GraalVM™

JDK 14 und GraalVM im Java Ökosystem

Wolfgang Weigend

Master Principal Solution Engineer | global Java Team

Java Technology & GraalVM and Architecture

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, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

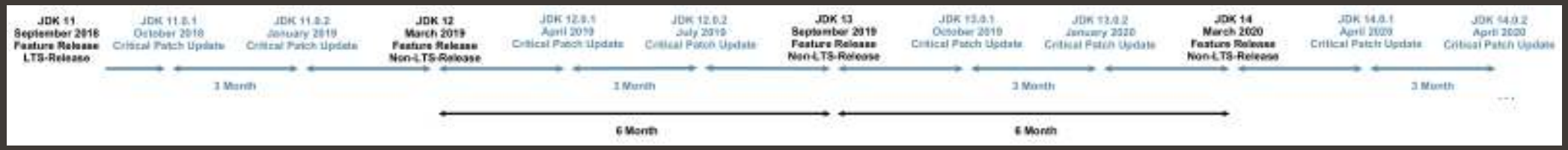
GraalVM Native Image early adopter status

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

Agenda

1. JDK 14 Features
2. Java Eco System and Commitment to Open Source
3. GraalVM Architecture
4. GraalVM Performance
5. GraalVM Downloads
6. GraalVM Footprint
7. GraalVM Native Image
8. Summary

JDK Version Numbers and Java Critical Patch Updates



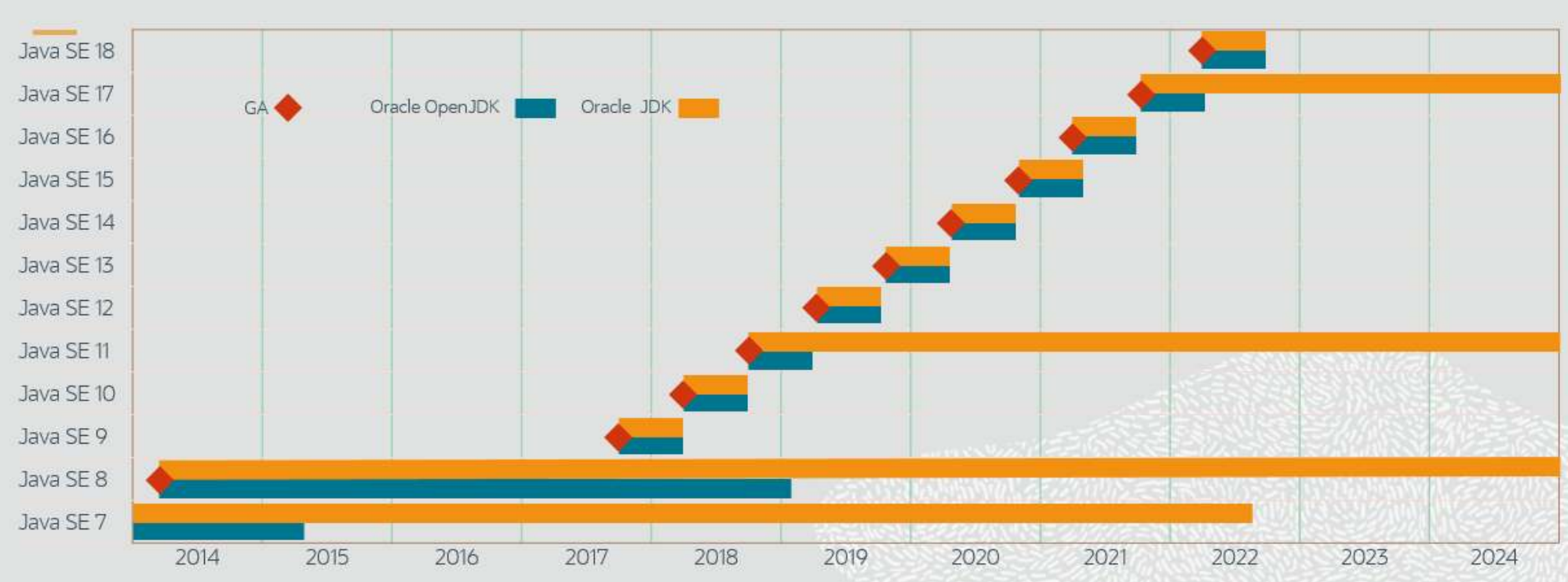
- Rules for Java CPU's
  - Main release for security vulnerabilities
  - Covers all JDK families (14, 13, 12, 11, 8, 7, 6)
  - CPU release triggers Auto-update
  - Dates published 12 months in advance
  - Security Alerts are released as necessary
  - Based off the previous (non-CPU) release
  - Released simultaneously on java.com and OTN

- JDK 14.0.1 - Security Baselines

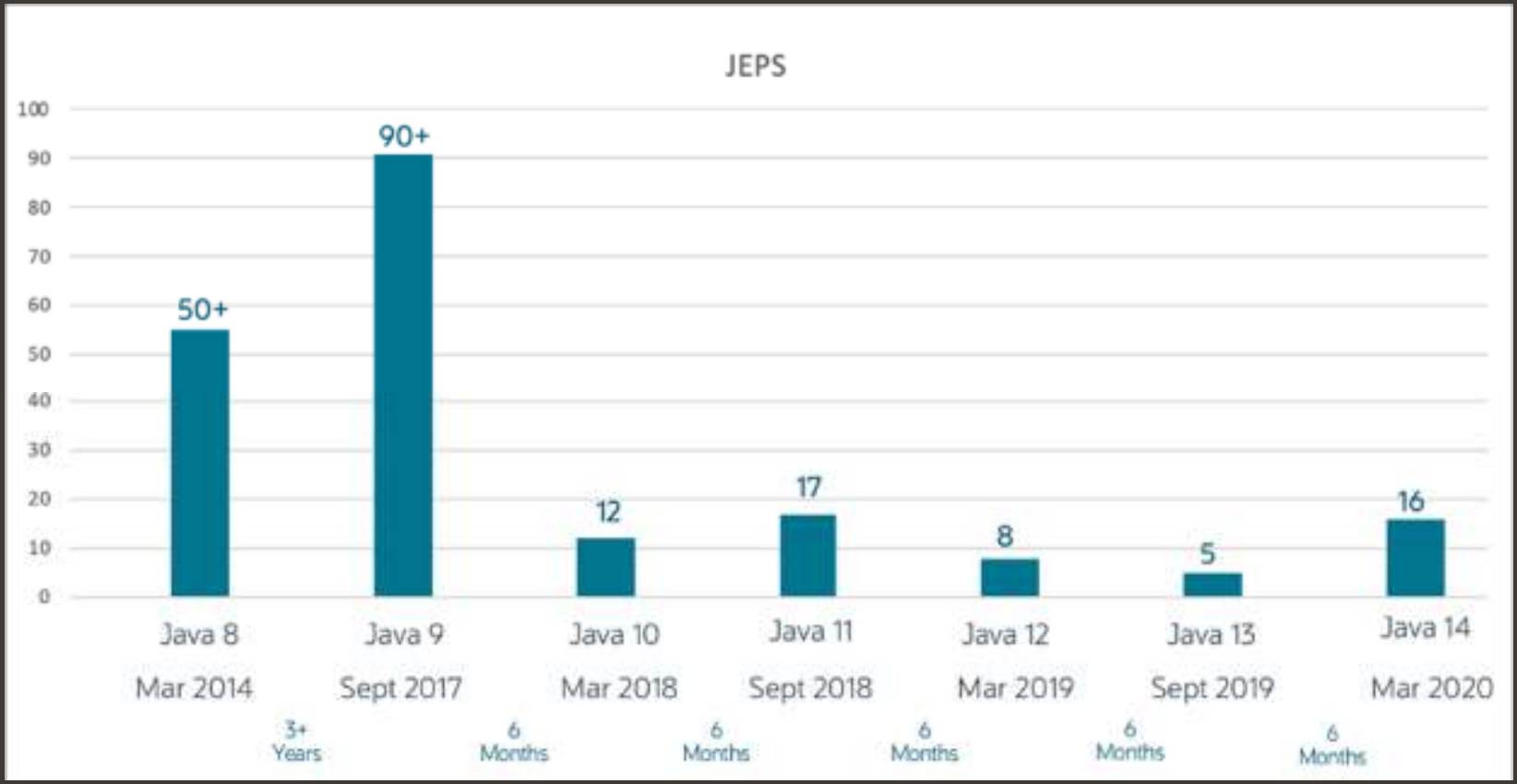
JRE Family Version	JRE Security Baseline (Full Version String)
13	13.0.2
12	12.0.2
11	11.0.7+8
10	10.0.2
9	9.0.4
8	1.8.0_251-b08
7	1.7.0_261-b07
6	1.6.0_221

JDK 6 Month Release Cadence

Option to use Oracle JDK or Oracle OpenJDK

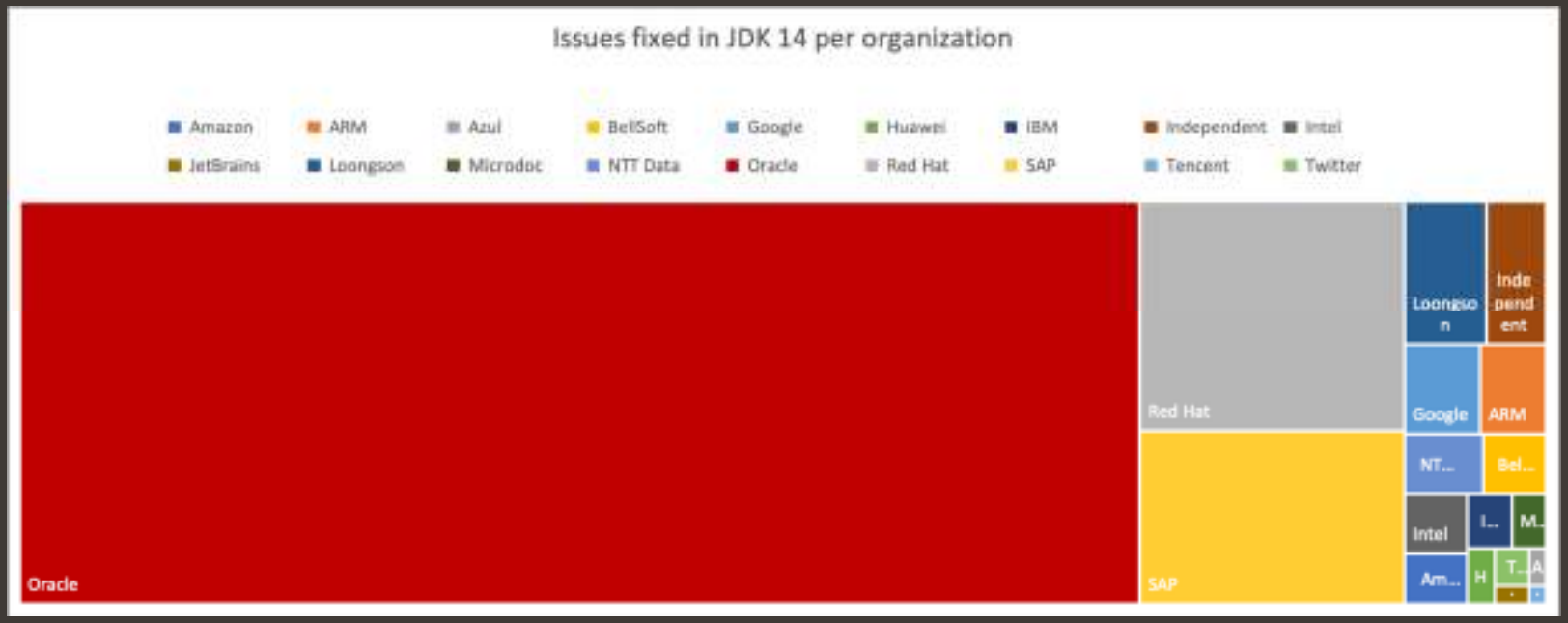


Targeted JEP's per JDK Release



Issues fixed in JDK 14 per organization

https://blogs.oracle.com/java-platform-group/the-arrival-of-java-14



- JDK BUG System commits
- Overall 1986 JIRA issues marked as fixed in JDK 14
- 1458 issues were completed by Oracle
- 528 issues were contributed by individual developers and developers working for other organizations

JDK 14 Standard and Preview Features

```
$ javac HelloWorld.java                                  // Do not enable any preview features
$ javac --release 14 --enable-preview HelloWorld.java    // Enable all preview features of JDK 14
$ java --enable-preview HelloWorld                       // Run with preview features of JDK 14
```

```
$ javac --release 14 --enable-preview HelloWolfgang.java
$ java HelloWolfgang
Error: LinkageError occurred while loading main class HelloWolfgang
        java.lang.UnsupportedClassVersionError: Preview features are not enabled for
HelloWolfgang (class file version 58.65535). Try running with '--enable-preview'
$ java --enable-preview HelloWolfgang
Hello Wolfgang!
Date today = 2020-04-14
Time now = 20:57:07.180454400
```

JDK 14 – Features – JEP's

- 305: Pattern Matching for instanceof (Preview)
- 343: Packaging Tool (Incubator)
- 345: NUMA-Aware Memory Allocation for G1
- 349: JFR Event Streaming
- 352: Non-Volatile Mapped Byte Buffers
- 358: Helpful NullPointerExceptions
- 359: Records (Preview)
- 361: Switch Expressions (Standard)
- 362: Deprecate the Solaris and SPARC Ports
- 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector
- 364: ZGC on macOS
- 365: ZGC on Windows
- 366: Deprecate the ParallelScavenge + SerialOld GC Combination
- 367: Remove the Pack200 Tools and API
- 368: Text Blocks (Second Preview)
- 370: Foreign-Memory Access API (Incubator)

https://openjdk.java.net/projects/jdk/14/

JEP 305: Pattern Matching (Preview)

https://openjdk.java.net/jeps/305

- Enhance the Java programming language with pattern matching for the instanceof operator

```
if (obj instanceof String) {
    String s = (String) obj;
    // use s
}



// new form
if (obj instanceof String s) {
     // use s here
}
```

JDK 14 Switch Expressions (Standard)

- Switch Expressions

```
int numLetters;
switch (day) {
   case MONDAY:
   case FRIDAY:
   case SUNDAY:
      numLetters = 6;
      break;
   case TUESDAY:
      numLetters = 7;
      break;
   case THURSDAY:
   case SATURDAY:
      numLetters = 8;
      break;
   case WEDNESDAY:
      numLetters = 9;
      break;
   default:
      throw new IllegalStateException (”wat: " + day);
}
```

After:

```
int numLetters = switch (day) {
   case MONDAY, FRIDAY, SUNDAY -> 6;
   case TUESDAY                -> 7;
   case THURSDAY, SATURDAY     -> 8;
   case WEDNESDAY              -> 9;
};
```

JEP 361: Switch Expressions (Standard)

- JDK 14 standard feature
- Unchanged from the second preview in JDK 13
- Without the --enable-preview flag
- https://openjdk.java.net/jeps/361

JDK 14 Text Blocks (Second Preview)

- Text Blocks (Preview)

```
String html =  "<html>\n" +
               "   <body>\n" +
               "       <p>Hello, world</p>\n" +
               "   </body>\n" +
               "</html>\n";
```

```
String html = """
              <html>
                 <body>
                    <p>Hello, world</p>
                 </body>
              </html>
              """;
```

JEP 368: JDK 14 Text Blocks (Second Preview)

New escape sequences

To allow finer control of the processing of newlines and white space, we introduce two new escape sequences.

First, the \<line-terminator> escape sequence explicitly suppresses the insertion of a newline character.

For example, it is common practice to split very long string literals into concatenations of smaller substrings, and then hard wrap the resulting string expression onto multiple lines:

```
String literal = "Der Satz fäng an " +
                 "und geht weiter " +
                 "bis zum Schluss.";
```

With the \<line-terminator> escape sequence this could be expressed as:

```
String text = """
                Der Satz fäng an \
                und geht weiter \
                bis zum Schluss.\
                """;
```

For the simple reason that character literals and traditional string literals don't allow embedded newlines, the \<line-terminator> escape sequence is only applicable to text blocks.

JEP 368: JDK 14 Text Blocks (Second Preview)

Second, the new \s escape sequence simply translates to a single space (\u0020).

Escape sequences aren't translated until after incident space stripping, so \s can act as fence to prevent the stripping of trailing white space. Using \s at the end of each line in this example guarantees that each line is exactly six characters long:

```
String colors = """
       red  \s
       green\s
       blue \s
       """;
```

The \s escape sequence can be used in both text blocks and traditional string literals.

JEP 359: Records (Preview)

https://openjdk.java.net/jeps/359

- Records provide a compact syntax for declaring classes that are nothing more, or mostly, plain carriers that serves as simple aggregates
- Example: `record Point(int x, int y) { }`
- Records acquire many standard members automaticaly
  - A private final field for each component
  - A public read accessor for each component
  - A public constructor
  - Implementations or equals, hashCode and toString
- *Replacement of data class with „records“*
- *Records as a “simple data encapsulation”*
- *“Records are serialized differently than ordinary serializable or externalizable objects. The serialized form of a record object is a sequence of values derived from the record components.”* https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/io/ObjectInputStream.html#record-serialization

Packaging Tool (Incubator)

What is jpackage?

- A JDK command line tool for packaging self-contained Java applications
  - Meant for Java application developers (not end users)
  - Input: a pre-built Java application, a Java runtime
  - Output: a native application package
  - Supported on: Linux, macOS, Windows
  - Provides a straightforward way to give the end user a natural installation experience on their platform
- Defined by JEP 343:
  - https://openjdk.java.net/jeps/343

Why do we need jpackage? (1)

- As of JDK 11:
  - Shared, auto-updated Java runtime (System JRE) no longer available
  - Java Web Start / applets are gone (and incomplete without the above)
- New deployment model: bundled application + Java runtime
  - jpackage takes a Java runtime and your application, and creates a native package that you can distribute
    - jpackage runs jlink to create runtime, including application if modular
    - Developer can run jlink prior to jpackage for additional customization

Why do we need jpackage? (2)

- jlink provides part of the puzzle
  - Creates a custom Java runtime from the JDK
    - can add additional library or application modules
    - can exclude unneeded modules
    - same file system layout as the JDK
  - Developers can copy that plus their app and zip it up
    - but this is not a “ready-to-distribute” bundle
    - it's a collection of files, not an installable package or application
    - no system integration
- What then?
  - Today Java developers “roll their own” using various third-party tools

Where does jlink fit in?

- Use jlink to create a custom runtime image:
  - Include only the modules you need
  - Add your own modules or library modules

- jpackage will run jlink for you in most cases, but you can run it directly for more control over the Java runtime image you use

Using jlink

```
// Custom runtime with only the java.base module
jlink --output my-jdk --add-modules java.base

// Java runtime with the specified modules (includes java.base)
jlink --output my-jdk --add-modules java.desktop,java.datatransfer

// Java runtime with your modular application and its dependencies
// Strip unneeded files
jlink --output my-jdk --module-path mymod.jar --add-modules mymod \
  --no-man-pages --no-header-files  --strip-native-commands
```

Features of jpackage (1)

- Can be used to package:
  - Desktop apps
  - Command line apps
- Creation of an application image
- Support the following native packaging formats:
  - Linux: rpm (Red Hat), deb (Debian)
  - Mac: dmg, pkg
  - Windows: exe, msi
- Specify JDK / app args that will be used when launching the app
- Can be invoked from command line, or via the ToolProvider API

Features of jpackage (2)

- Package applications that integrate into the native platform:
  - Set file associations to launch app when file with associated suffix is opened
  - Launching from platform-specific menu group, e.g., Start menu on Windows
  - Desktop shortcuts
  - Option to specify update rules for installable packages (such as in rpm/deb)
  - Multiple launchers can be created from the same application image
- List of required tools:
  - Linux (Red Hat) : rpmutil (4.0d or later)
  - Linux (Ubuntu) : dpkg, fakeroot
  - macOS: XCode command line tools
  - Windows: WiX Toolset (3.0 or later)

Features of jpackage (3)

- What does jpackage *not* do?
  - Does not solve how to get the platform installer to the user
    - no browser / web integration
  - No cross-platform deployment
  - No cross-compilation (e.g., must run on macOS to produce dmg)
  - No support for jnlp
- JNLPConverter demo can ease transition:
  - Takes a jnlp file + resources (jar files) runs jpackage
  - Doesn't handle all cases, but will help you convert

JEP349: JFR Event Streaming (1)

- Expose JDK Flight Recorder data for continuous monitoring

- Goals
  - Provide an API for the continuous consumption of JFR data on disk, both for in-process and out-of-process applications
  - Record the same set of events as in the non-streaming case, with overhead less than 1% if possible
  - Event streaming must be able to co-exist with non-streaming recordings, both disk and memory based

- Non-Goals
  - Provide synchronous callbacks for consumers
  - Allow consumption of in-memory recordings

JEP349: JFR Event Streaming (2)

The following example prints the overall CPU usage and locks contended for more than 10 ms

```
try (var rs = new RecordingStream()) {
  rs.enable("jdk.CPULoad").withPeriod(Duration.ofSeconds(1));
  rs.enable("jdk.JavaMonitorEnter").withThreshold(Duration.ofMillis(10));
  rs.onEvent("jdk.CPULoad", event -> {
    System.out.println(event.getFloat("machineTotal"));
  });
  rs.onEvent("jdk.JavaMonitorEnter", event -> {
    System.out.println(event.getClass("monitorClass"));
  });
  rs.start();
}
```

JEP349: JFR Event Streaming (3)

The RecordingStream class implements the interface jdk.jfr.consumer.EventStream that provides a uniform way to filter and consume events regardless if the source is a live stream or a file on disk

```
public interface EventStream extends AutoCloseable {
  public static EventStream openRepository();
  public static EventStream openRepository(Path directory);
  public static EventStream openFile(Path file);

  void setStartTime(Instant startTime);
  void setEndTime(Instant endTime);
  void setOrdered(boolean ordered);
  void setReuse(boolean reuse);

  void onEvent(Consumer<RecordedEvent> handler);
  void onEvent(String eventName, Consumer<RecordedEvent handler);
  void onFlush(Runnable handler);
  void onClose(Runnable handler);
  void onError(Runnable handler);
  void remove(Object handler);

  void start();
  void startAsync();

  void awaitTermination();
  void awaitTermination(Duration duration);
  void close();
}
```

JEP358: Helpful NullPointerExceptions (1)

- Improve the usability of NullPointerExceptions generated by the JVM by describing precisely which variable was null

- The JVM throws a NullPointerException (NPE) at the point in a program where code tries to dereference a null reference. By analyzing the program's bytecode instructions, the JVM will determine precisely which variable was null, and describe the variable (in terms of source code) with a null-detail message in the NPE. The null-detail message will then be shown in the JVM's message, alongside the method, filename, and line number.

- *Mittels Programm-ByteCode-Command-Analyse erkennt die JVM, welche Variable den Wert Null ergibt*

JEP358: Helpful NullPointerExceptions (2)

The JVM displays an exception message on the same line as the exception type, which can result in long lines. For readability in a web browser, this JEP shows the null-detail message on a second line, after the exception type.

For example, an NPE from the assignment statement a.i = 99; would generate this message:

```
Exception in thread "main" java.lang.NullPointerException:
          Cannot assign field "i" because "a" is null
        at Prog.main(Prog.java:5)
```

If the more complex statement a.b.c.i = 99; throws an NPE, the message would dissect the statement and pinpoint the cause by showing the full access path which led up to the null:

```
Exception in thread "main" java.lang.NullPointerException:
          Cannot read field "c" because "a.b" is null
        at Prog.main(Prog.java:5)
```

Giving the full access path is more helpful than giving just the name of the null field because it helps the developer to navigate a line of complex source code, especially if the line of code uses the same name multiple times.

JEP's targeted to JDK 15, so far

This release will be the Reference Implementation of Java SE 15, as specified by JSR 390 in the Java Community Process.

- Features
- 371: Hidden Classes
- 372: Remove the Nashorn JavaScript Engine
- 377: ZGC: A Scalable Low-Latency Garbage Collector
- 378: Text Blocks
- 379: Shenandoah: A Low-Pause-Time Garbage Collector

JEP 371: Hidden Classes

- Introduce hidden classes, which are classes that cannot be used directly by the bytecode of other classes
- Hidden classes are intended for use by frameworks that generate classes at run time and use them indirectly, via reflection
- A hidden class may be defined as a member of an access control nest, and may be unloaded independently of other classes

Java Eco System and Commitment to Open Source

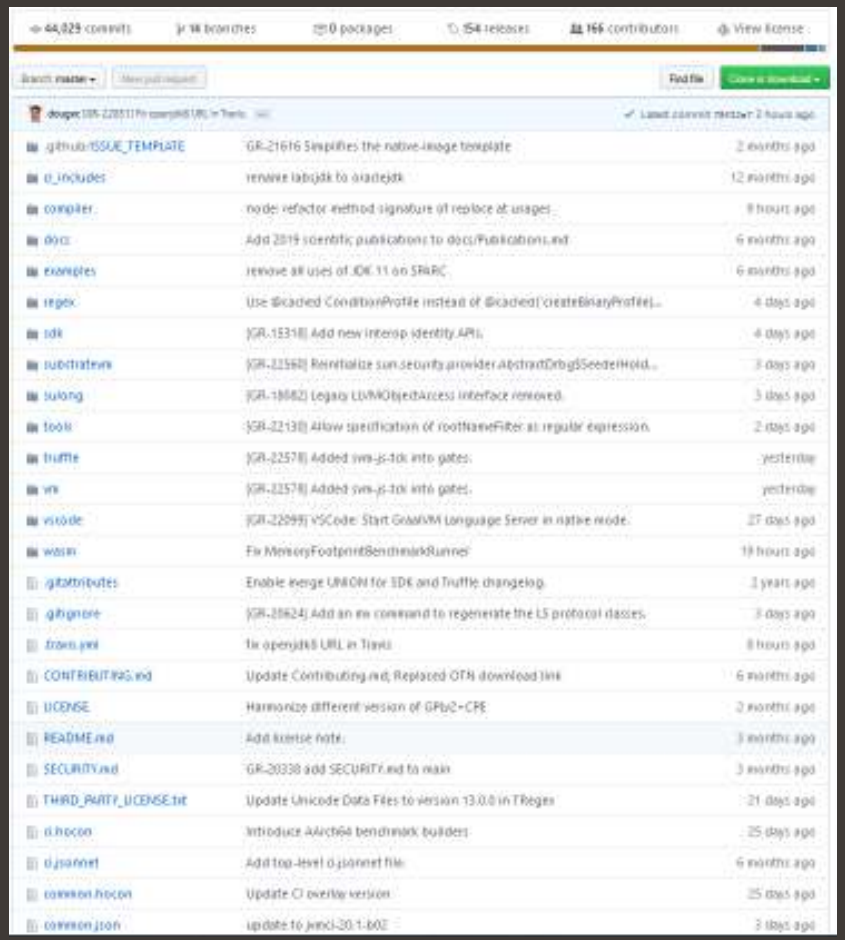
GraalVM

Java Eco System

- 12 Million developers run Java
- #1 Programming language
- #1 Developer choice for the cloud
- 45 Mrd. active Java Virtual Machines
- 25 Mrd. cloud connected Java Virtual Machines

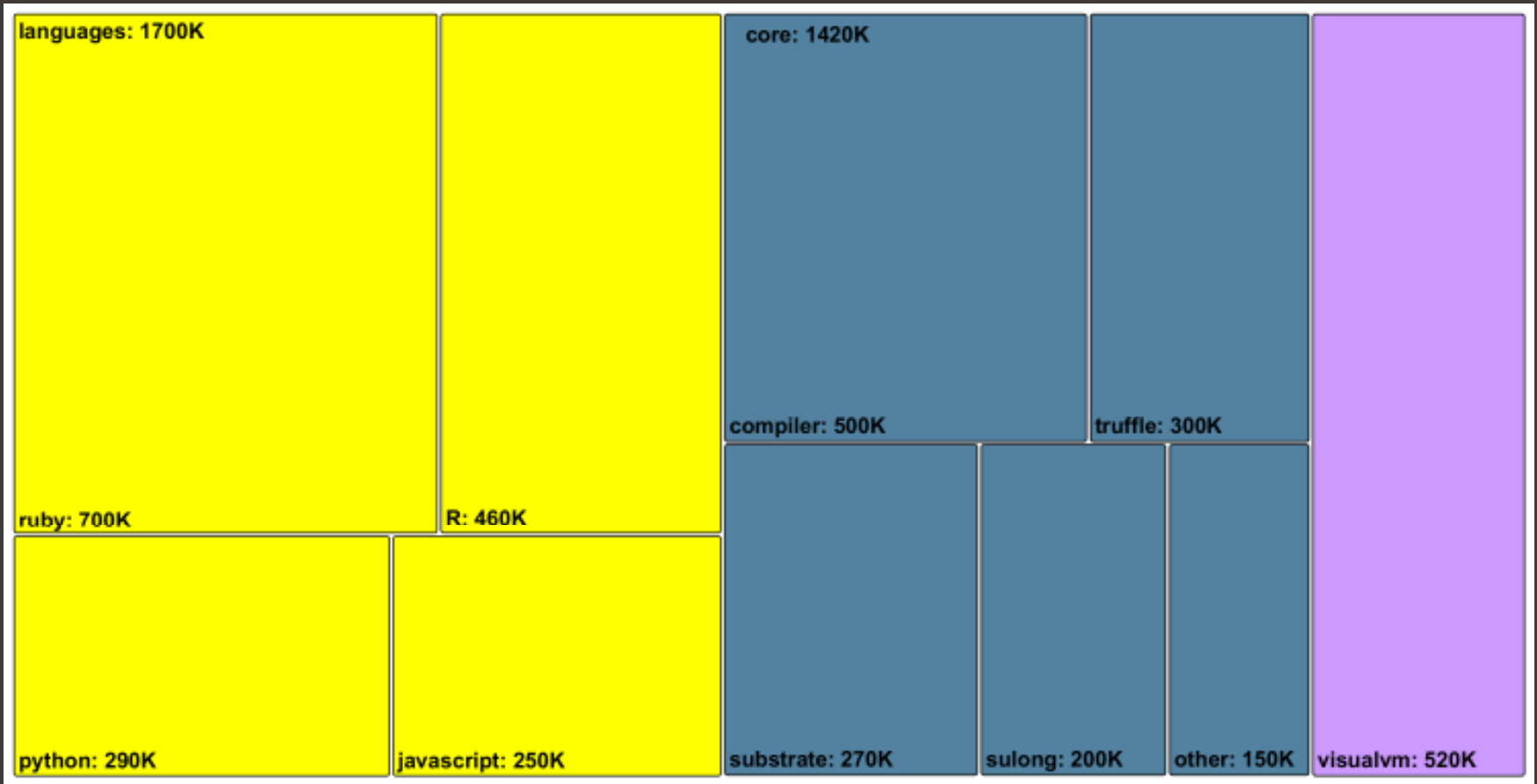
Graal on GitHub

https://github.com/oracle/graal



GraalVM Open Source Lines of Code

GraalVM Community Edition is built from the sources of 3.6 million lines of code originated by the GraalVM team and collaborators, and additionally million lines of sources from projects we depend on like Java, Node.js and others



GraalVM Project Advisory Board

https://www.graalvm.org/community/advisory-board/

- **The main goals of the GraalVM Project Advisory Board are to:**
  - Discuss community engagement and contributor interaction
  - Provide cumulative feedback from the community and partner ecosystem
  - Discuss ways to drive project awareness and adoption
    - The Board will meet at least every three months, and the meeting notes will be published at graalvm.org
    - Alina Yurenko is the board's coordinator

- **Initial board composition with members nominated by 12 different companies:**
  - Bernd Mathiske, Amazon. Creator of the Maxine VM, interested in GraalVM Community Edition, GraalVM Native Image, and AWS Lambda on GraalVM.
  - Bruno Caballero, Microdoc. Work on GraalVM integrations in the embedded space.
  - Chris Seaton, Shopify. Contributors to TruffleRuby — GraalVM Ruby implementation.
  - Chris Thalinger, Twitter. Runs GraalVM Community Edition in production on a large scale system and shares their experience with the community.
  - Fabio Niephaus, Hasso Plattner Institute. Academic collaborators and developers of GraalSqueak — A Squeak/Smalltalk implementation for GraalVM.
  - Graeme Rocher, Object Computing Inc., Developers of Micronaut — a framework for building microservice and serverless applications, integrated with GraalVM.
  - Johan Vos, Gluon. Works on JavaFX and client (desktop/mobile/embedded) support for GraalVM native images.
  - Max Rydahl Andersen, Red Hat. Developer on Quarkus — A Kubernetes Native Java stack tailored for OpenJDK HotSpot and GraalVM, crafted from the best of breed Java libraries and standards.
  - Michael Hunger, Neo4j. Integrated with GraalVM to support polyglot dynamic languages for user-defined-procedures in Neo4j, a JVM-based graph database.
  - Sébastien Deleuze, Pivotal. Spring Framework committer, works on Spring GraalVM native support.
  - Thomas Wuerthinger, Oracle. GraalVM Founder and Project Lead.
  - Xiaohong Gong, Arm Technology China. Works on GraalVM Compiler Optimizations on AArch64.

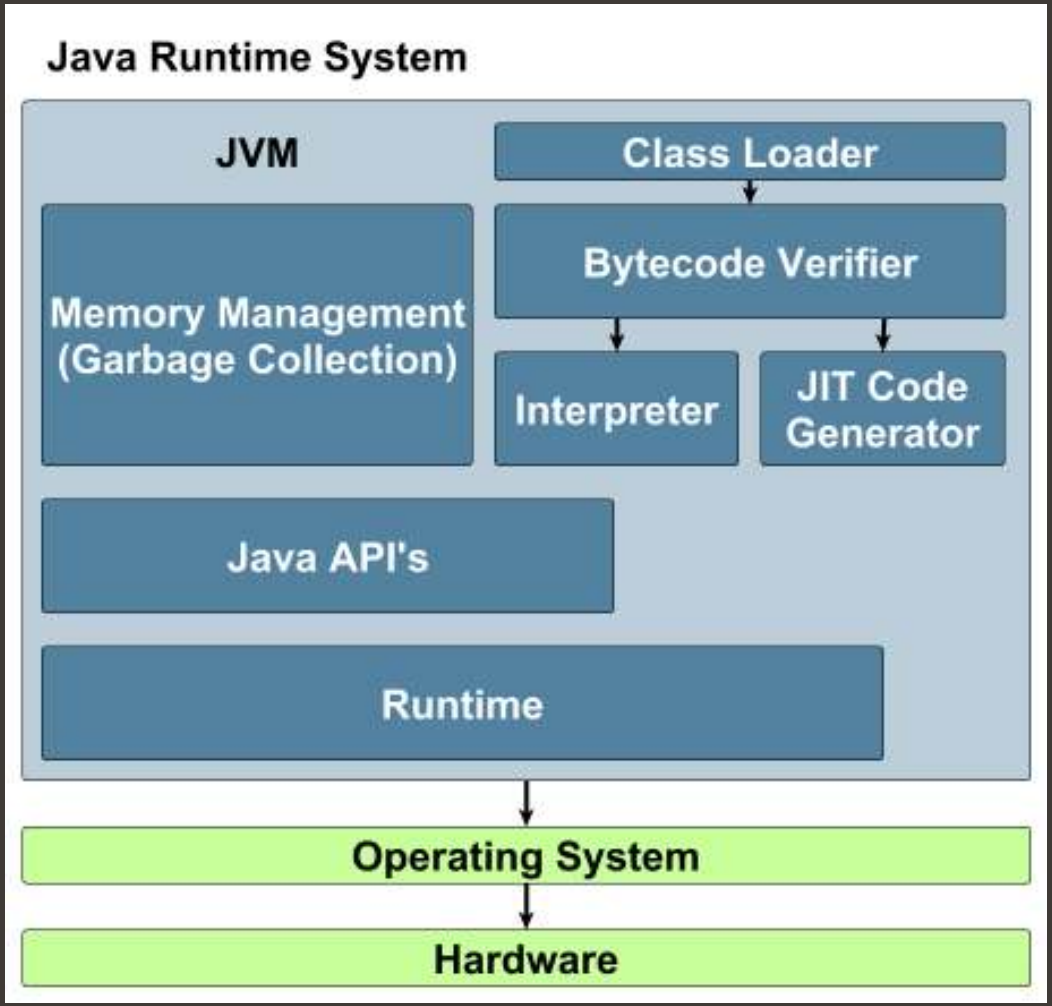
GraalVM Architecture

GraalVM Repository Structure

The GraalVM main source repository includes these components

- *Graal SDK* contains long term supported APIs of GraalVM.
- *Graal compiler* written in Java that supports both dynamic and static compilation and can integrate with the Java HotSpot VM or run standalone.
- *Truffle* language implementation framework for creating languages and instrumentations for GraalVM.
- *Tools* contains a set of tools for GraalVM languages implemented with the instrumentation framework.
- *Substrate VM* framework that allows ahead-of-time (AOT) compilation of Java applications under closed-world assumption into executable images or shared objects.
- *Sulong* is an engine for running LLVM bitcode on GraalVM.
- *TRegex* is an implementation of regular expressions which leverages GraalVM for efficient compilation of automata.
- *VM* includes the components to build a modular GraalVM image.

Java Runtime mit JVM



JIT Compiler

- C1 Client Compiler
  - Minimiert Startup-Zeit
- C2 Server Compiler
  - Dauerhafte Performance-Verbesserungen
  - Intensivere Analyse vom ausgeführten Code
  - Optimierungen können besser platziert werden

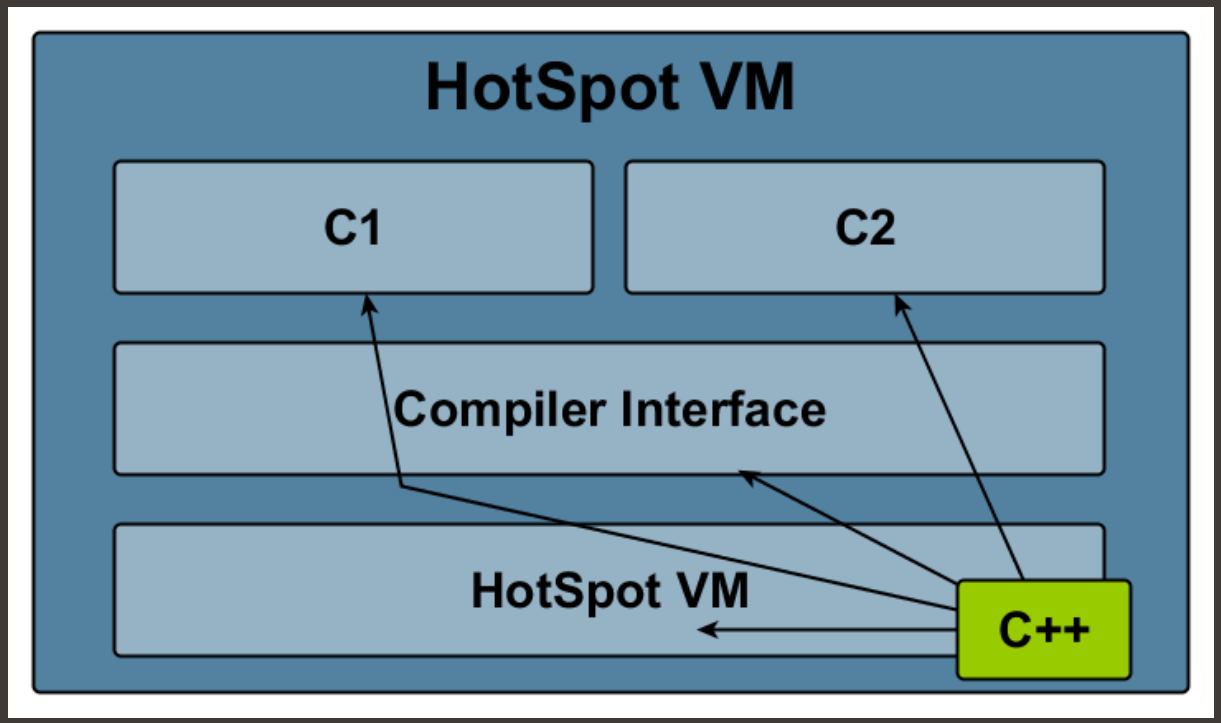
JIT Compiler with Tiered Compilation

- C1 Client Compiler
  - Minimiert Startup-Zeit
  - java -client -XX:+TieredCompilation
- C2 Server Compiler
  - Läuft mit, aber ohne Tiered Compilation
- Tiered-Compilation Ausführungs-Level
  - Level 0: interpreted code
  - Level 1: simple C1 compiled code (with no profiling)
  - Level 2: limited C1 compiled code (with light profiling)
  - Level 3: full C1 compiled code (with full profiling)
  - Level 4: C2 compiled code (uses profile data from the previous steps)

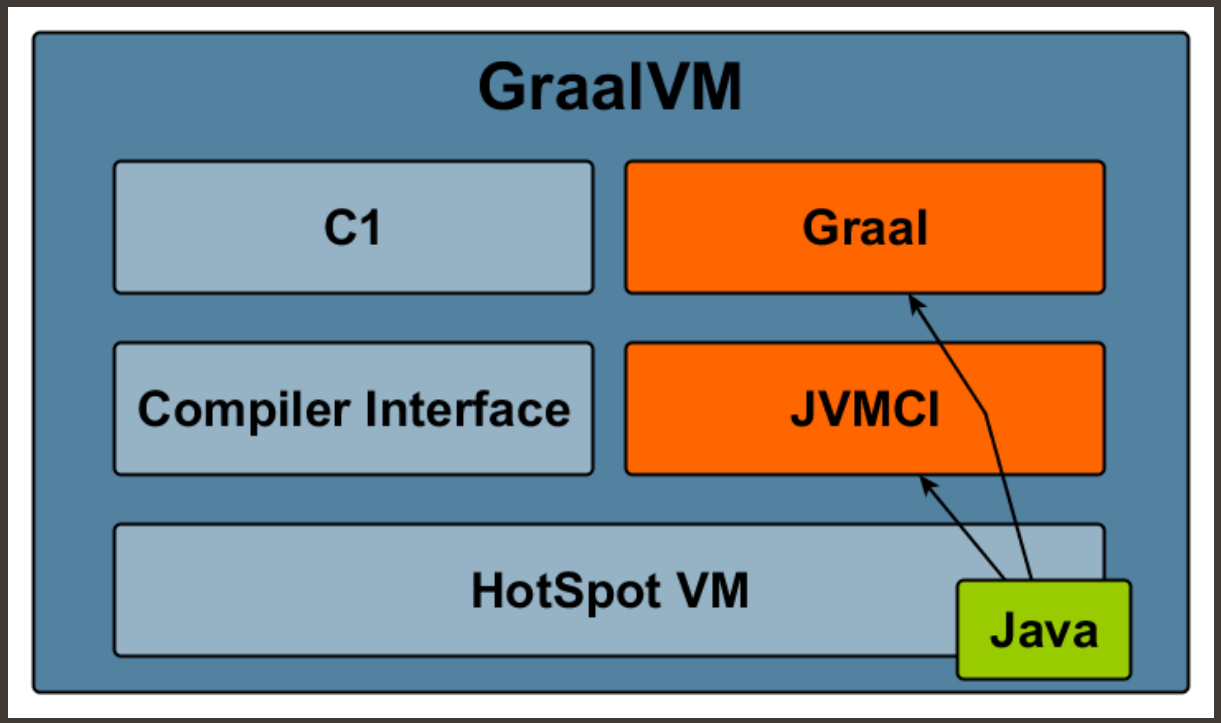
JIT Compiler working

- Inlining
  - Code der aufzurufenden Methode/Funktion anstelle des Aufrufs
- On-Stack Replacement
  - Loop-Compilation, ohne auf den Methodenaufruf zu warten
- Escape Analysis
  - Automatische Stack-Allokation, ohne GC
- De-Optimierung
  - Optimierung rückgängig machen

JIT Compiler written in C++



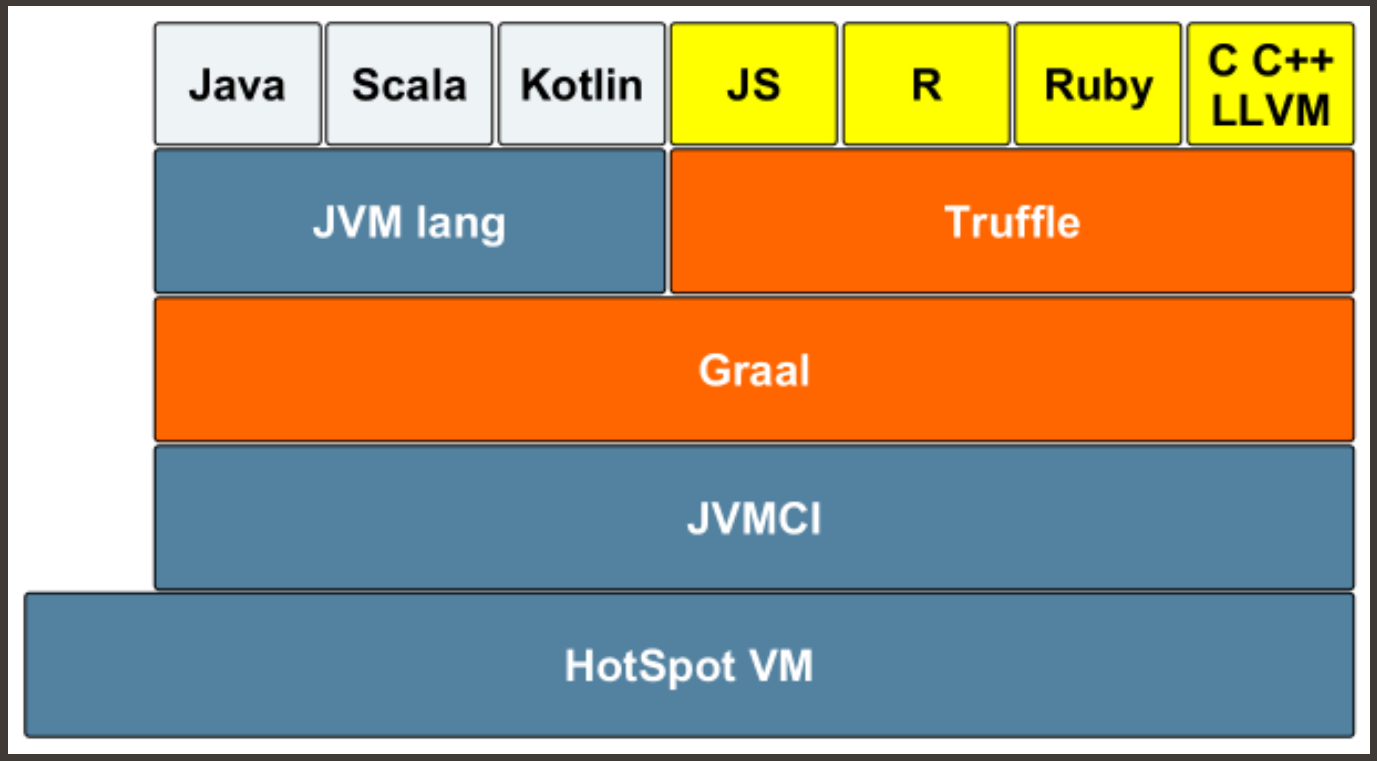
JIT Compiler written in Java



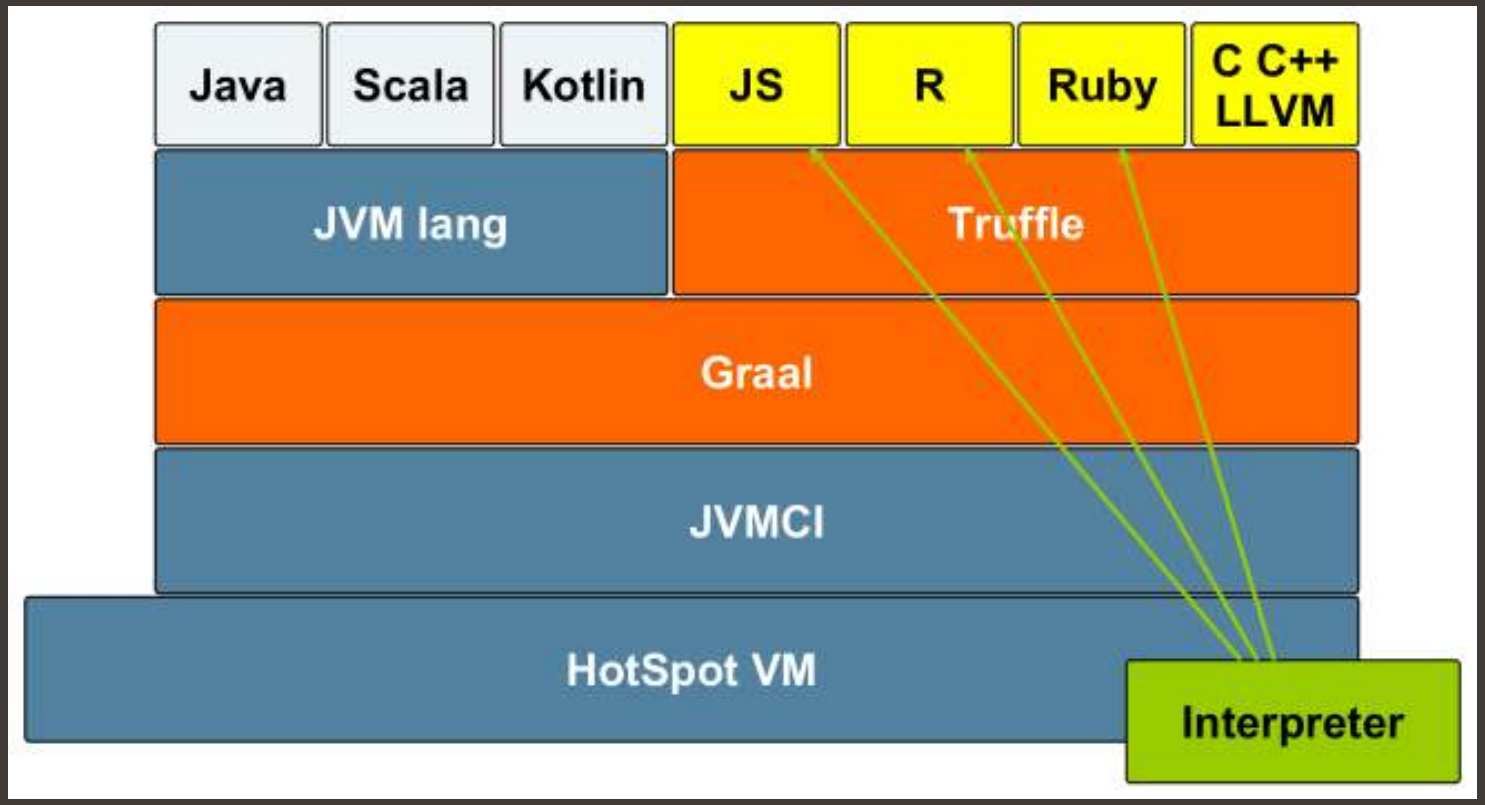
GraalVM

- Graal
  - JIT Compiler
    - Graal in GraalVM - A new Java JIT Compiler
  - Graal integrated via Java Virtual Machine Compiler Interface (JVM CI)
  - Use a JDK with Graal (jdk.internal.vm.compiler)
  - java –XX:+UnlockExperimentalVMOptions –XX:+EnableJVMCI –XX:+UseJVMCICompiler –jar my_file.jar
- Truffle
  - Language Implementation Framework
- Substrate VM
  - Runtime Library and a set of tools for building Java AOT compiled code

GraalVM - Polyglot (1)

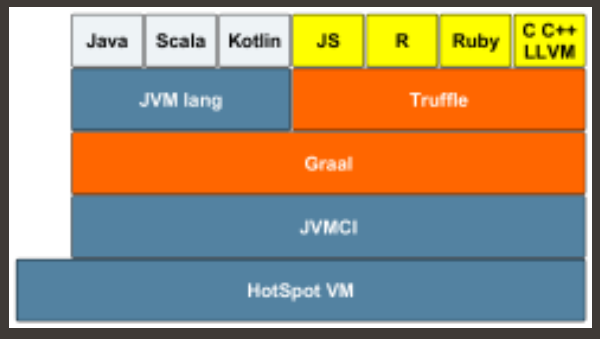


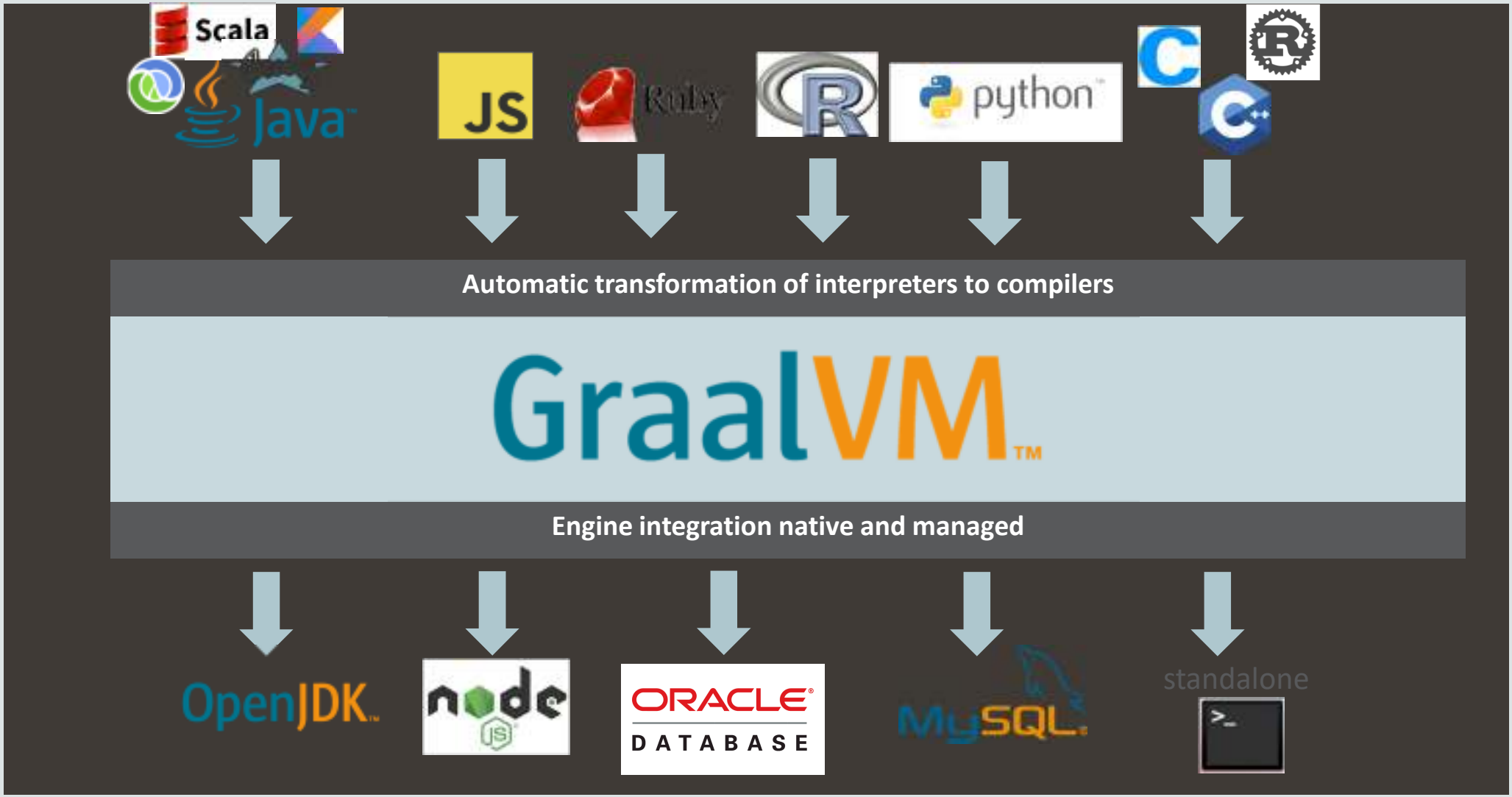
GraalVM - Polyglot (2)



GraalVM - Language Usability

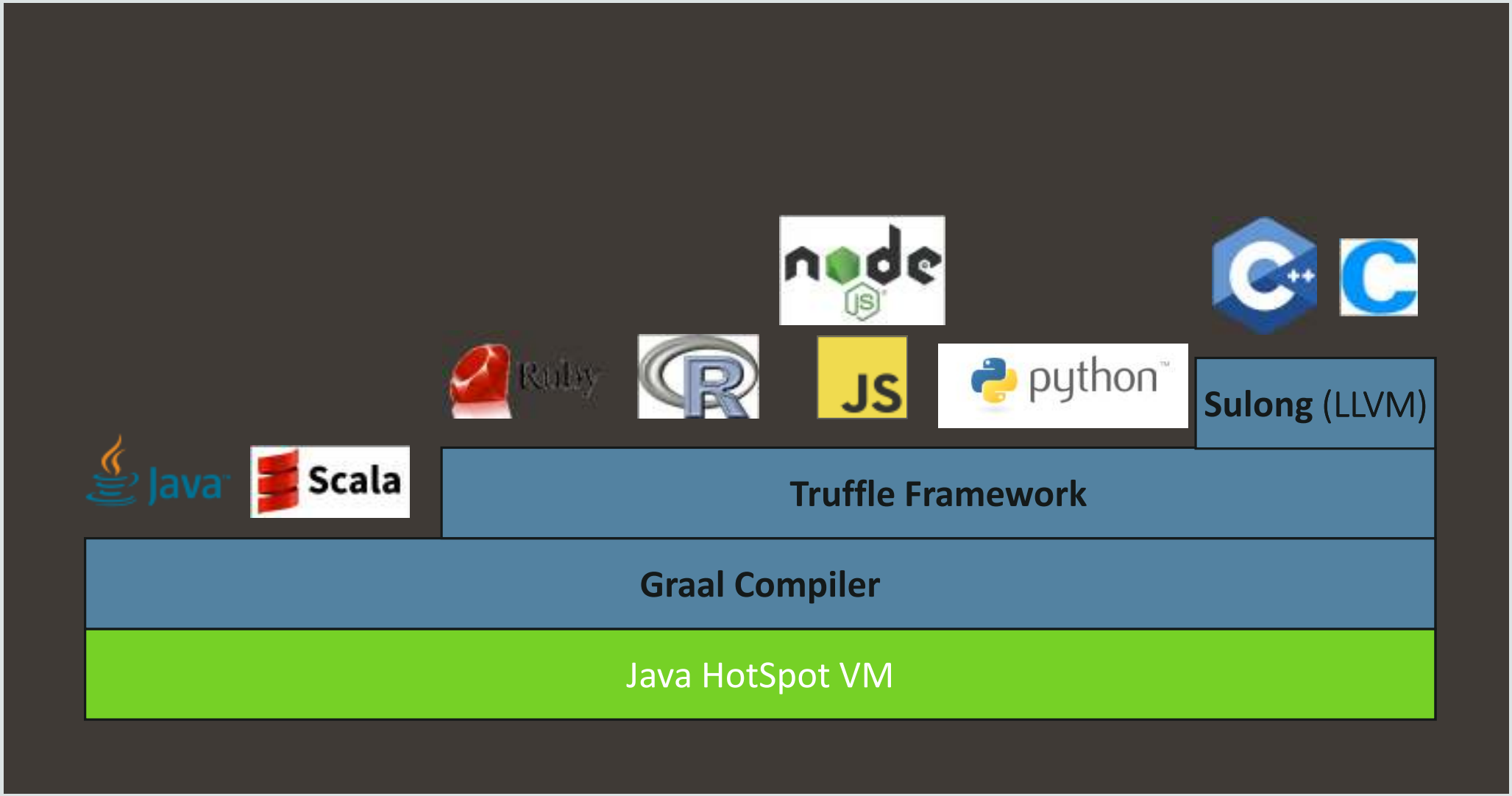
Production-Ready	Experimental	Visionary
Java	Ruby	Python
Scala, Groovy, Kotlin	R	VSCode Plugin
JavaScript	LLVM Tool Chain	GPU Integration
Node.js		WebAssembly
Native Image		LLVM Backend
VisualVM		





Graal Compiler

Java HotSpot VM

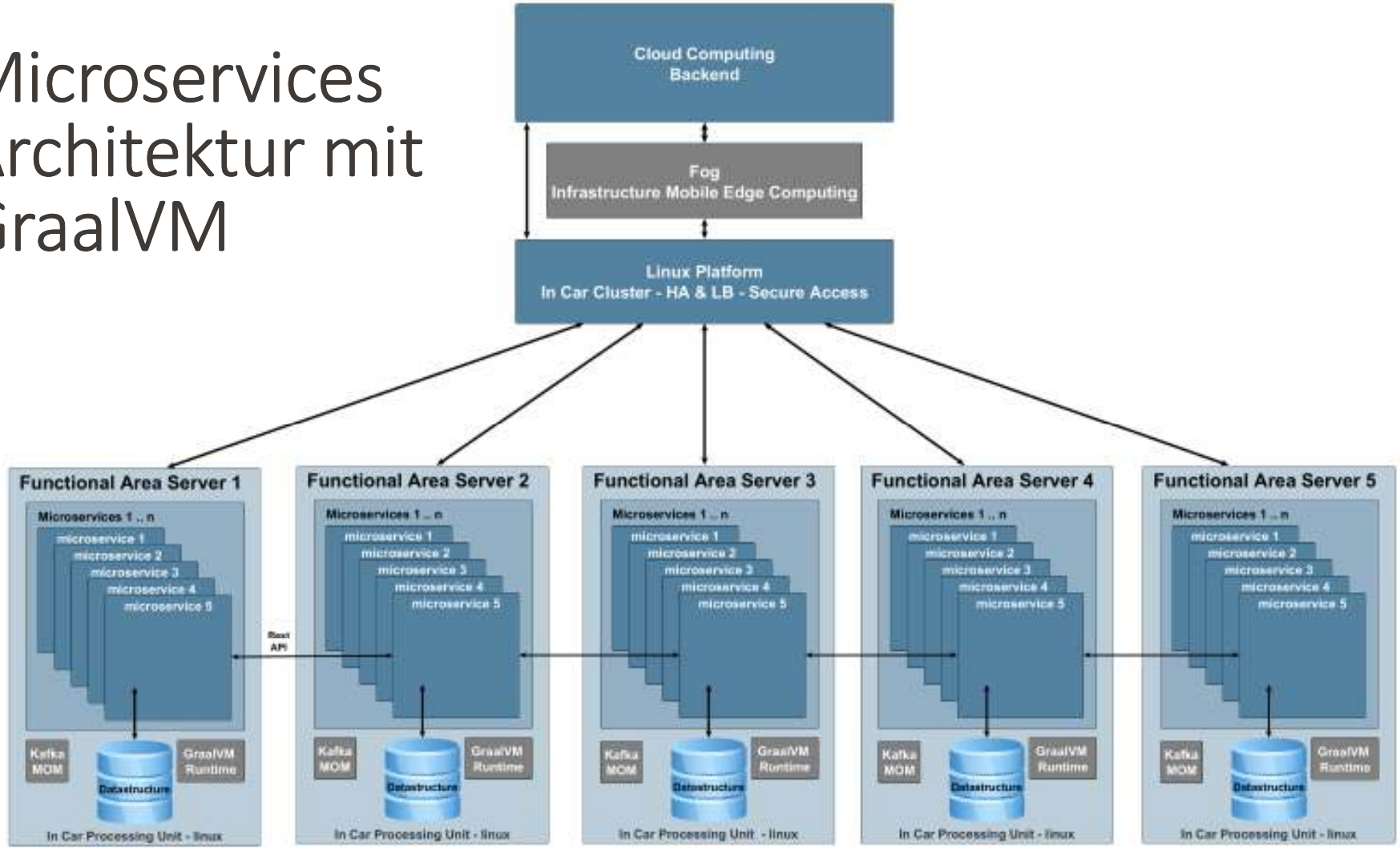


GraalVM - Top 10 Einsatzgebiete

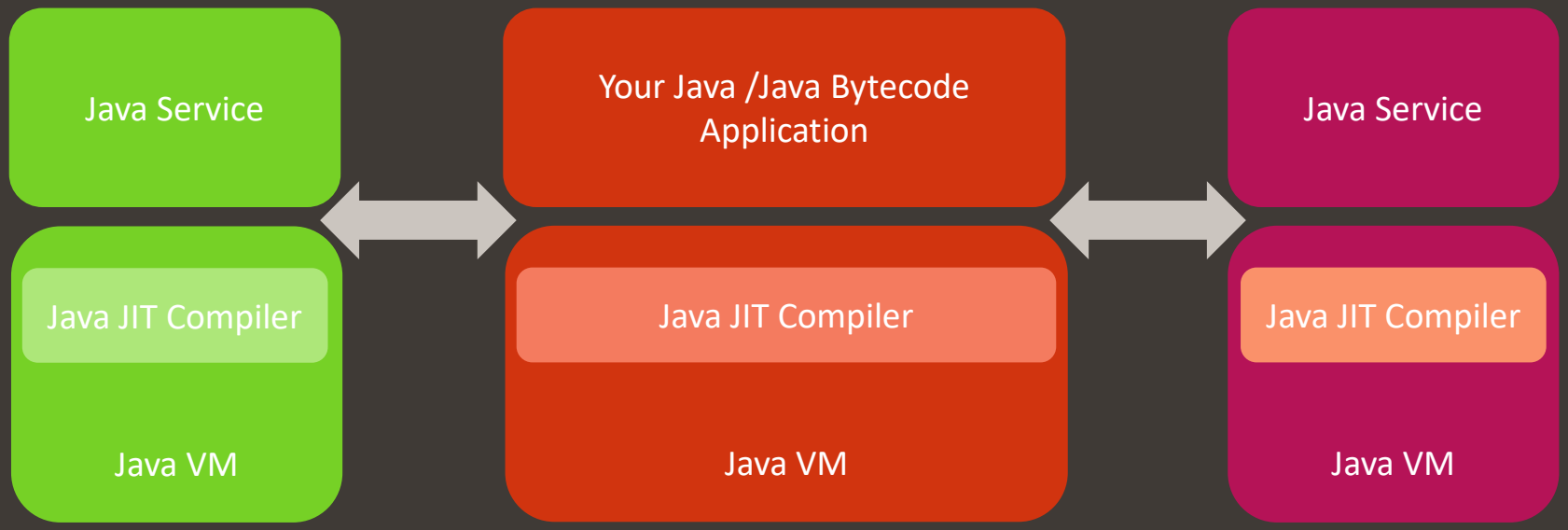
- Test your applications with GraalVM
  - Documentation and downloads
    http://www.graalvm.org
- Connect your technology with GraalVM
  - Integrate GraalVM into your application
  - Run your own programming language or DSL
  - Build language-agnostic tools
- Performance – Native Image
  - Startup time 20ms
  - Memory consumption less than 20MB

1. High-performance modern Java
2. Low-footprint, fast-startup Java
3. Combine JavaScript, Java, Ruby, and R
4. Run native languages on the JVM
5. Tools that work across all languages
6. Extend a JVM-based application
7. Extend a native application
8. Java code as a native library
9. Polyglot in the database
10. Create your own language

Microservices Architektur mit GraalVM



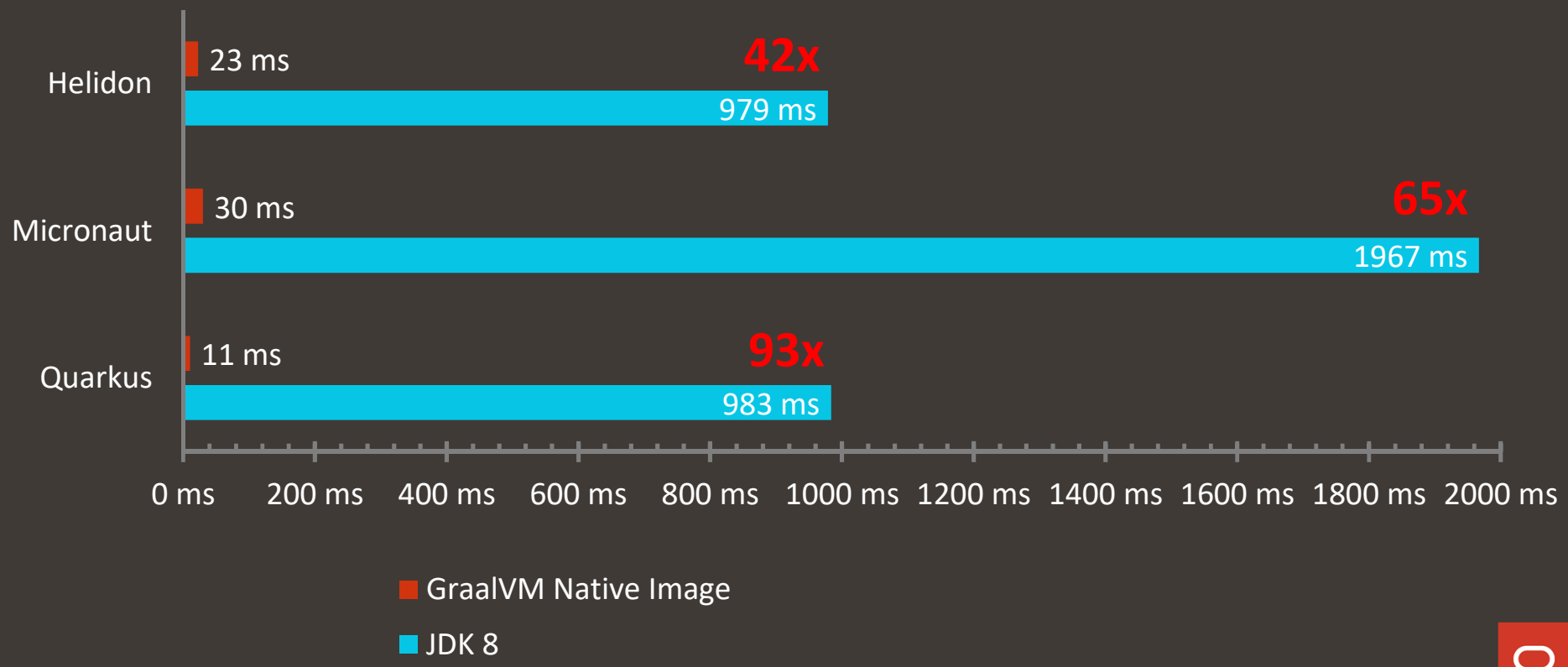
What GraalVM is for Microservices and Cloud Runtime



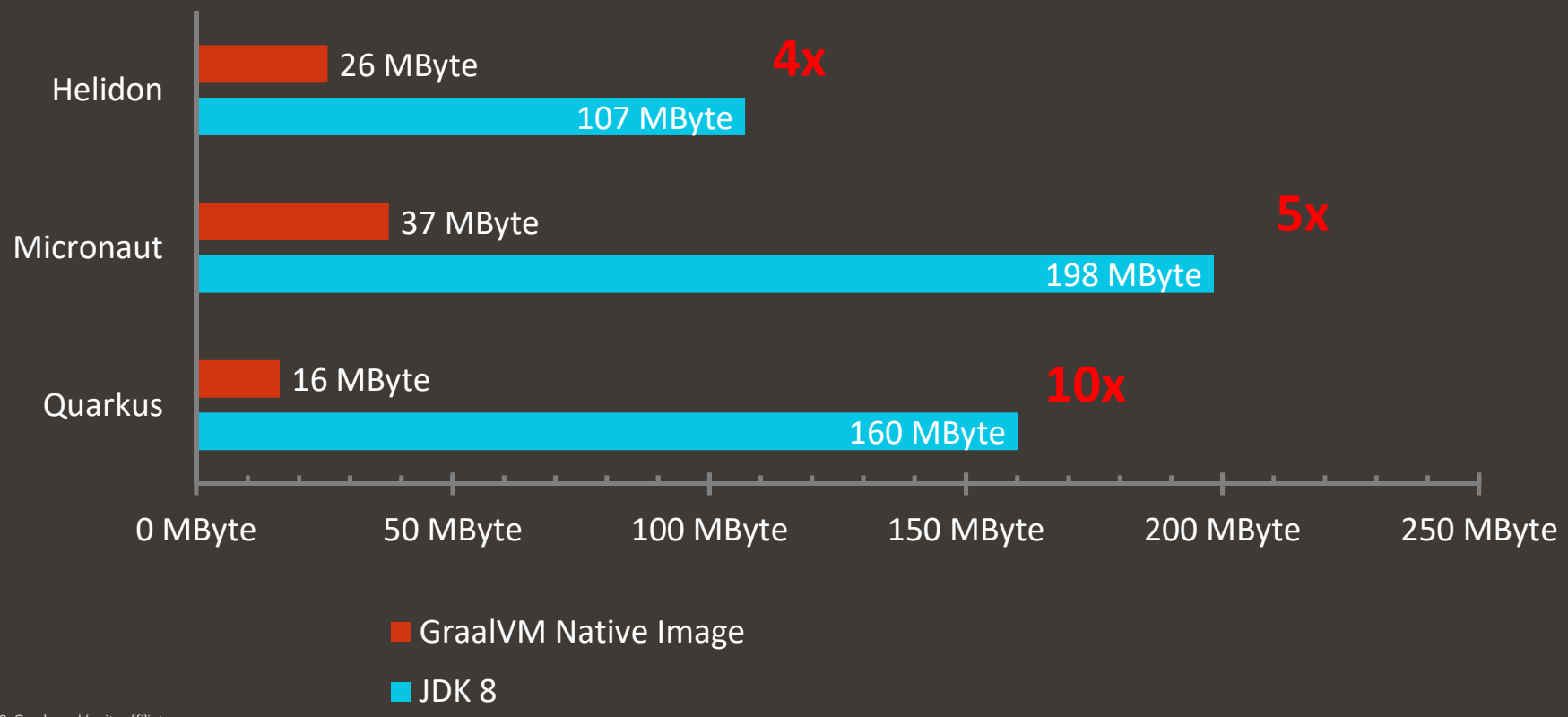
What GraalVM is for Microservices and Cloud Runtime

Up to 5x Less Memory
100x Faster Startup

Cloud Services – Startup Time



Cloud Services – Memory Footprint



GraalVM Performance

GraalVM Performance on Java

High Performance

Your Java /Java Bytecode Application

Java JIT Compiler

Java VM

GraalVM Performance on Java

32% Faster Execution on average

Your Java /Java Bytecode Application

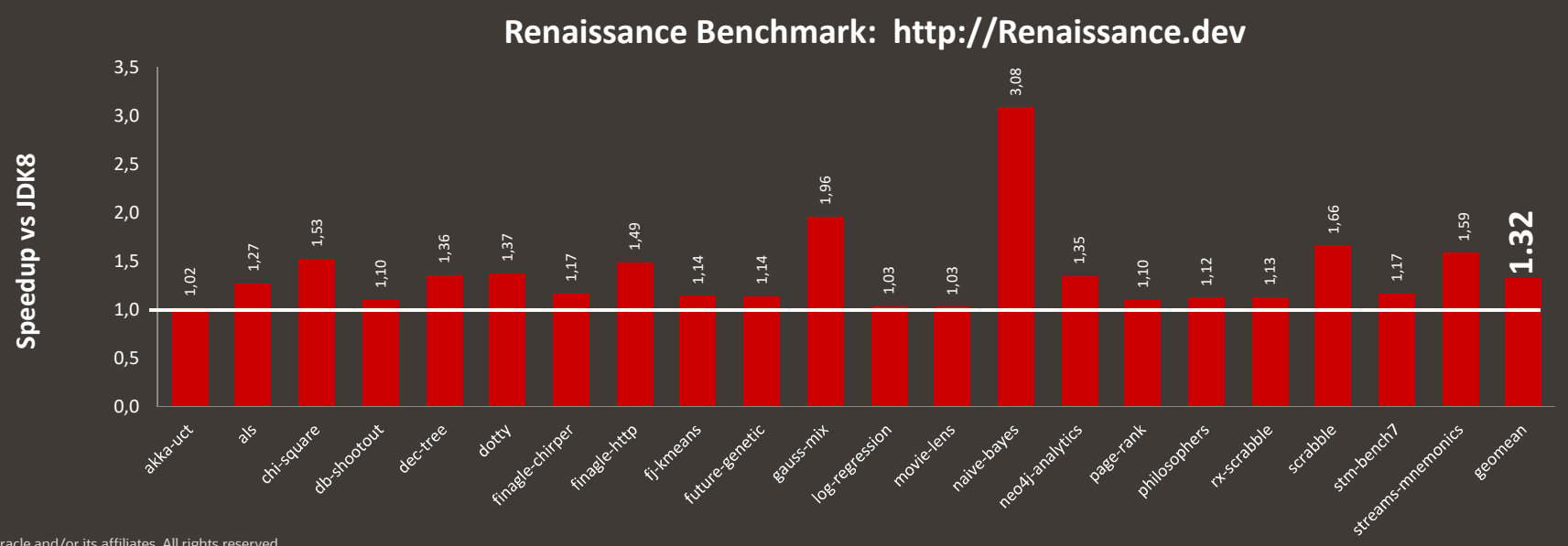
GraalVM JIT Compiler

Java VM

Better Performance: Java

Oracle GraalVM Enterprise Edition speeds up Java applications by 32% on average.

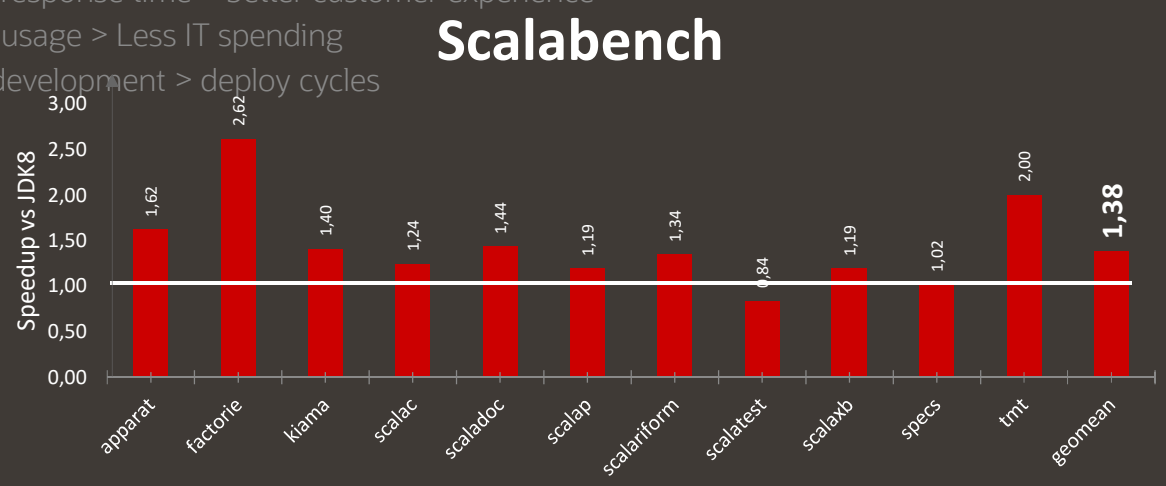
- Renaissance is the best benchmark for Java. It represents large, real-world Java applications.
- GraalVM Enterprise hast 62 optimizations algorithms that optimizes your existing Java Code while running on GraalVM
- For some time of Workloads the performance increase can be even bigger



Better Performance: Scala

- Implementing GraalVM Enterprise for Scala, customers enjoy even higher performance improvement (average of 38%).
- A 38% performance improvement translates into:

  Shorter application response time > better customer experience
  Less memory/CPU usage > Less IT spending
  Faster build time - development > deploy cycles



GraalVM Downloads

GraalVM Downloads

Community Edition

GraalVM Community is available for free for any use. It is built from the GraalVM sources available on GitHub. We provide pre-built binaries for Linux/X86, Linux/ARM, macOS, and Windows platforms on x86 64-bit systems. Support for the Windows and Linux/ARM platforms, and the Python, Ruby and R languages is experimental.

Note
GraalVM Community Edition contains significant technology from other projects including OpenJDK and Node.js which are not maintained by the GraalVM community. GraalVM Enterprise Edition is recommended for production applications.

Enterprise Edition

GraalVM Enterprise provides additional performance, security, and scalability relevant for running applications in production. You can get a version of GraalVM Enterprise that is free for evaluation and developing new applications via the Oracle Technology Network (OTN), or a commercially licensed version for production use via the Oracle Store.

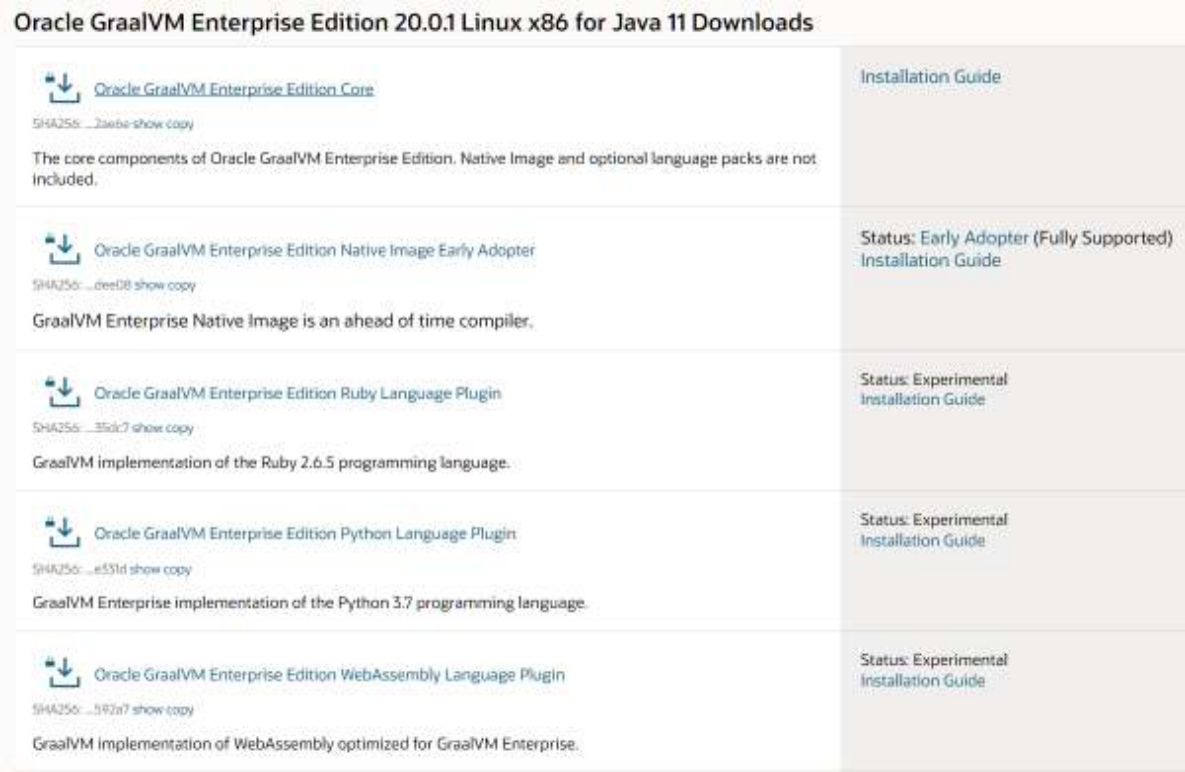
We provide binaries for Linux, macOS, and Windows platforms on x86 64-bit systems. Support for the Windows and Linux/ARM platforms, and the Python, Ruby and R languages is experimental.

GraalVM – Downloads

GraalVM is distributed as ***Community Edition*** and ***Enterprise Edition***. Listed below are bundles available:

- **GraalVM EE 20.0.1 based on Oracle Java 8u251**
- GraalVM CE 20.0.0 based on OpenJDK 8u242
- **GraalVM EE 20.0.1 based on Oracle Java 11.0.7**
- GraalVM CE 20.0.0 based on OpenJDK 11.0.6

- **OS: Linux, macOS, Windows**
- https://www.graalvm.org/downloads/

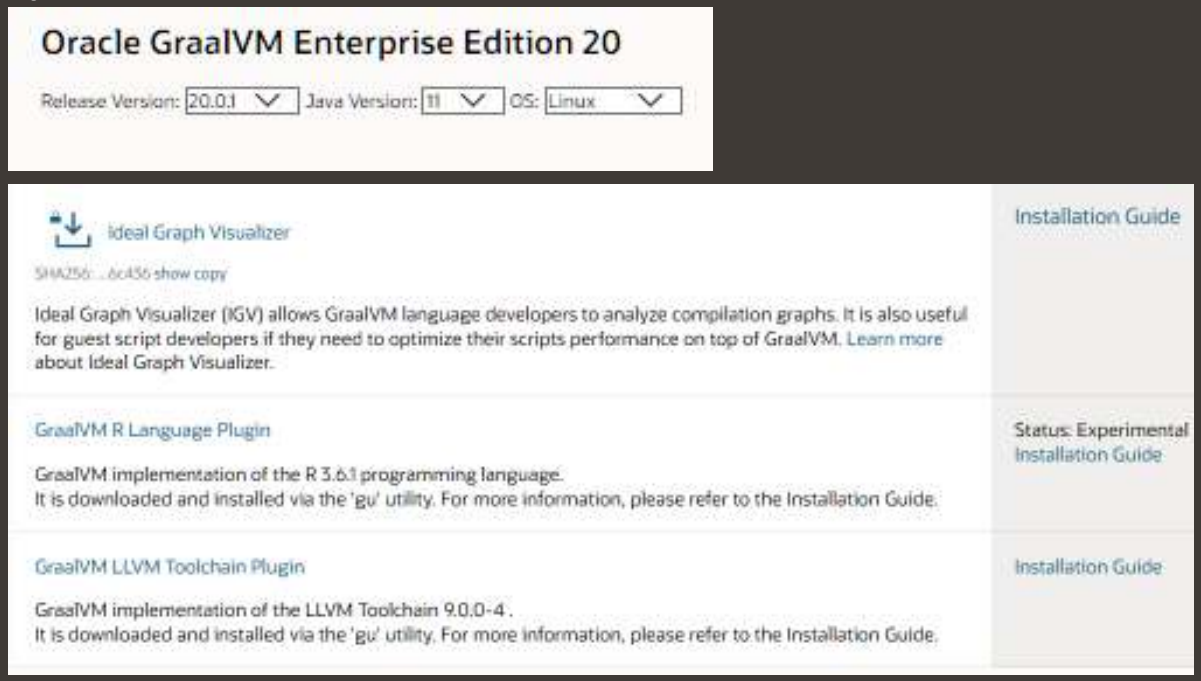


GraalVM – Downloads

GraalVM is distributed as ***Community Edition*** and ***Enterprise Edition***. Listed below are bundles available:

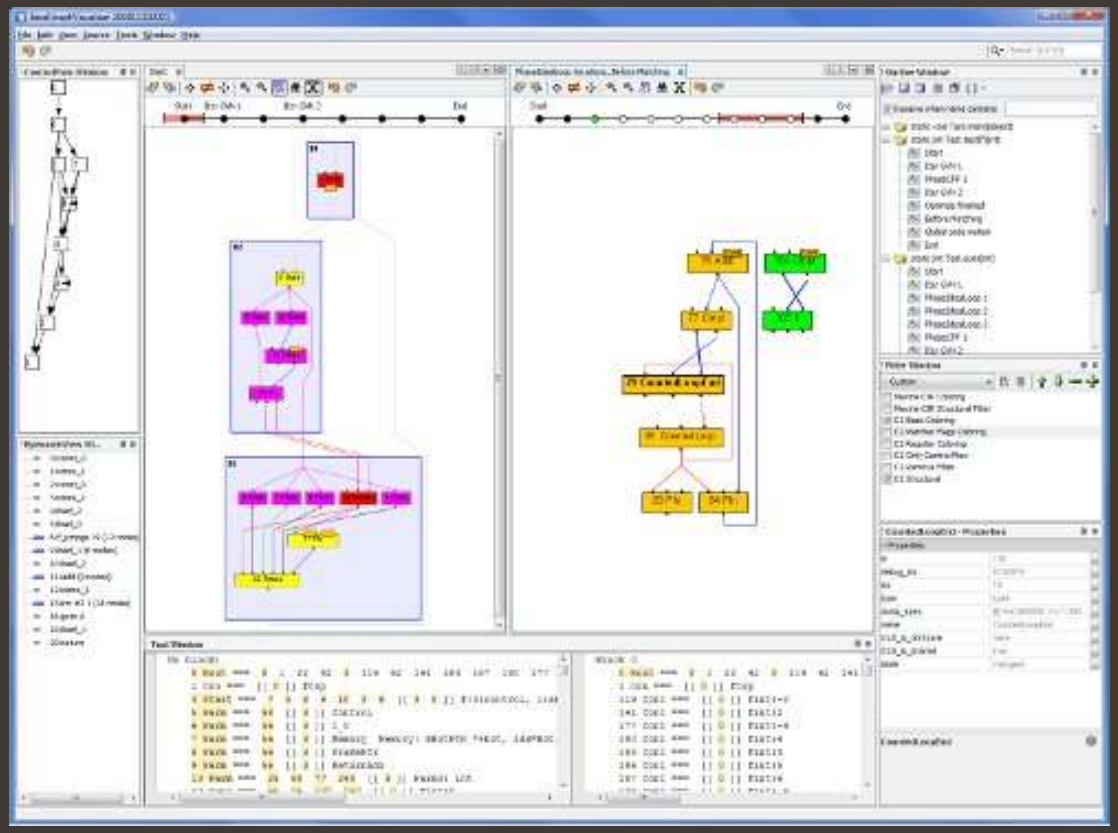
- **GraalVM EE 20.0.1 based on Oracle Java 8u251**
- GraalVM CE 20.0.0 based on OpenJDK 8u242
- **GraalVM EE 20.0.1 based on Oracle Java 11.0.7**
- GraalVM CE 20.0.0 based on OpenJDK 11.0.6

- **OS: Linux, macOS, Windows**
- https://www.graalvm.org/downloads/



GraalVM – Ideal Graph Visualizer

https://www.oracle.com/downloads/graalvm-downloads.html



- Ideal Graph Visualizer (IGV) allows GraalVM language developers to analyze compilation graphs
- It is also useful for guest script developers if they need to optimize their scripts performance on top of GraalVM
- IGV is no longer a part of GraalVM distribution

cmd.exe

```
C:\graalvm\bin>java -version
java version "11.0.7" 2020-04-14 LTS
Java(TM) SE Runtime Environment GraalVM EE 20.0.1 (build 11.0.7+8-LTS-jvmci-20.0-b04)
Java HotSpot(TM) 64-Bit Server VM GraalVM EE 20.0.1 (build 11.0.7+8-LTS-jvmci-20.0-b04, mixed mode, sharing)
```

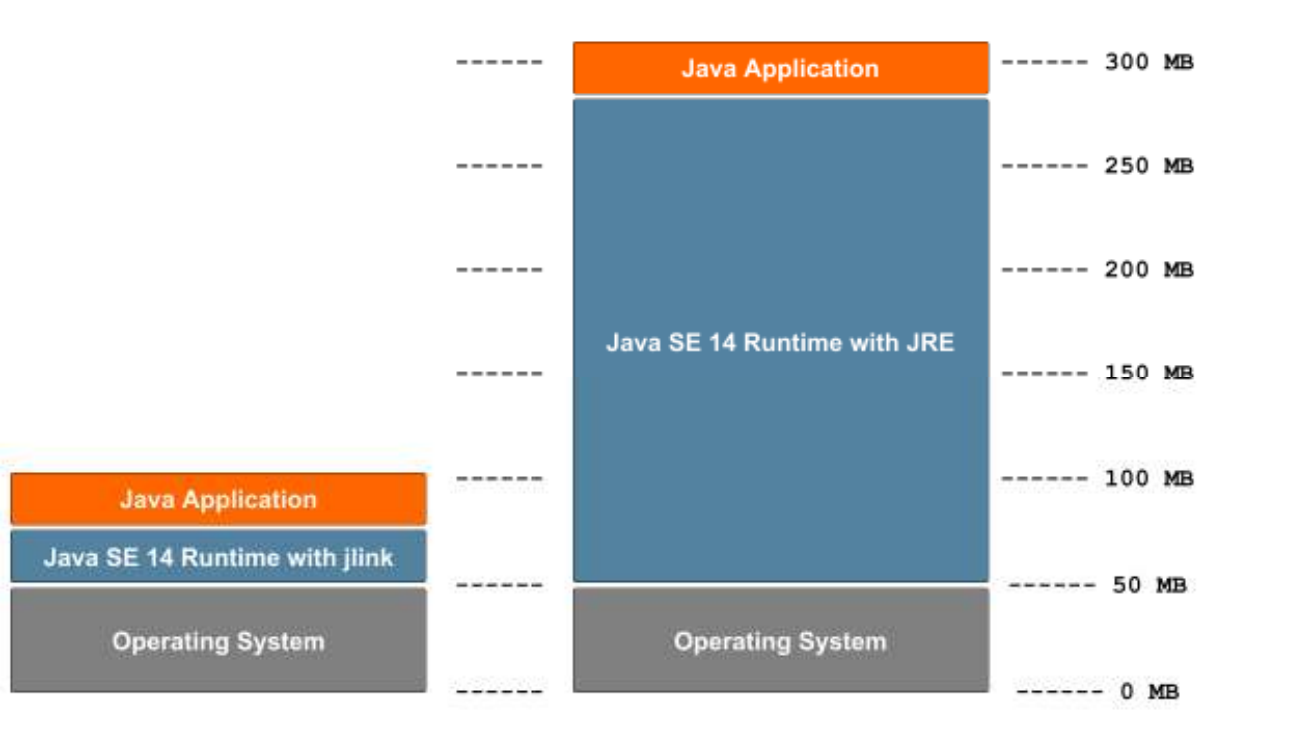
cmd.exe

```
C:\graalvm\bin>javap -c HelloWolfgang.class
Compiled from "HelloWolfgang.java"
public class HelloWolfgang {
  public HelloWolfgang();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String Hello Wolfgang!
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: invokestatic  #5                  // Method java/time/LocalDate.now:()Ljava/time/LocalDate;
      11: astore_1
      12: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
      15: aload_1
      16: invokedynamic #6,  0              // InvokeDynamic #0:makeConcatWithConstants:(Ljava/time/LocalDate;)Ljava/lang/String;
      21: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      24: invokestatic  #7                  // Method java/time/LocalTime.now:()Ljava/time/LocalTime;
      27: astore_2
      28: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
      31: aload_2
      32: invokedynamic #8,  0              // InvokeDynamic #1:makeConcatWithConstants:(Ljava/time/LocalTime;)Ljava/lang/String;
      37: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      40: return
}
```

GraalVM Footprint

Run smaller images – Remove parts of Java you don't use



Java SE 14 with jlink vs. Java SE 14 with JRE

New Project: Leyden (1)

- New Project, Leyden, whose primary goal will be to address the long-term pain points of Java's slow startup time, slow time to peak performance, and large footprint.
- Leyden will address these pain points by introducing a concept of _static images_ to the Java Platform, and to the JDK.
  - A static image is a standalone program, derived from an application, which runs that application -- and no other.
  - A static image is a closed world: It cannot load classes from outside the image, nor can it spin new bytecodes at run time.

New Project: Leyden (2)

- Project Leyden will take inspiration from past efforts to explore this space, including the GNU Compiler for Java and the Native Image feature of GraalVM. Leyden will add static images to the Java Platform Specification, and we expect that GraalVM will evolve to implement that specification. Developers who use only the standard, specified static-image feature will then be able to switch with ease between Leyden (in the JDK), Native Image (in GraalVM), and whatever other conforming implementations may arise, choosing amongst tradeoffs of compile time, startup time, and image size.

- We do not intend to implement Leyden by merging the Native Image code from GraalVM into the JDK. Leyden will, rather, be based upon existing components in the JDK such as the HotSpot JVM, the `jaotc` ahead-of-time compiler, application class-data sharing, and the `jlink` linking tool.

Run smaller images – Java

- GraalVM compiles Java source to a single native binary
- Tiny image sizes
- Low VM overhead

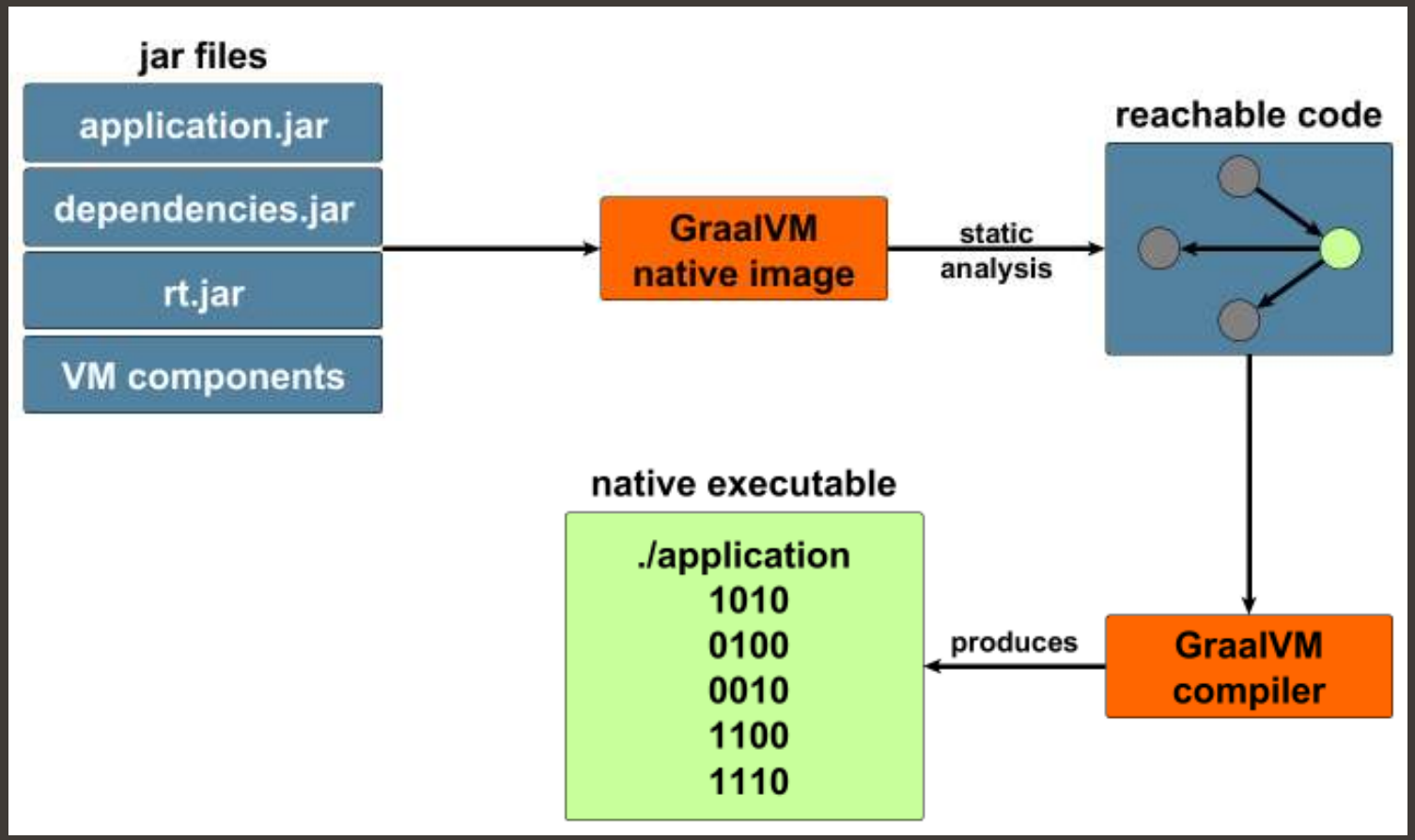
GraalVM Native Image

GraalVM Native Image

- Makes your Java code ready for the Cloud
- Instant startup
- Low memory footprint
- Single self-contained binary

```
$ javac HelloWorld.java
$ time java HelloWorld
user 0.070s
$ native-image HelloWorld
$ time ./helloworld
user 0.005s
```

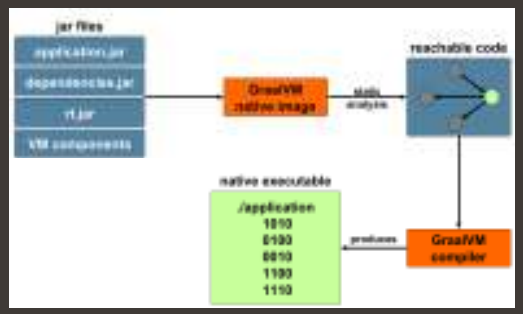
GraalVM Native Image



GraalVM Native Image Generation Options

https://www.graalvm.org/docs/reference-manual/native-image/

The **`native-image`** command line needs to provide the class path for all classes using the familiar option from the **`java`** launcher: **`-cp`** is followed by a list of directories or .jar files, separated by **`:`**. The name of the class containing the **`main`** method is the last argument; or you can use **`-jar`** and provide a **`.jar`** file that specifies the **`main`** method in its manifest.



The syntax of the native-image command is:

- **`native-image [options] class`** to build an executable file for a class in the current working directory. Invoking it executes the native-compiled code of that class.

- **`native-image [options] -jar`** `jarfile` to build an image for a jar file.

GraalVM Native Image Generation Options

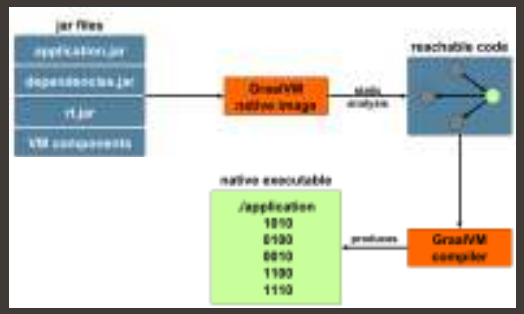
https://www.graalvm.org/docs/reference-manual/native-image/

Oracle GraalVM Enterprise Edition Native Image Early Adopter

GraalVM Native Image is ahead of time compilation functionality and is offered as an early adopter preview.

Oracle GraalVM Enterprise Edition Native Image preview for Linux (19.2.1)

(SHA1 Hash - 46356d73233bb0d03c9322bf4ad376f17598d20b)




```
wolf@wolf-ThinkPad-T450:~/graal/graalvm-ee-19.2.1$ gu -L install '/home/wolf/Downloads/native-image-installable-svm-svmee-linux-amd64-19.2.1.jar'
Processing component archive: /home/wolf/Downloads/native-image-installable-svm-svmee-linux-amd64-19.2.1.jar
Installing new component: Native Image (org.graalvm.native-image, version 19.2.1)
wolf@wolf-ThinkPad-T450:~/graal/graalvm-ee-19.2.1$
```

```
$ /src
$ /home/wolf/graal/graalvm-ee-19.2.1/jre/bin/native-image -cp HelloWorld.class
```

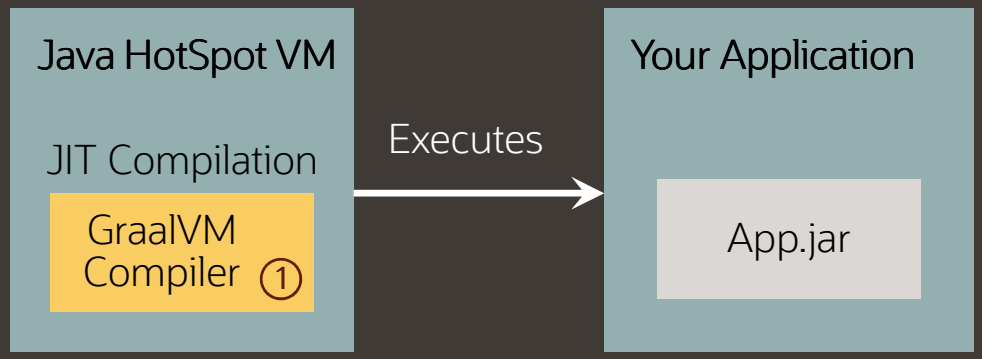
Closed World Assumption

- The points-to analysis needs to see all bytecode
  - Otherwise aggressive AOT optimizations are not possible
  - Otherwise unused classes, methods, and fields cannot be removed
  - Otherwise a class loader / bytecode interpreter is necessary at run time
- Dynamic parts of Java require configuration at build time
  - Reflection, JNI, Proxy, resources, …
  - That's what this talk is about
- No loading of new classes at run time

Image Heap

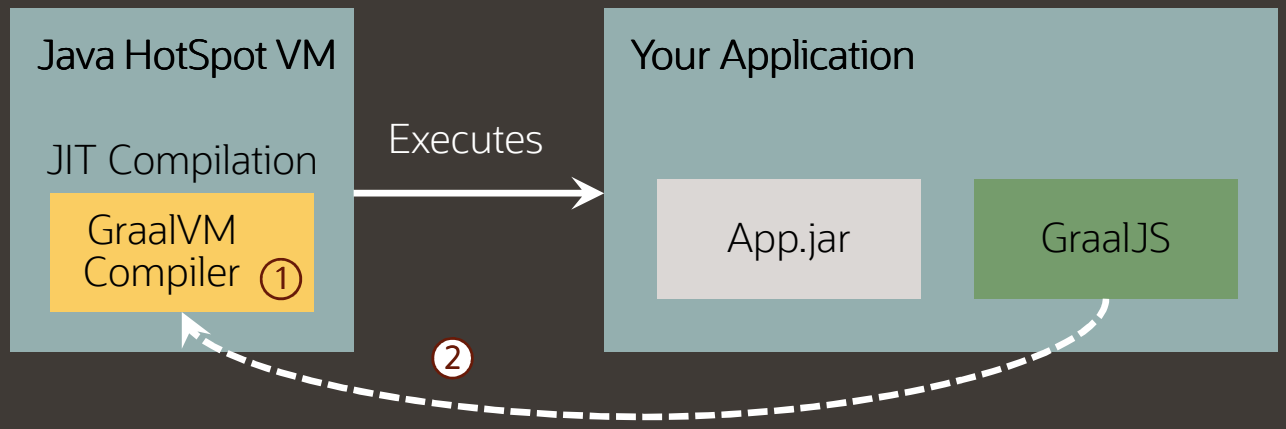
- Execution at run time starts with an initial heap: the “image heap”
  - Objects are allocated in the Java VM that runs the image generator
  - Heap snapshotting gathers all objects that are reachable at run time
- Do things once at build time instead at every application startup
  - Class initializers, initializers for static and static final fields
  - Explicit code that is part of a so-called “Feature”
- Examples for objects in the image heap
  - java.lang.Class objects, Enum constants

One Compiler, Many Configurations



1 Compiler configured for just-in-time compilation inside the Java HotSpot VM

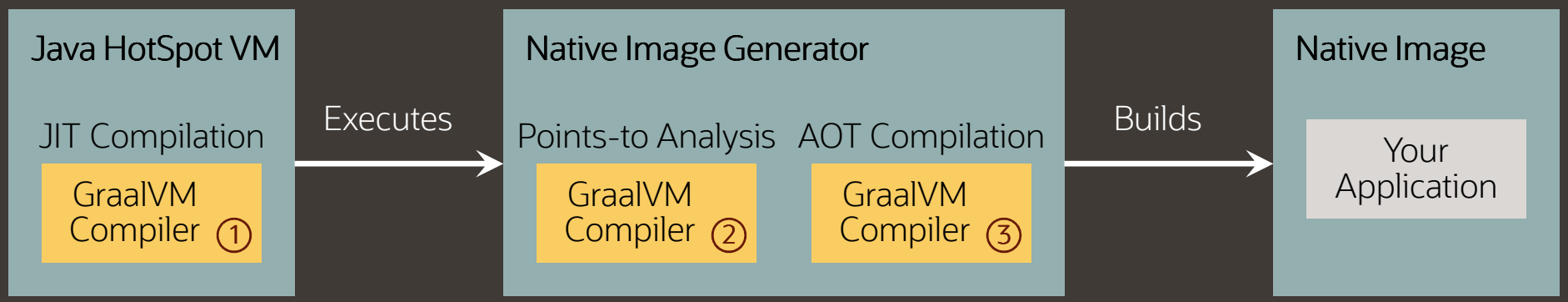
One Compiler, Many Configurations



1. Compiler configured for just-in-time compilation inside the Java HotSpot VM
2. Compiler also used for just-in-time compilation of JavaScript code

One Compiler, Many Configurations

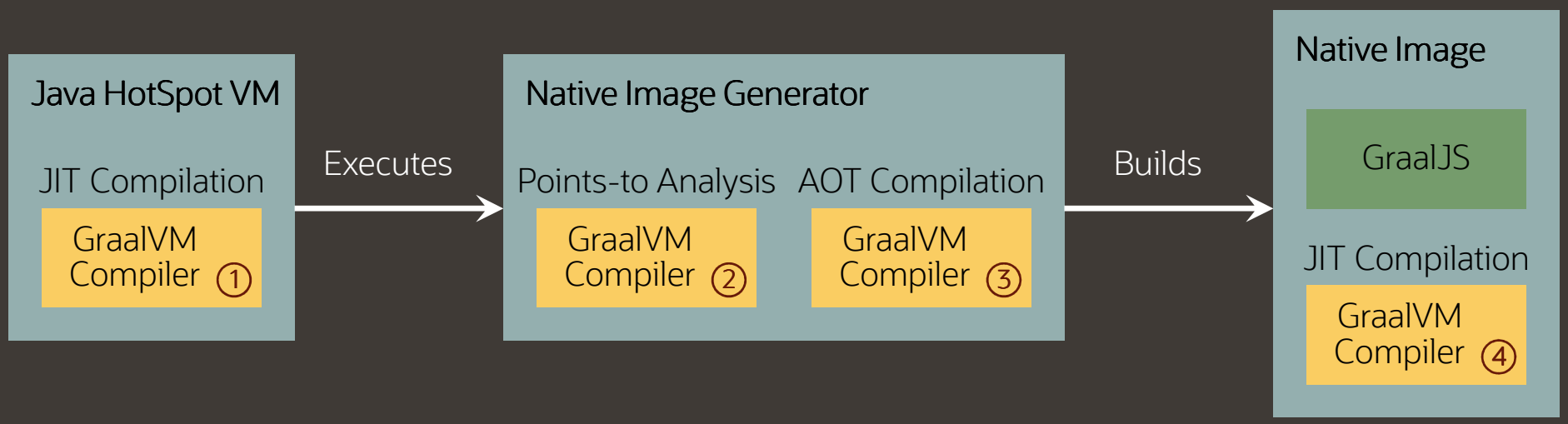
GraalVM



1. Compiler configured for just-in-time compilation inside the Java HotSpot VM
2. Compiler configured for static points-to analysis
3. Compiler configured for ahead-of-time compilation

One Compiler, Many Configurations

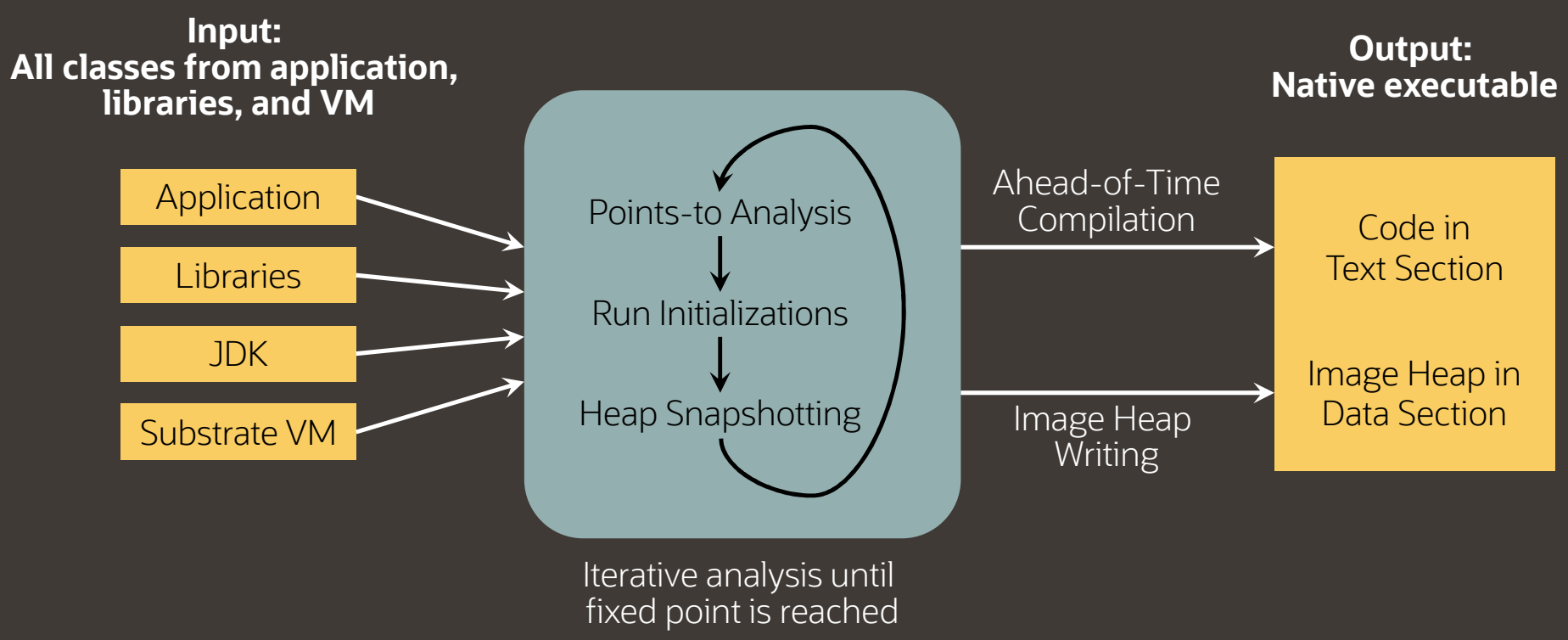
GraalVM



① Compiler configured for just-in-time compilation inside the Java HotSpot VM
② Compiler configured for static points-to analysis
③ Compiler configured for ahead-of-time compilation
④ Compiler configured for just-in-time compilation inside a Native Image

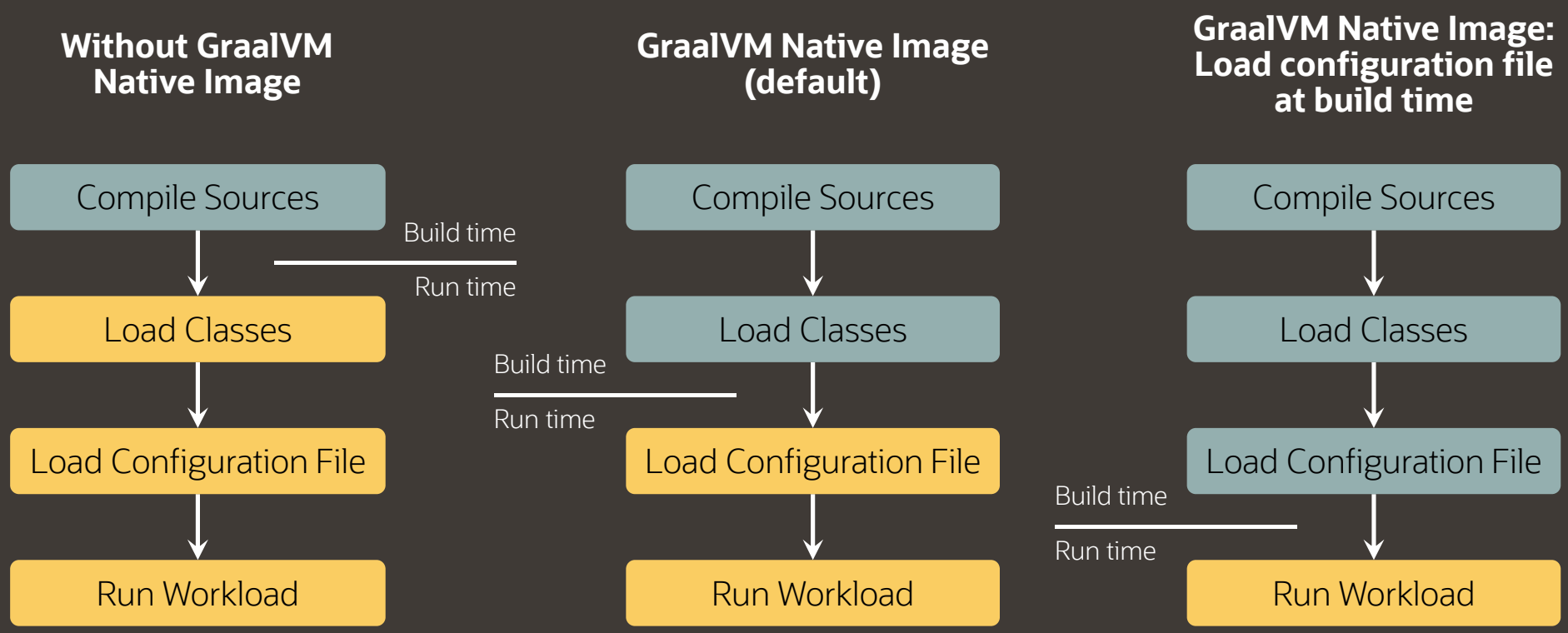
Native Image - Details

GraalVM



Benefits of the Image Heap

GraalVM



Roadmap

GraalVM Version Roadmap – Major Versions

We release new major versions of GraalVM every 3 months on a predictable schedule, always to the closest Tuesday to the 17th of the month of February, May, August, and November

Major releases become inactive once a new release is published

Only the last major release of the year continues to be updated for the full next year

GraalVM Version Roadmap - Critical Patch Updates

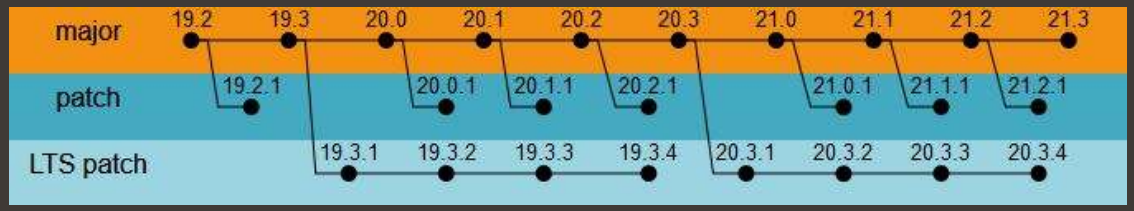
Critical Patch Updates (CPU) for GraalVM follow the schedule for all CPU releases of Oracle as described here.

The release happens quarterly always on the closest Tuesday to the 17th of the month of January, April, July, and October

All active releases receive patch updates

GraalVM Version Roadmap – Release Dates

Find below a graphical visualizations of the release roadmap and the dates and version numbers of upcoming releases



- Aug 20, 2019: 19.2
- Oct 15, 2019: 19.2.1 (CPU)
- Nov 19, 2019: 19.3
- Jan 14, 2020: 19.3.1 (CPU)
- Feb 18, 2020: 20.0
- Apr 14, 2020: 19.3.2 (CPU), 20.0.1 (CPU)
- May 19, 2020: 20.1
- Jul 14, 2020: 19.3.3 (CPU), 20.1.1 (CPU)
- Aug 18, 2020: 20.2
- Oct 20, 2020: 19.3.4 (CPU), 20.2.1 (CPU)
- Nov 17, 2020: 20.3
- Jan 19, 2021: 20.3.1 (CPU)
- Feb 16, 2021: 21.0
- Apr 20, 2021: 20.3.2 (CPU), 21.0.1 (CPU)
- May 18, 2021: 21.1
- Jul 20, 2021: 20.3.3 (CPU), 21.1.1 (CPU)
- Aug 17, 2021: 21.2
- Oct 19, 2021: 20.3.4 (CPU), 21.2.1 (CPU)
- Nov 16, 2021: 21.3

Summary

Building a universal VM is a community effort

- GraalVM is the new universal VM
  - Documentation and downloads:
    - http://www.graalvm.org
- Connect your technology with GraalVM
  - Integrate GraalVM into your application
  - Run your own programming language or DSL
  - Build language-agnostic tools
- Works well with open source projects
  - Eclipse Vert.x Tool-Kit, Fn Project, Gluon Client Plugin, Picocli Java-Command-Line-Parser
  - Helidon, Micronaut, Quarkus
- Features a native-image tool
  - Converts Java to native machine code using AOT
- Footprint – Native Image
  - Startup time 20ms
  - Memory consumption less than 20MB

https://jaxenter.de/java/graalvm-virtual-machine-java-oracle-91288

GraalVM

Danke!

Wolfgang.Weigend@oracle.com

Twitter: @wolflook

GraalVM™

https://jaxenter.de/java/graalvm-virtual-machine-java-oracle-91288