k)

## Memory Footprint

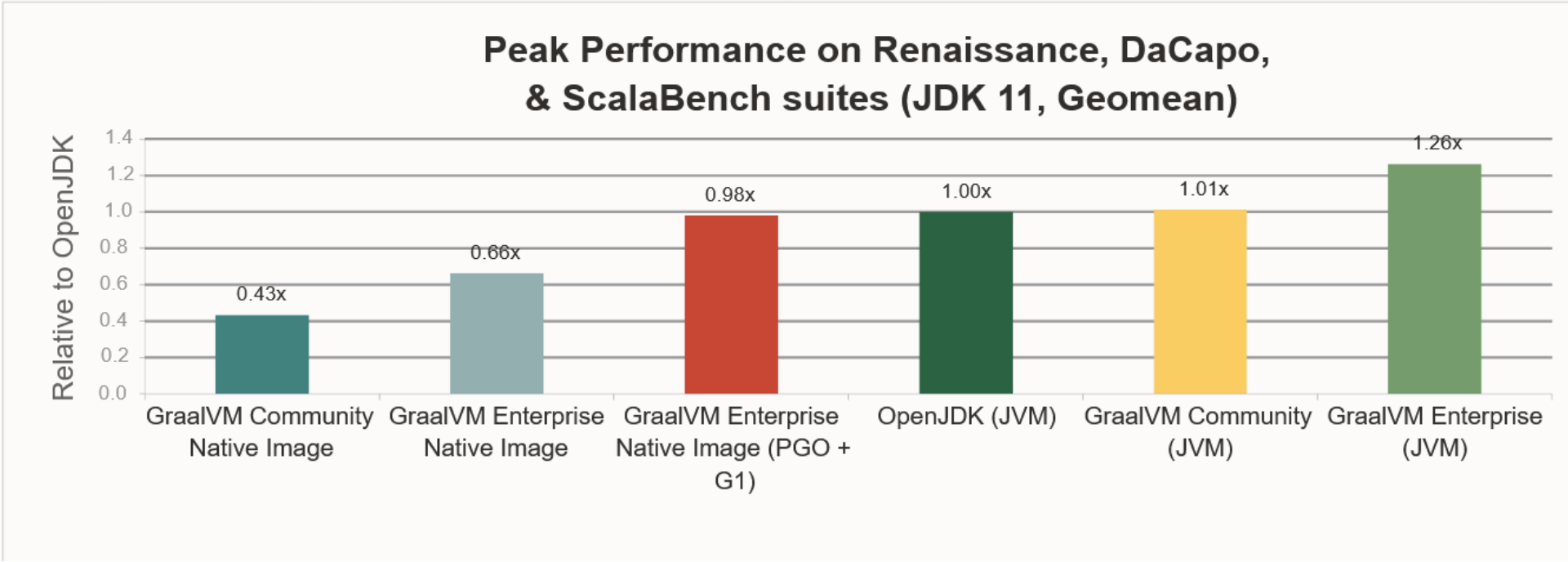
**JVM:** 300 MB
**native:** 120 MB

## Compile Time

**native:** 2:10 min

https://github.com/accso/graalvm-native/tree/main/spring-boot-greetingserver

# Performance of Native Images vs JVM




Source: https://blogs.oracle.com/java/post/graalvm-enterprise-213

# Closed-World Assumption: Everything Must be Known at Build Time

Dynamic features **must** be configured at build time or they won‘t be available at runtime

- Reflection
- Proxy Classes
- Resources
- Dynamic Classloading
- …

```
…
{
  "name": "ch.qos.logback.classic.pattern.DateConverter",
  "allDeclaredFields": true,
  "allDeclaredConstructors": true,
  "allDeclaredMethods": true
},
{
  "name": "ch.qos.logback.classic.pattern.LevelConverter",
  "allDeclaredFields": true,
  "allDeclaredConstructors": true,
  "allDeclaredMethods": true
},
…
```

GraalVM tools and some frameworks help with this

https://docs.oracle.com/en/graalvm/enterprise/22/docs/reference-manual/native-image/dynamic-features/

# Summary: Native Images are an Interesting Option for Some Workloads

Native images are suitable where

- → startup time and memory consumption are imporant
- → image size is important
- → long-term runtime performance is less important
- → applications are relatively small
- → not too many objects are created

In some of these cases consider using a different language

# Related Projects for the JVM

**jaotc:** AOT-compilation of selected code-parts as a shared library for the JVM (removed in Java 16)

https://docs.oracle.com/en/java/javase/13/docs/specs/man/jaotc.html

**Project Leyden**: Closed-world static images for OpenJDK

https://openjdk.org/projects/leyden/

**Class Data Sharing**: Improving startup time by storing class file snapshots

https://docs.oracle.com/javase/8/docs/technotes/guides/vm/class-data-sharing.html

# SHARING YOUR CHALLENGE

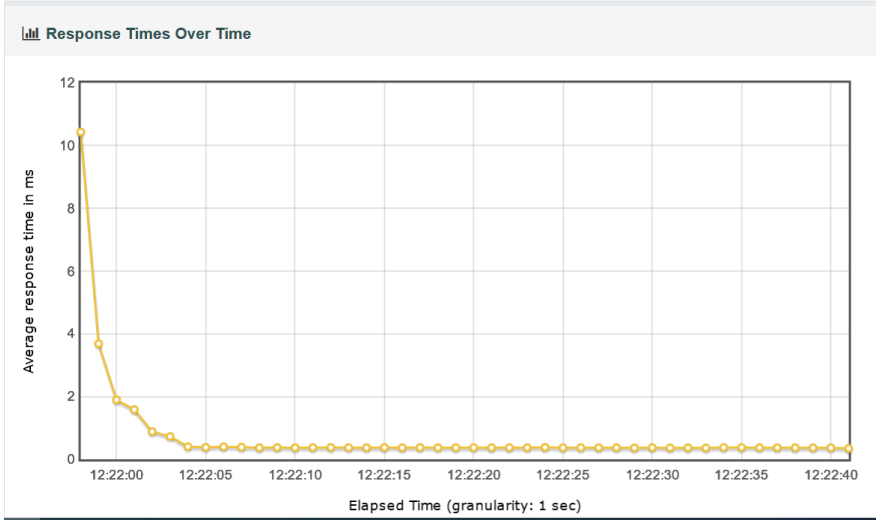
ACCSO
ACCELERATED SOLUTIONS

Accso – Accelerated Solutions GmbH
| +49 6151 13029-0
| info@accso.de
| www.accso.de

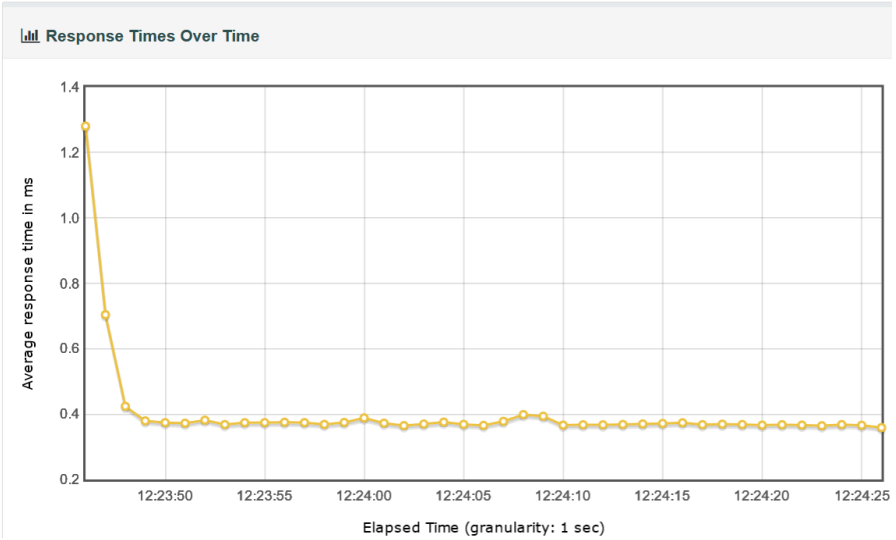
Hilpertstraße 12 | 64295 Darmstadt
Rahmhofstraße 2-4 | 60313 Frankfurt a.M.
Im Mediapark 6a | 50670 Köln
Balanstraße 55 | 81541 München

# Backup – Response Times over Time

## JVM – Cold

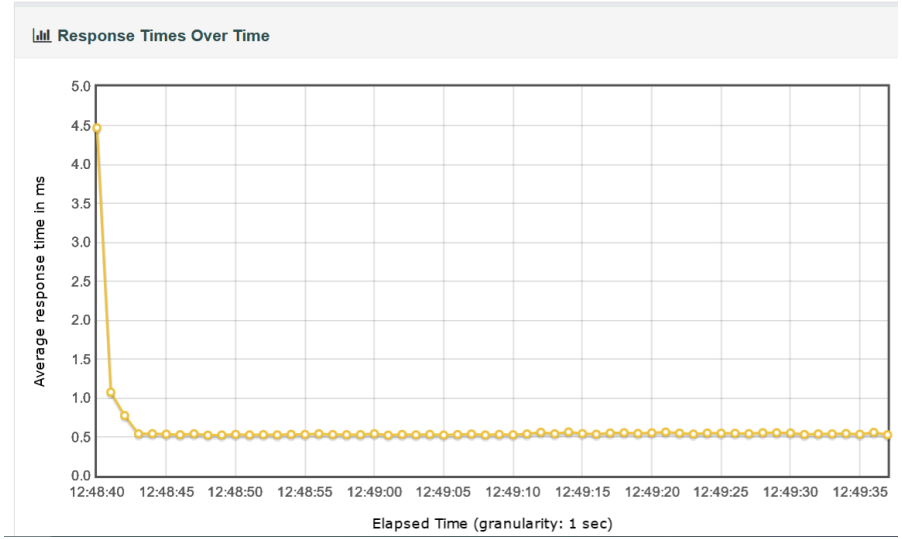



## JVM – Warm



# Backup – Response Times over Time

## Native – Cold




## Native – Warm