# What the CRaC...

Coordinated Restore at Checkpoint on the Java Virtual Machine

# ABOUTME.

Gerrit Grunwald | Developer Advocate | Azul

# SLIDES

# JAVA IS

# GREAT...

azul

# VIBRANT COMMUNITY...

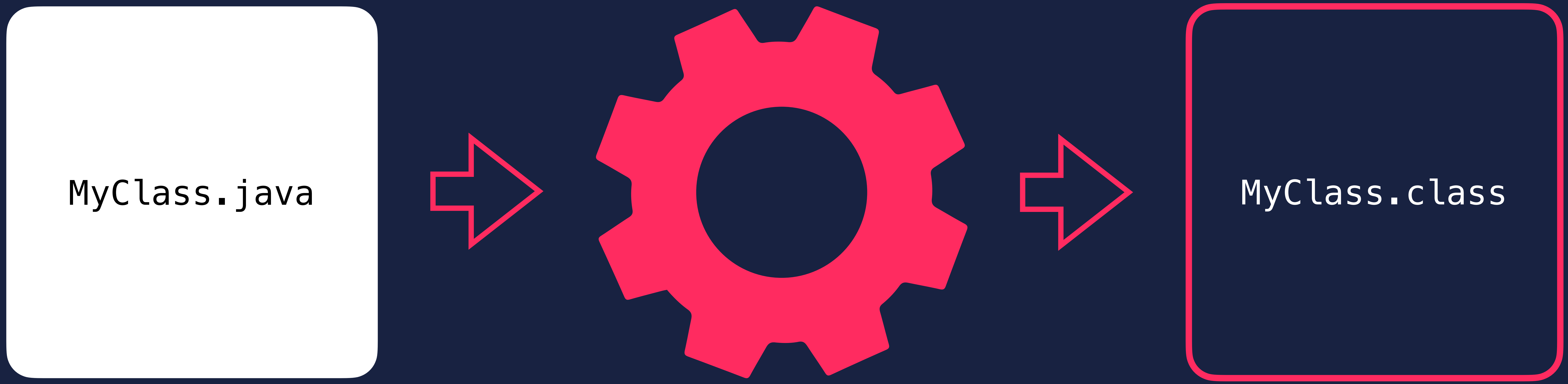
# HUNDREDS OF JUGs…

# THOUSANDS OF FOSS PROJECTS...

# JAVA VIRTUAL MACHINE

# HOW DOES IT WORK...

azul



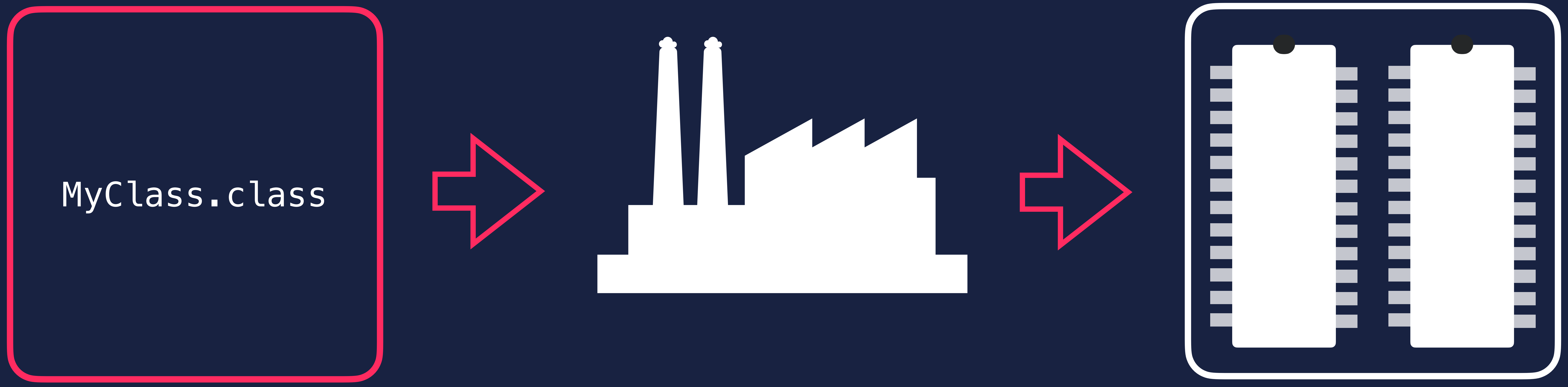
`MyClass.java`

SOURCE CODE

COMPILER

`MyClass.class`

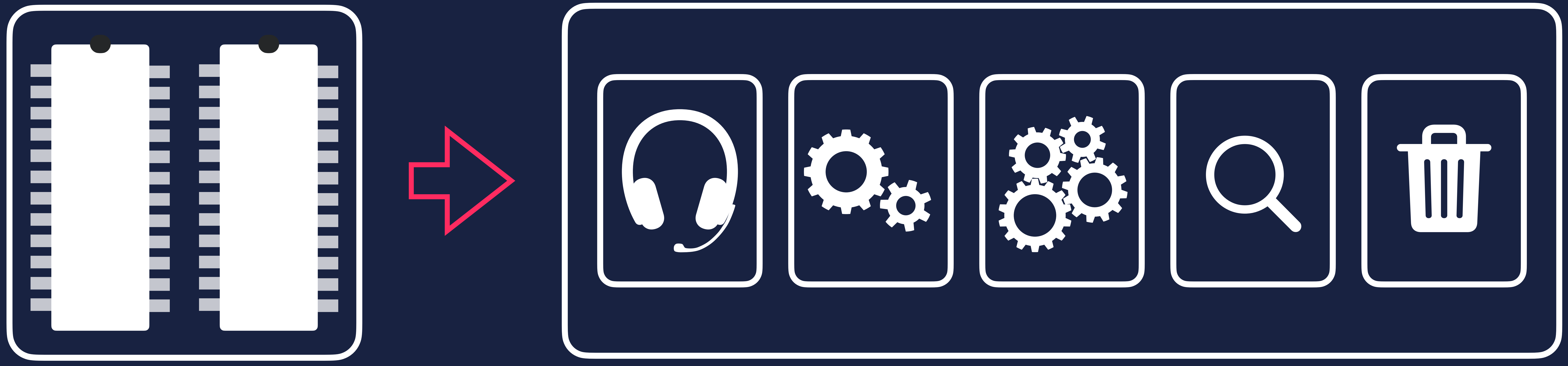
BYTE CODE



BYTE CODE

CLASS LOADER

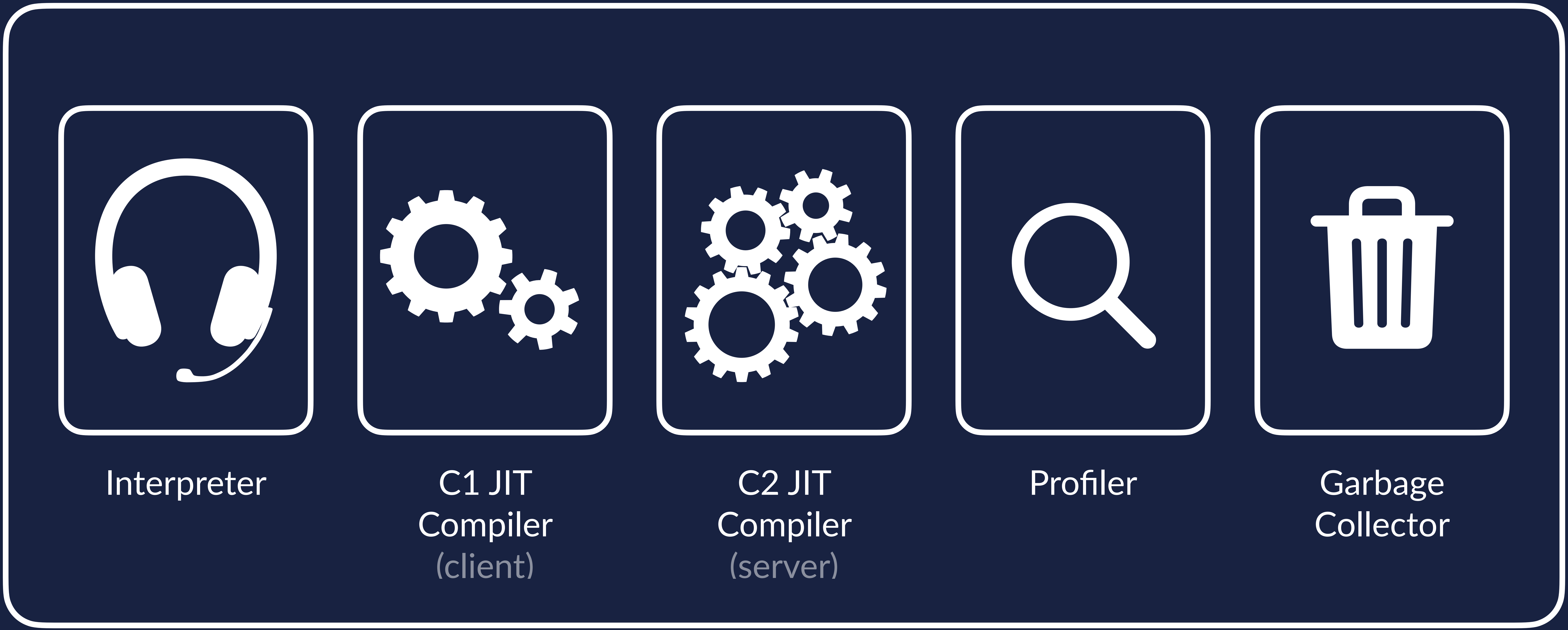
JVM MEMORY



JVM MEMORY

EXECUTION ENGINE

# EXECUTION ENGINE



Interpreter

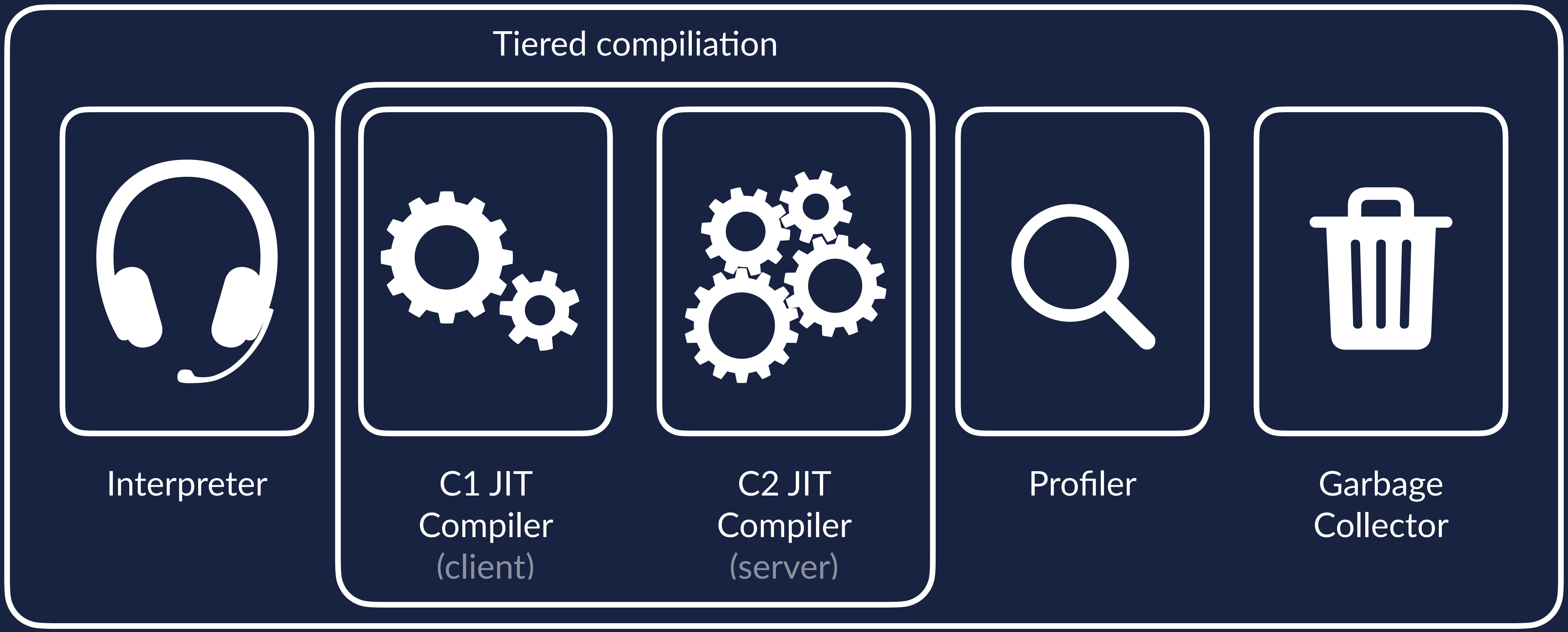
C1 JIT Compiler (client)

C2 JIT Compiler (server)

Profiler

Garbage Collector

# EXECUTION ENGINE



Tiered compilation

Interpreter

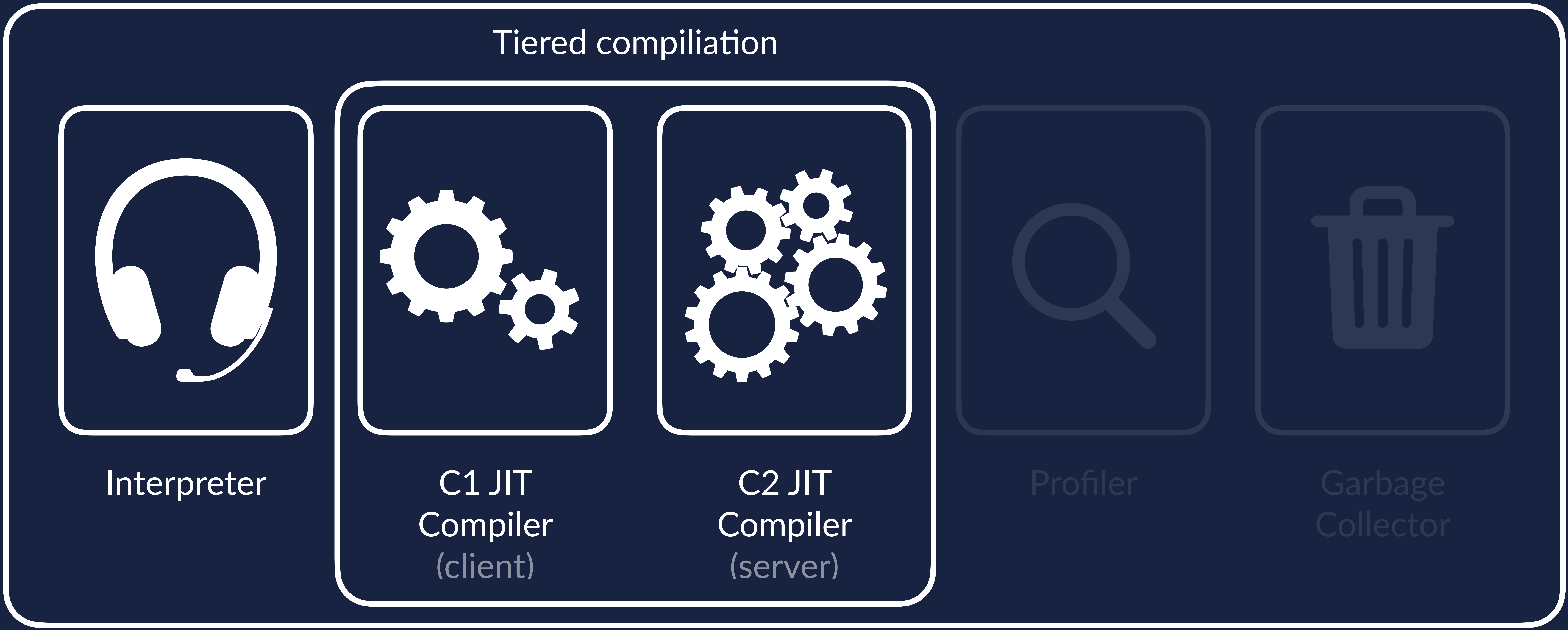
C1 JIT Compiler (client)

C2 JIT Compiler (server)

Profiler

Garbage Collector

# EXECUTION ENGINE



Tiered compiliation

Interpreter

C1 JIT Compiler (client)

C2 JIT Compiler (server)

Profiler

Garbage Collector

Converts ByteCode into instruction set of CPU

INTERPRETER

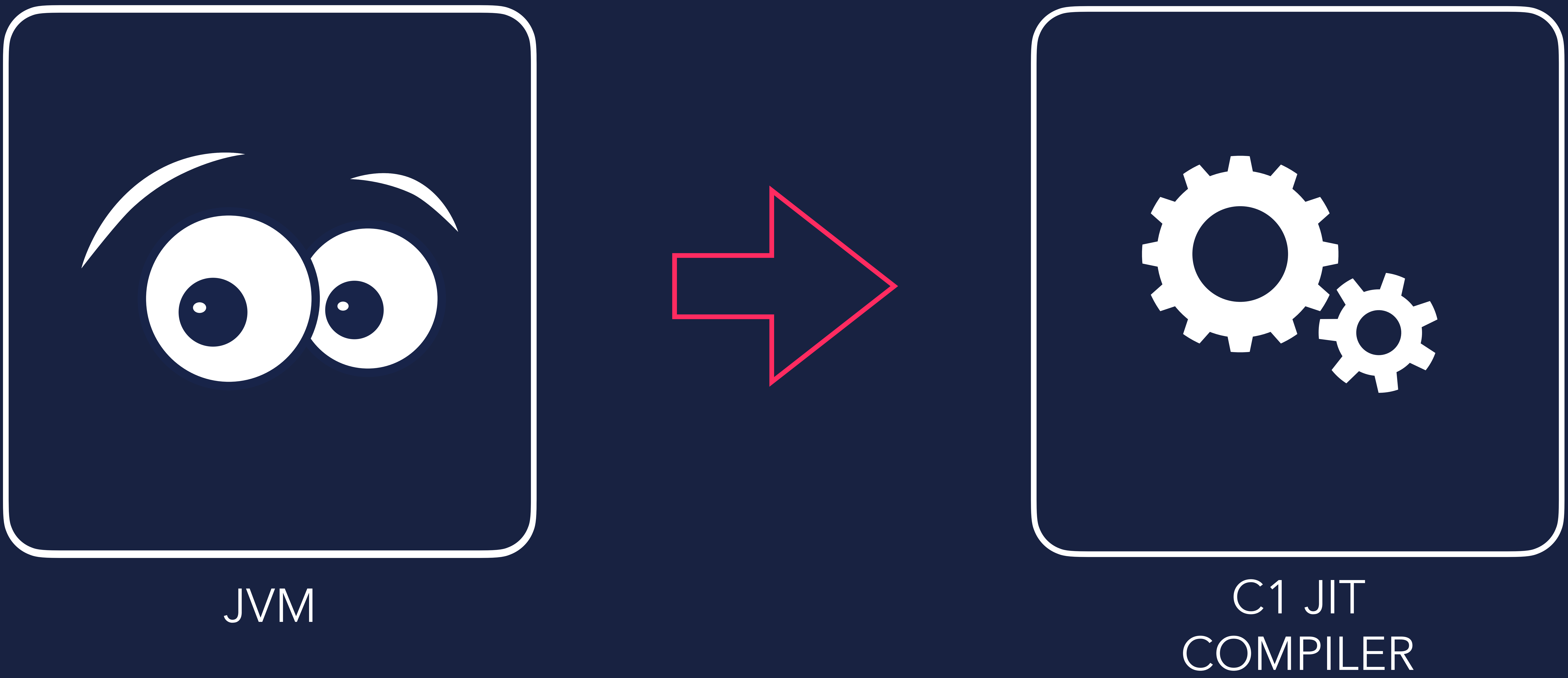
THRESHOLD REACHED

Detects hot spots by counting method calls

JVM

Pass the hot spot methods
to C1 JIT Compiler

Compiles code as quickly
as possible with low optimisation



JVM

C1 JIT
COMPILER

Compiles code as quickly as possible with low optimisation

C1 JIT COMPILER

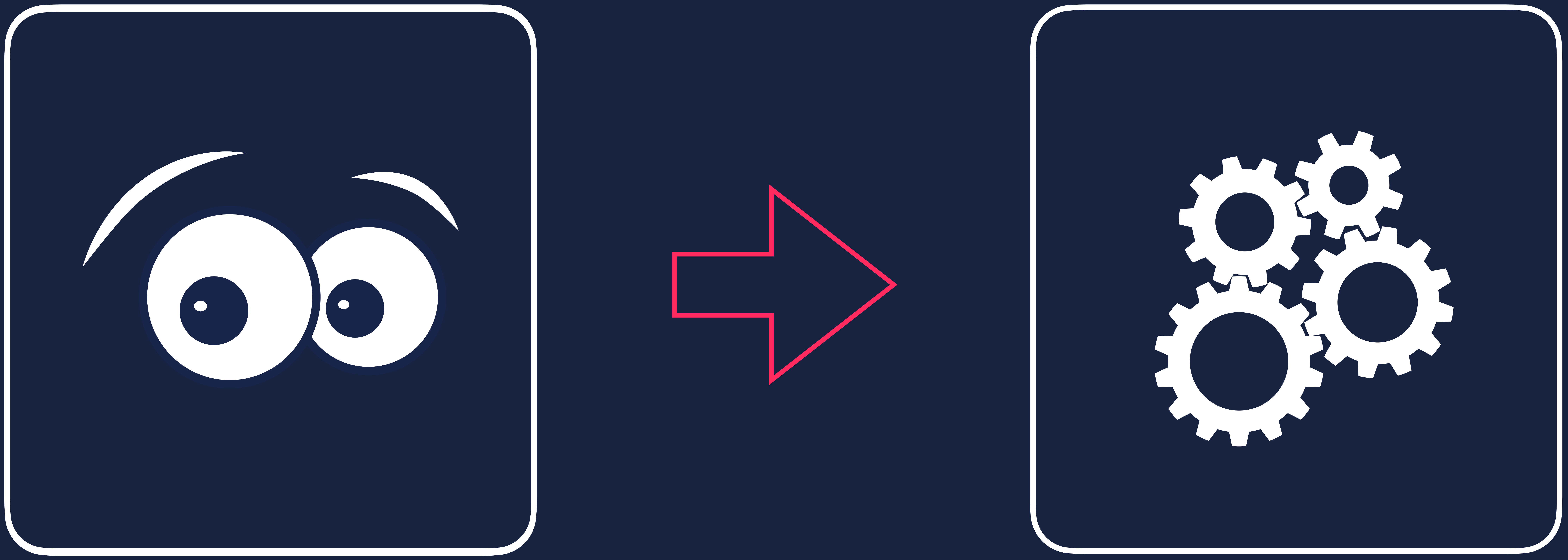
THRESHOLD REACHED

Profiles the running code (detecting hot code)

JVM

Pass the "hot" code to C2 JIT Compiler

Compiles code with best optimisation possible (slower)



JVM

C2 JIT COMPILER

# TIERED COMPILATION

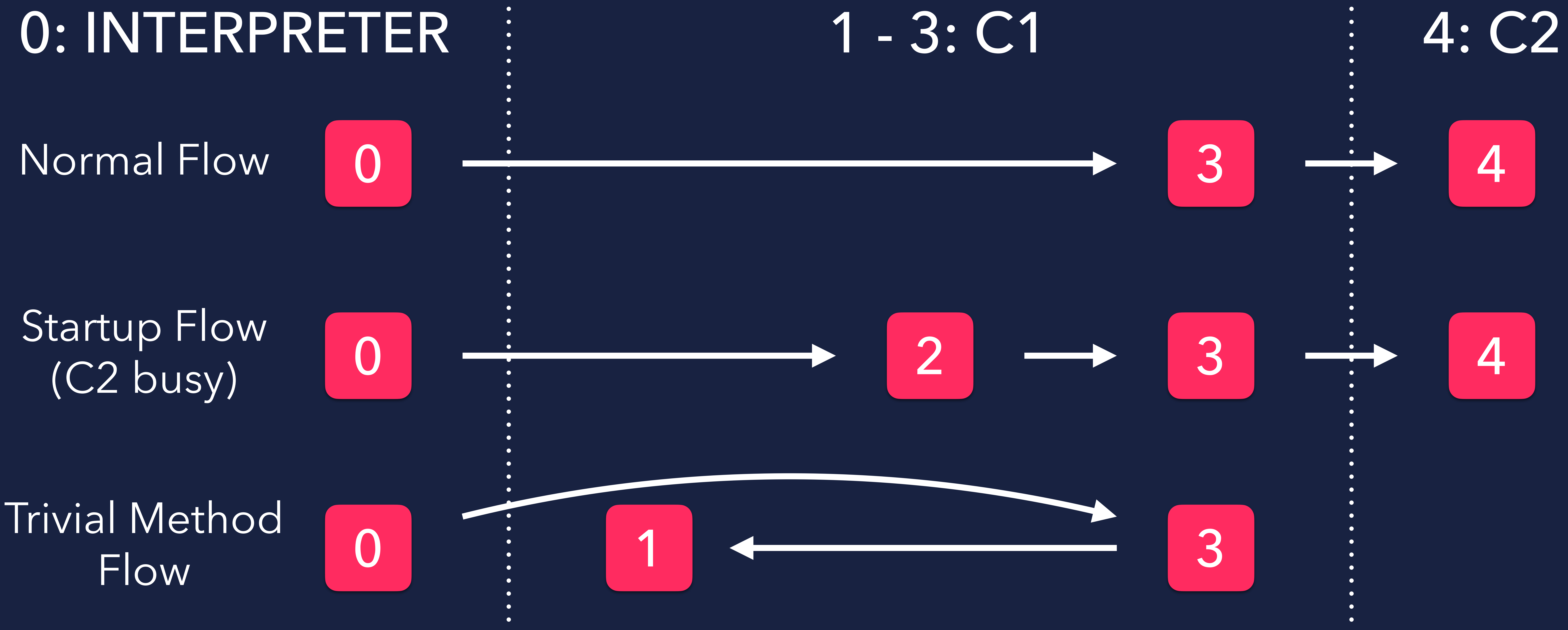
# LEVELS OF EXECUTION

- Level 0 - Interpreted code

- Level 1 - C1 compiled code (no profiling)
- Level 2 - C1 compiled code (basic profiling)
- Level 3 - C1 compiled code (full profiling)

- Level 4 - C2 compiled code (uses profile data from previous steps)

# LEVELS OF EXECUTION

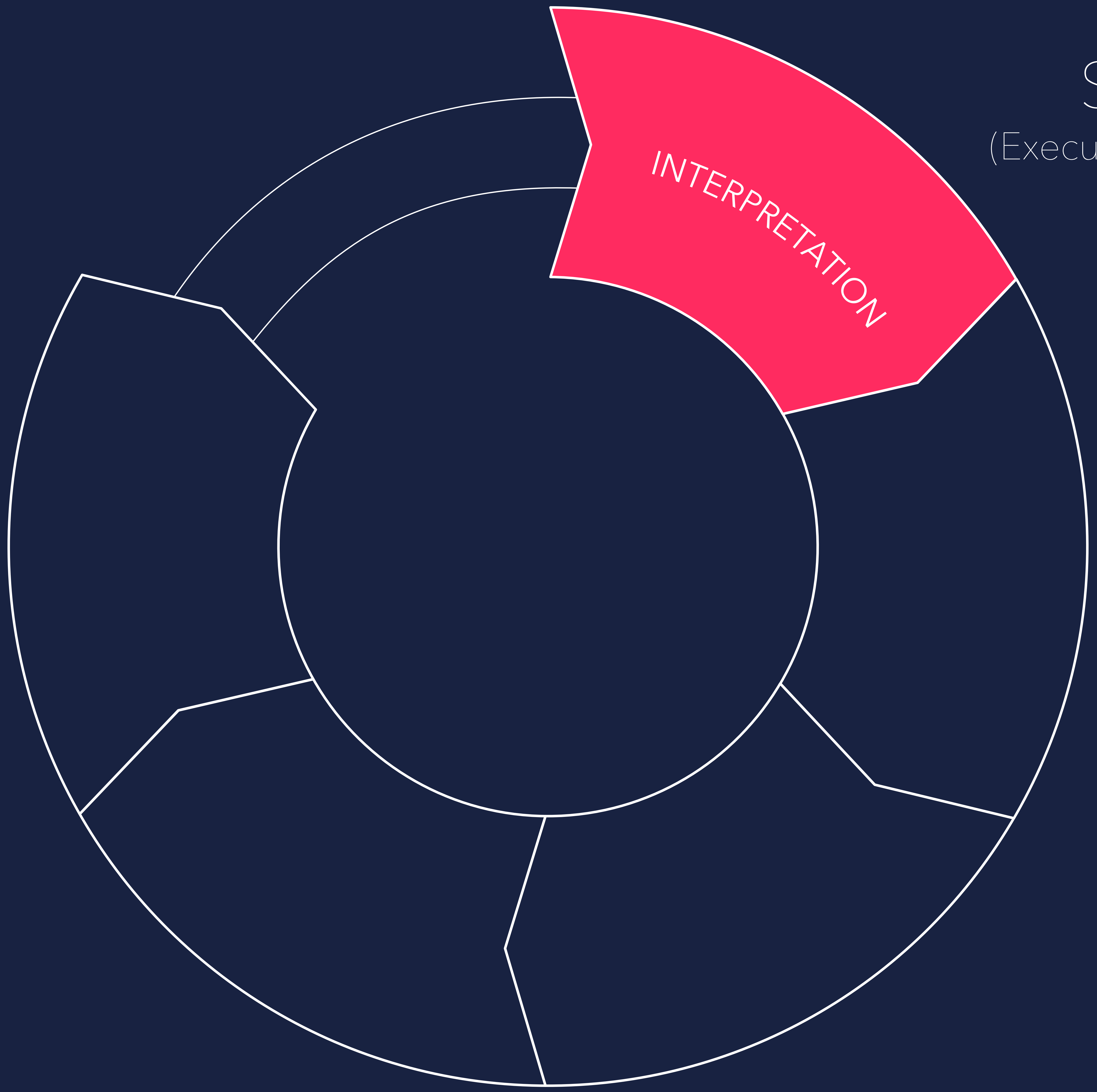


1: No profiling 2: Basic profiling 3: Full Profiling

azul

# EXECUTION CYCLE

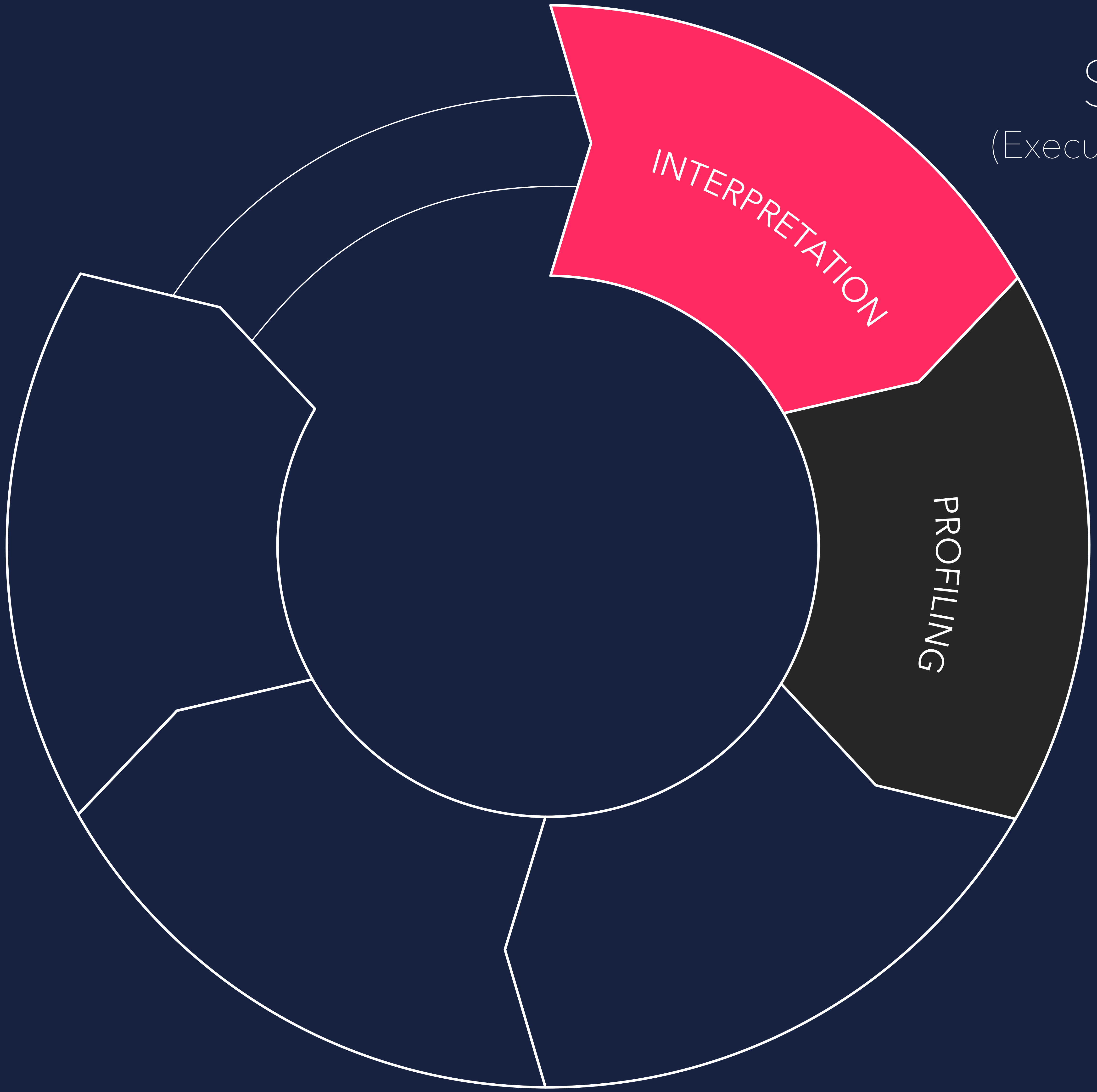
# EXECUTION CYCLE



Slow

(Execution Level 0)

# EXECUTION CYCLE



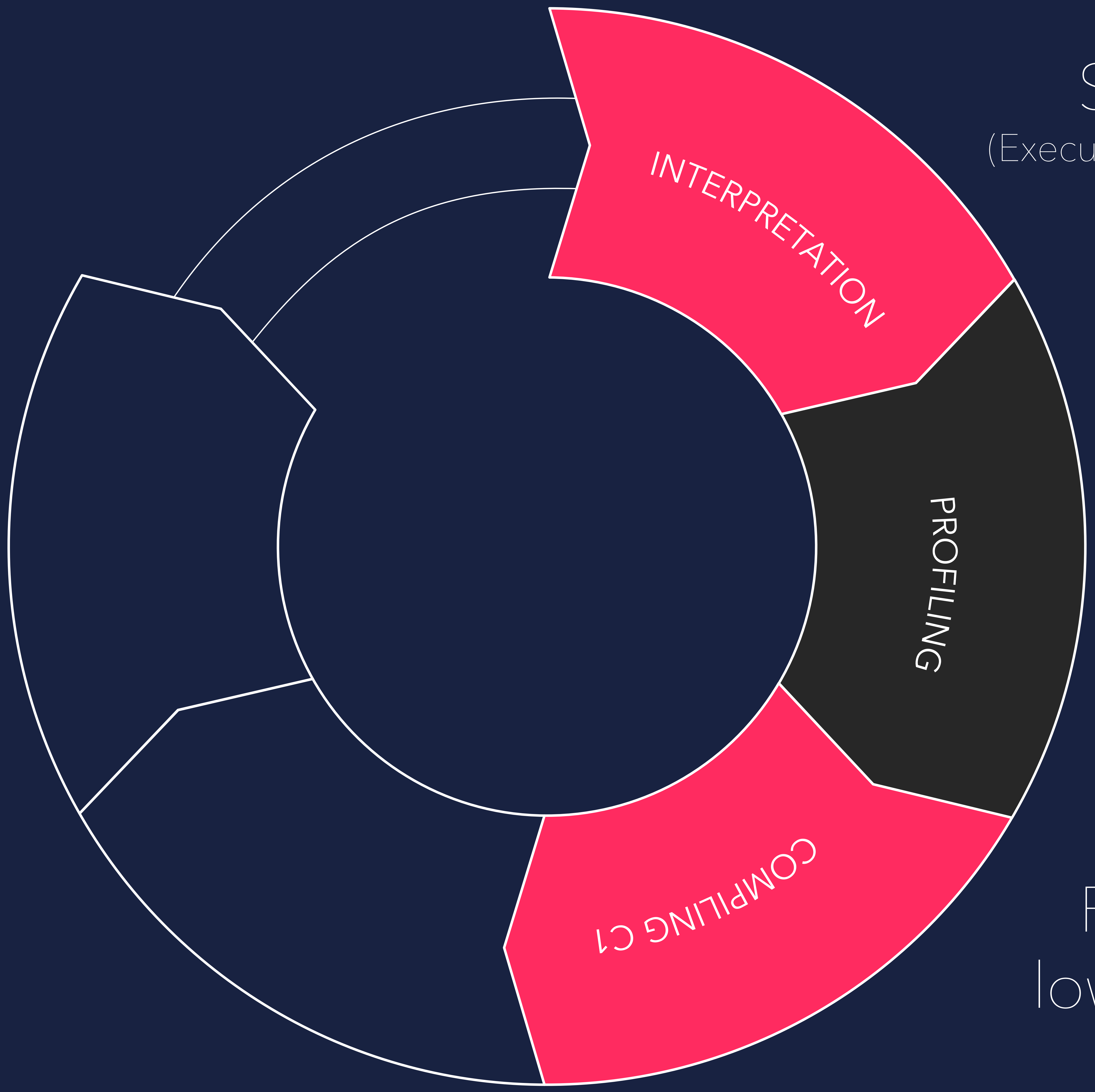
Slow

(Execution Level 0)

Finding "hotspots"

azul

# EXECUTION CYCLE



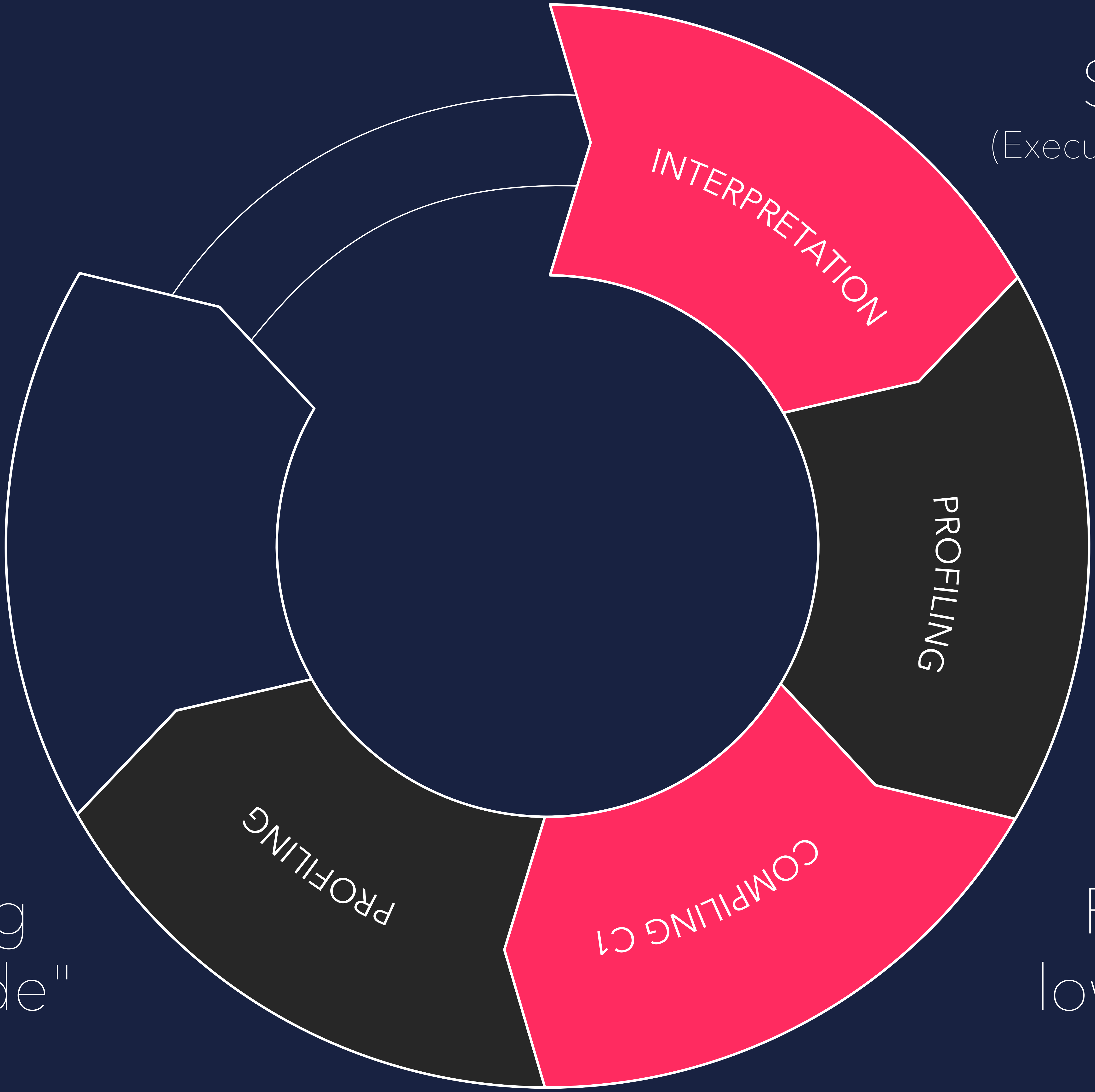
Slow
(Execution Level 0)

Finding "hotspots"

Fast compile, low optimisation
(Execution Level 3)

azul

# EXECUTION CYCLE



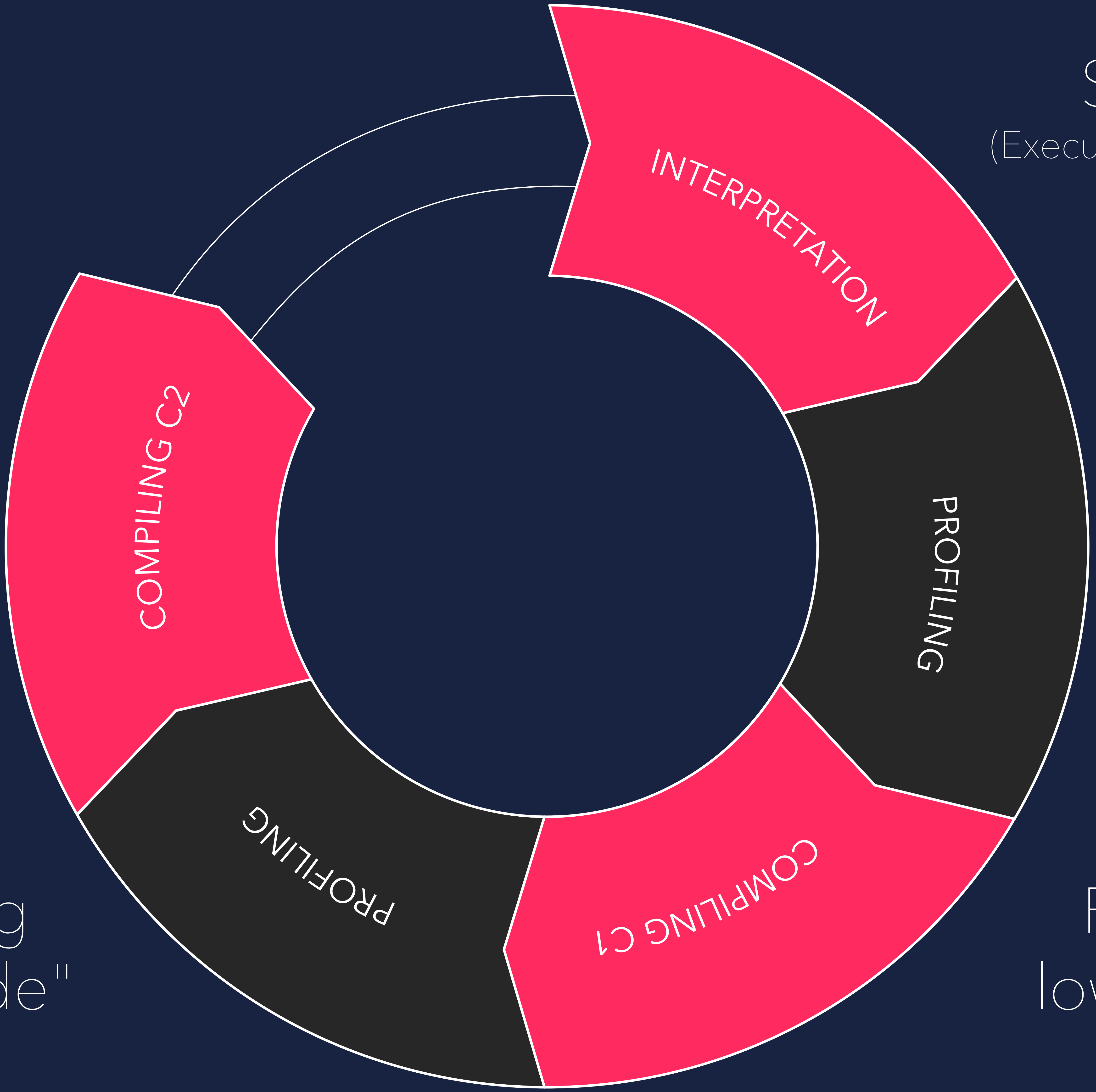
Slow
(Execution Level 0)

Finding
"hotspots"

Fast compile,
low optimisation
(Execution Level 3)

Finding
"hot code"

# EXECUTION CYCLE



Slow
(Execution Level 0)

Finding "hotspots"

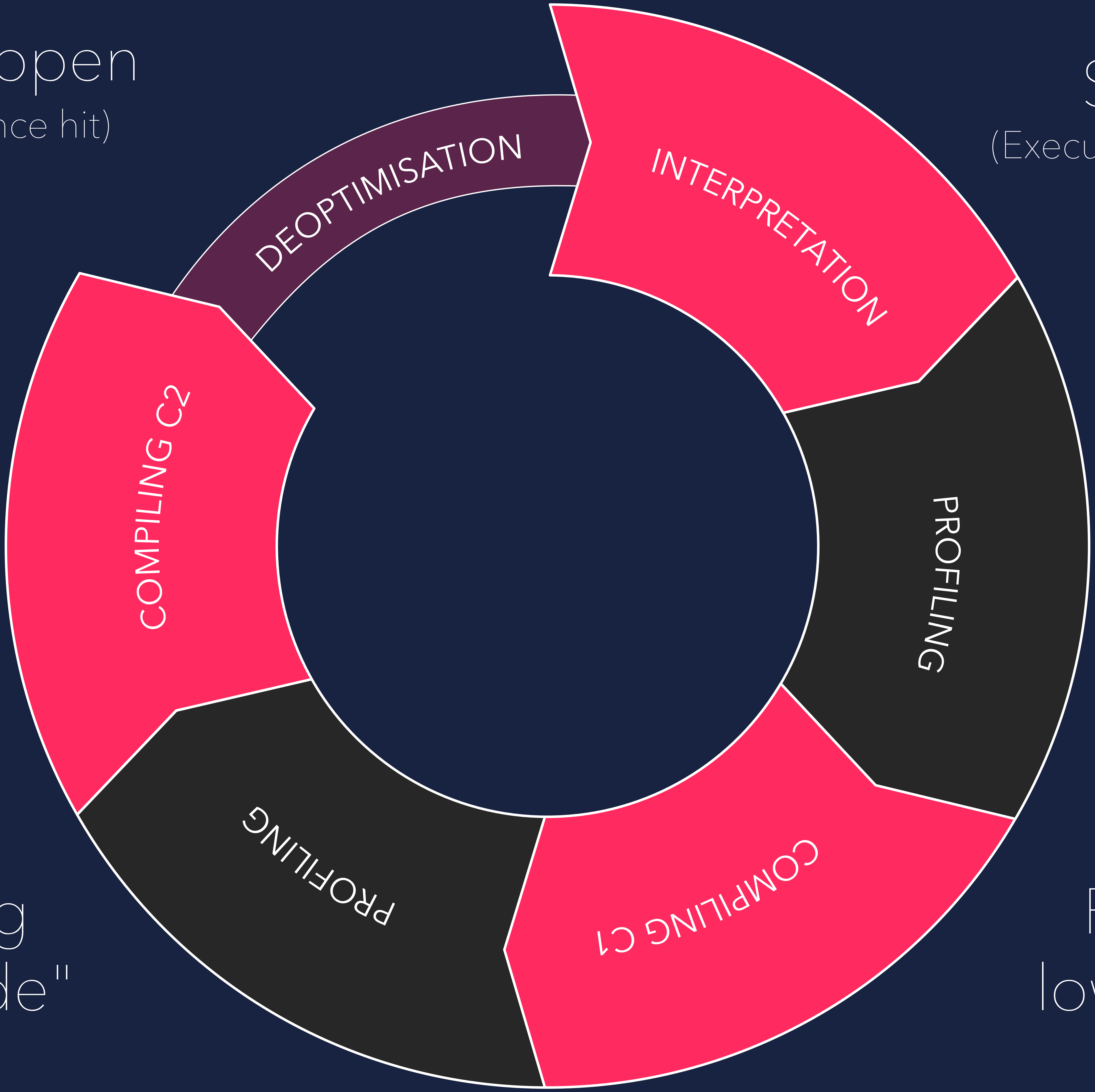
Fast compile, low optimisation
(Execution Level 3)

Finding "hot code"

Slower compile, high optimisation
(Execution Level 4)

azul

# EXECUTION CYCLE



Can happen
(performance hit)

Slow
(Execution Level 0)

Finding
"hotspots"

Fast compile,
low optimisation
(Execution Level 3)

Finding
"hot code"

Slower compile,
high optimisation
(Execution Level 4)

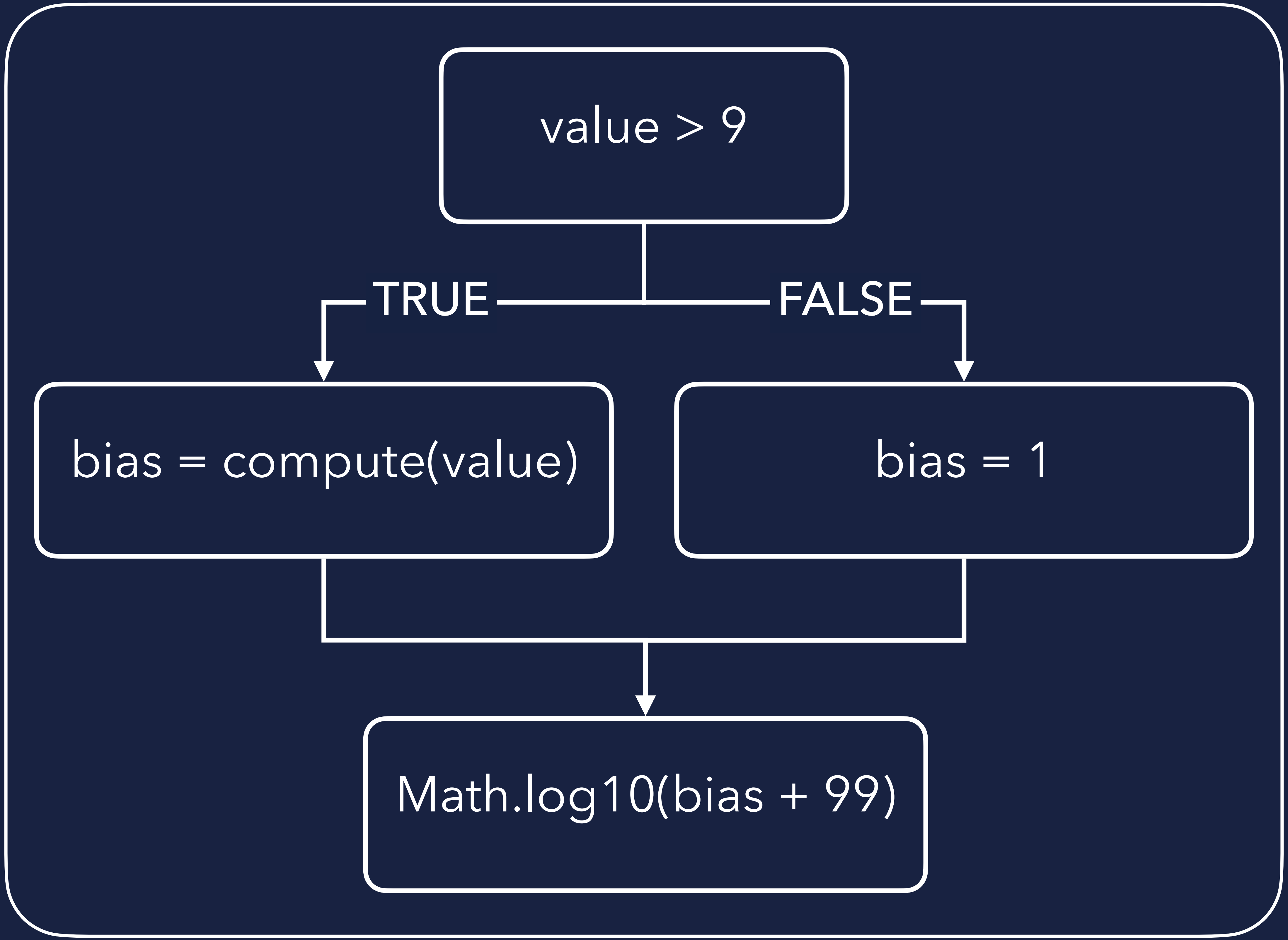
azul

# DEOPTIMISATION

# DEOPTIMISATION

## BRANCH ANALYSIS

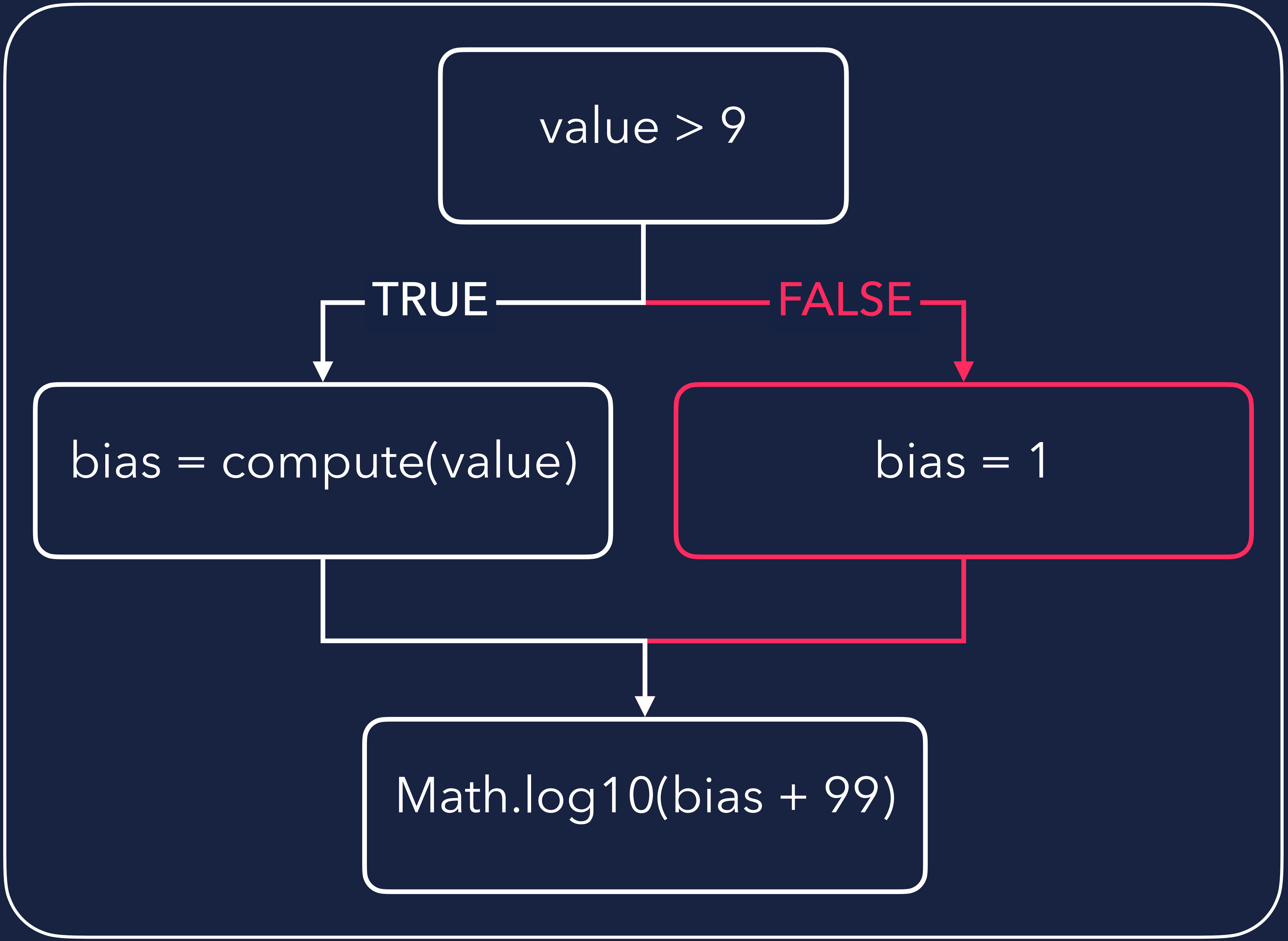
```
int computeMagnitude (int value) {
    var bias;
    if (value > 9) {
        bias = compute(value);
    } else {
        bias = 1:
    }
    return Math.log10(bias + 99);
}
```



# DEOPTIMISATION

## BRANCH ANALYSIS

```
int computeMagnitude (int value) {
    var bias;
    if (value > 9) {
        bias = compute(value);
    } else {
        bias = 1:
    }
    return Math.log10(bias + 99);
}
```

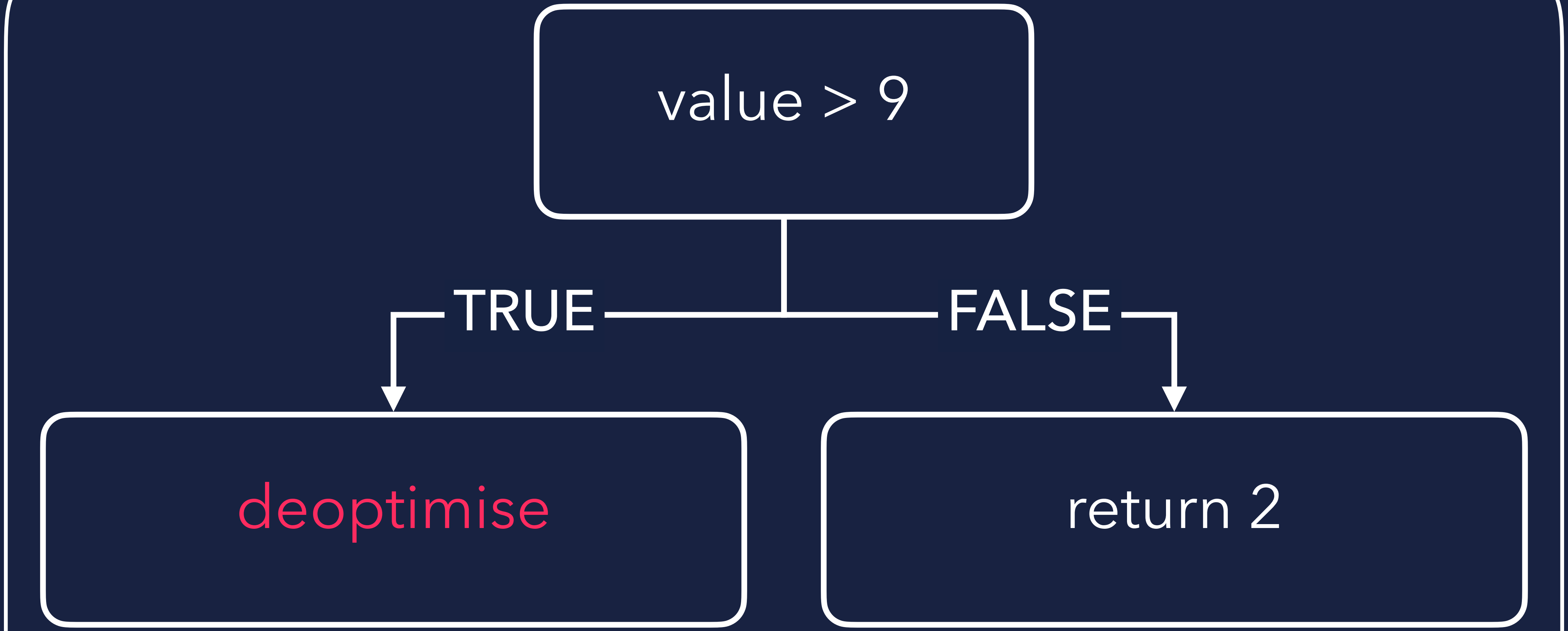


value was never greater than 9

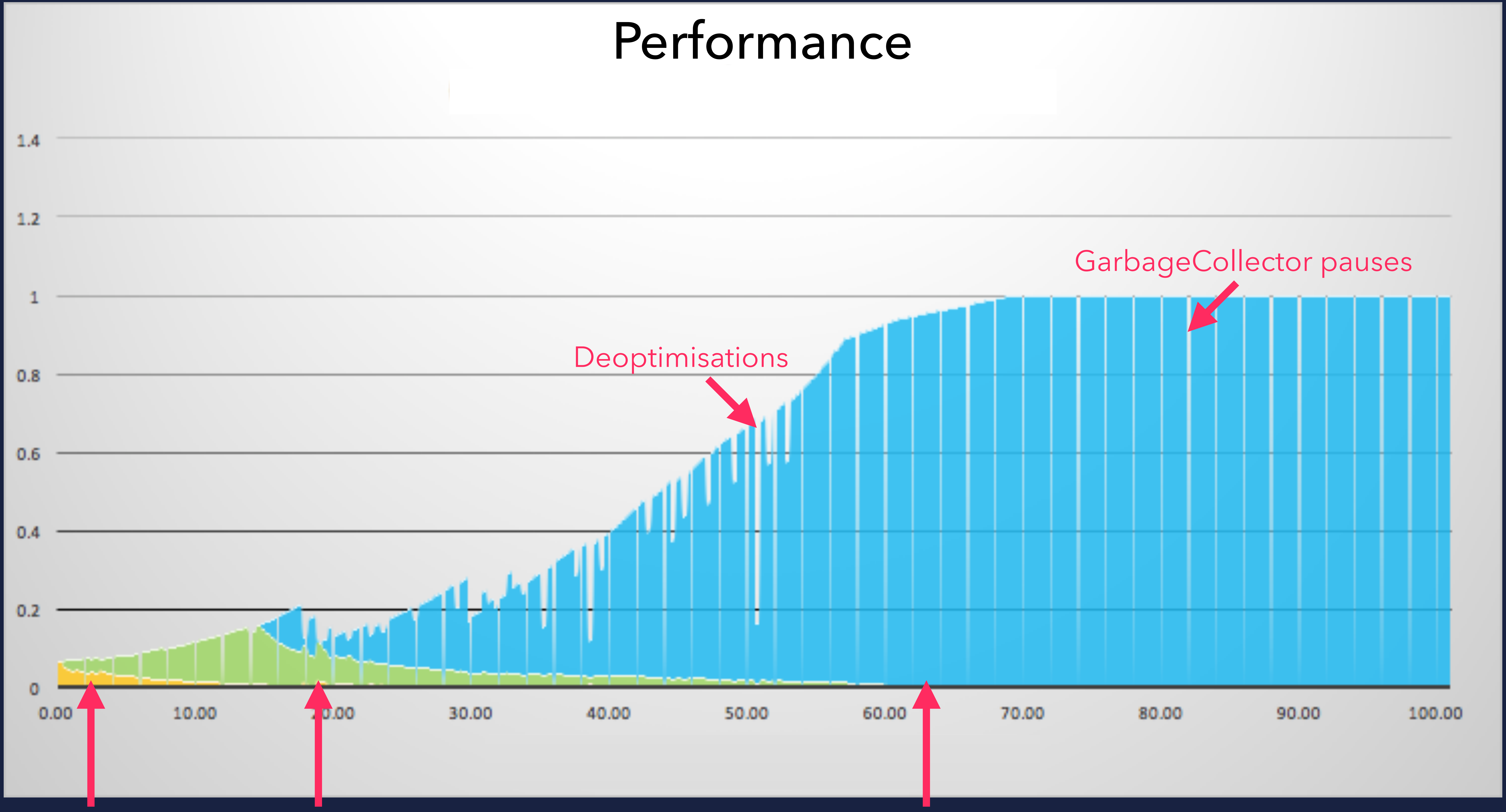
# DEOPTIMISATION

## BRANCH ANALYSIS

```
int computeMagnitude (int value) {
    if (value > 9) {
        uncommonTrap();
    }
    return 2; //Math.log10(100)
}
```



# JVM PERFORMANCE GRAPH



Interpreter C1 Compiler C2 Compiler

azul

# THAT'S GREAT...

azul

# ...BUT...

# ...IT TAKES TIME !

# JVM STARTUP

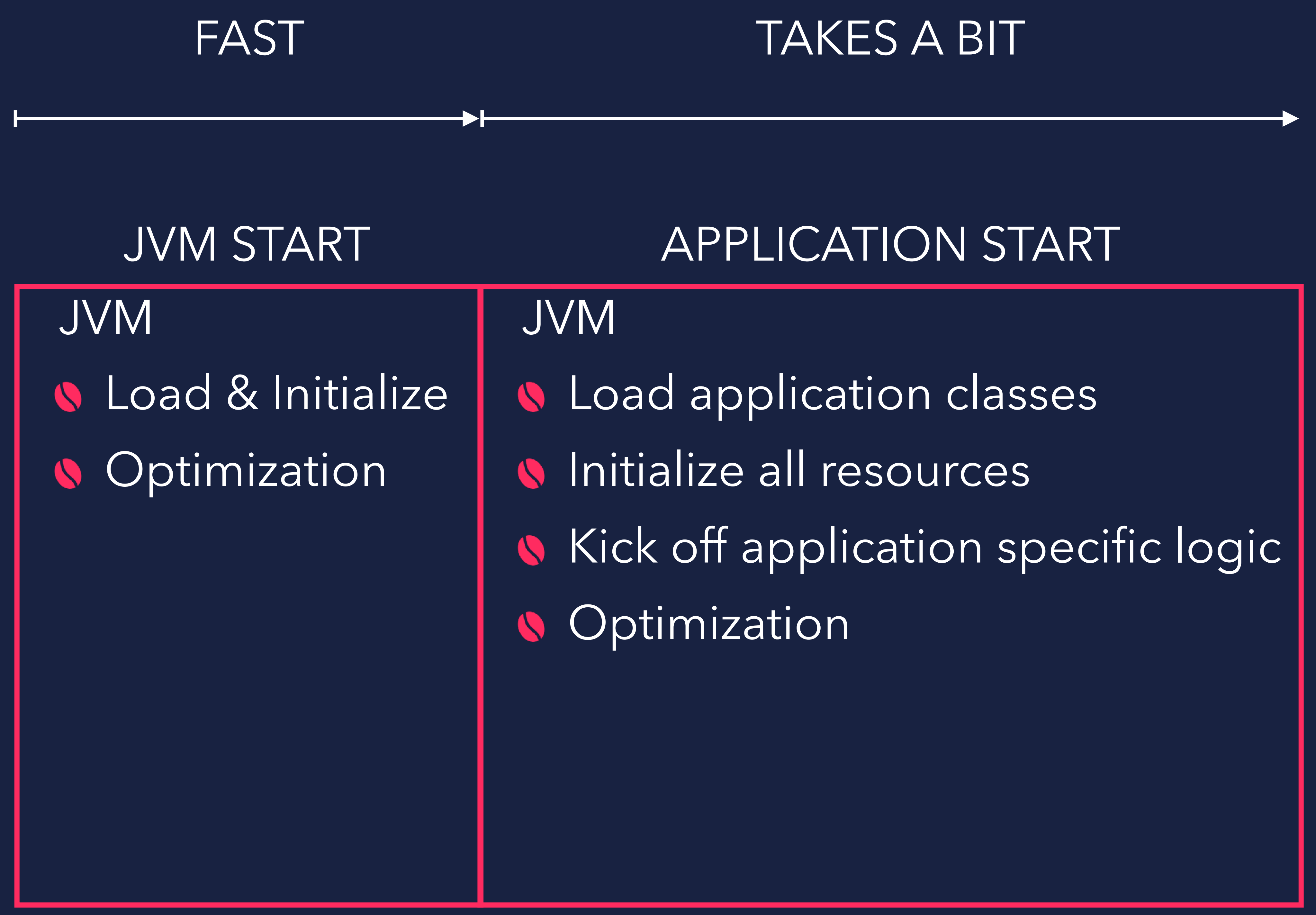
FAST

JVM START

JVM

- Load & Initialize
- Optimization

# JVM STARTUP



FAST

TAKES A BIT

JVM START

JVM

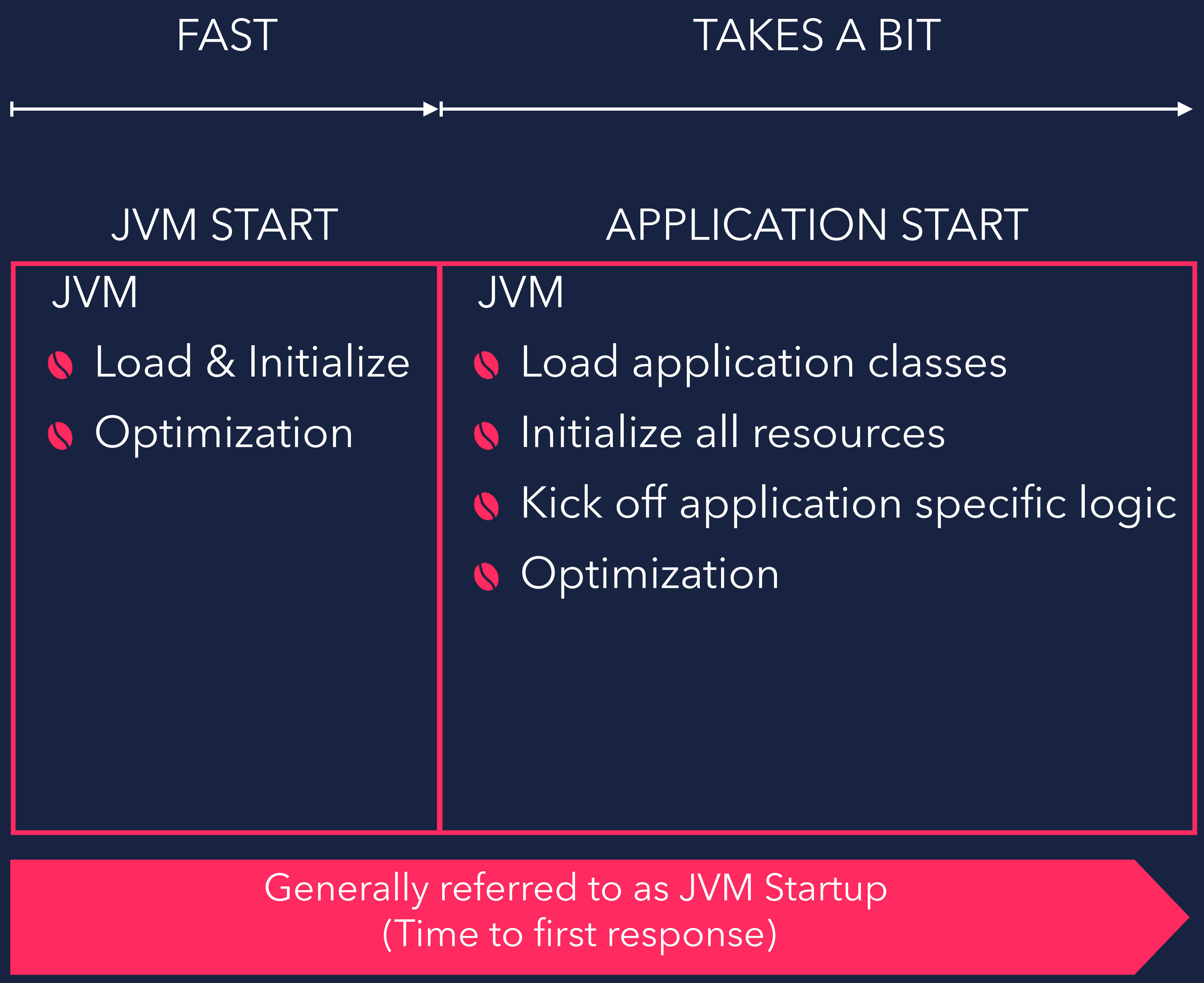
- Load & Initialize
- Optimization

APPLICATION START

JVM

- Load application classes
- Initialize all resources
- Kick off application specific logic
- Optimization

# JVM STARTUP



FAST

TAKES A BIT

JVM START

JVM

- Load & Initialize
- Optimization

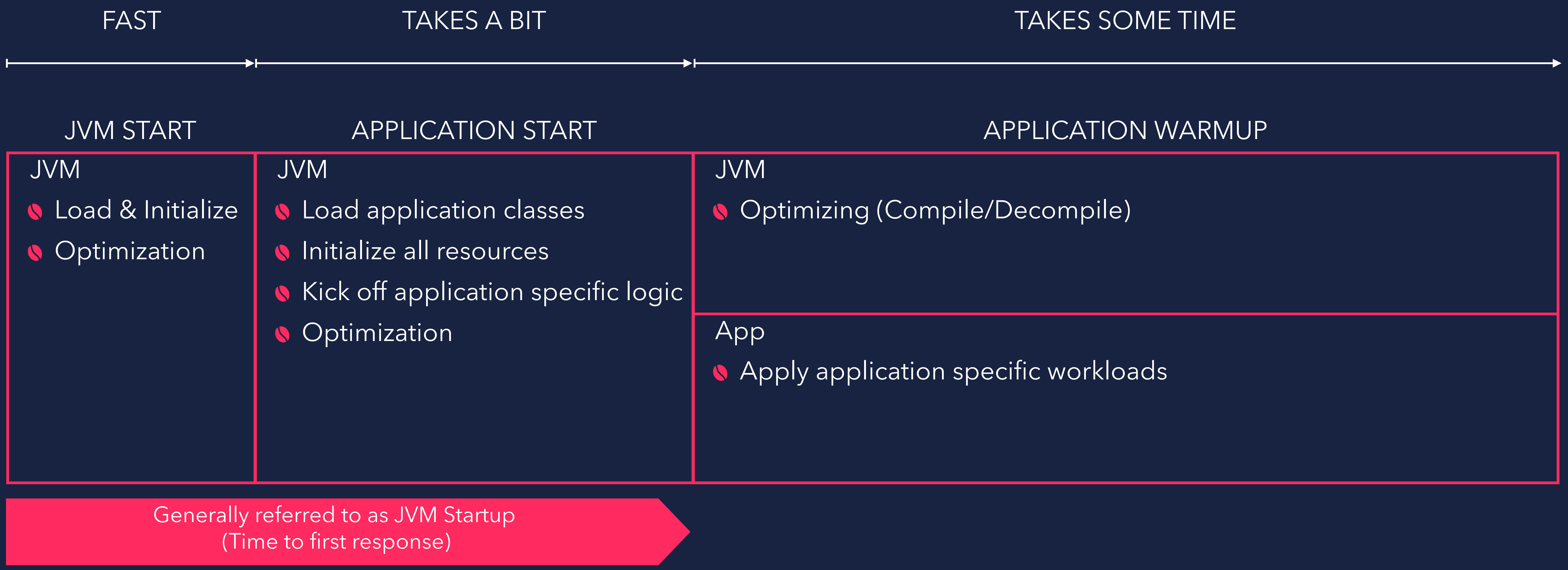
APPLICATION START

JVM

- Load application classes
- Initialize all resources
- Kick off application specific logic
- Optimization

Generally referred to as JVM Startup
(Time to first response)

# JVM STARTUP



FAST

TAKES A BIT

TAKES SOME TIME

JVM START

JVM

- Load & Initialize
- Optimization

APPLICATION START

JVM

- Load application classes
- Initialize all resources
- Kick off application specific logic
- Optimization

APPLICATION WARMUP

JVM

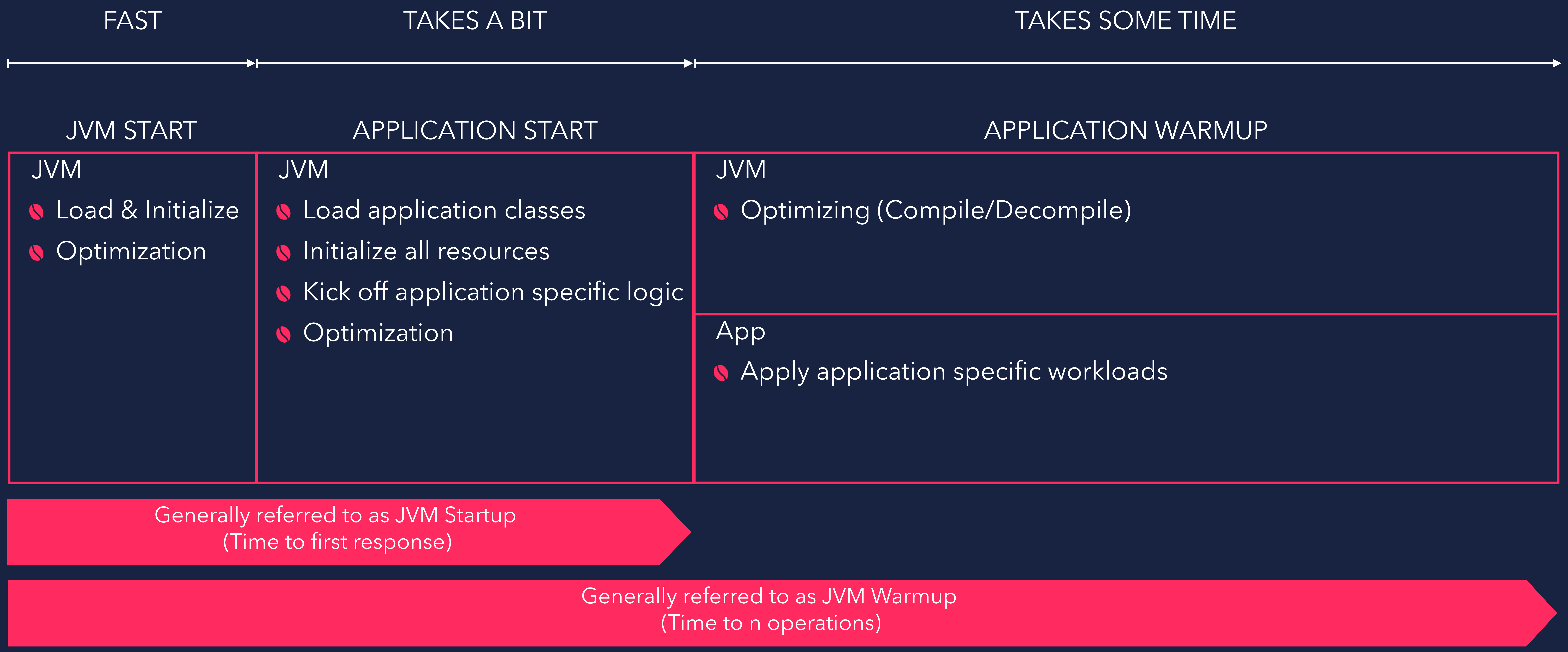
- Optimizing (Compile/Decompile)

App

- Apply application specific workloads

Generally referred to as JVM Startup
(Time to first response)

# JVM STARTUP



FAST

TAKES A BIT

TAKES SOME TIME

## JVM START

JVM

- Load & Initialize
- Optimization

## APPLICATION START

JVM

- Load application classes
- Initialize all resources
- Kick off application specific logic
- Optimization

## APPLICATION WARMUP

JVM

- Optimizing (Compile/Decompile)

App

- Apply application specific workloads

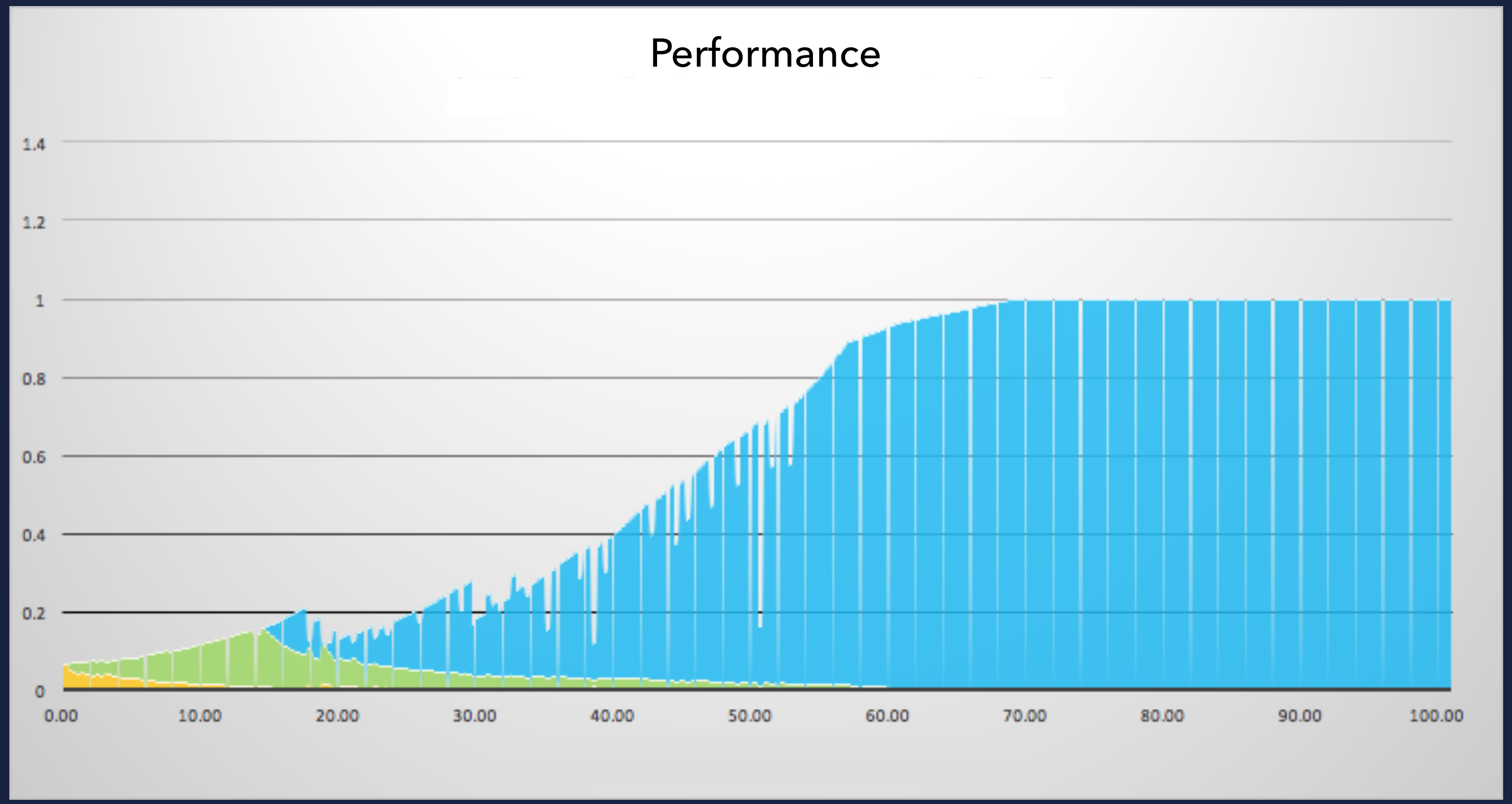
Generally referred to as JVM Startup
(Time to first response)

Generally referred to as JVM Warmup
(Time to n operations)

# MICROSERVICE ENVIRONMENT

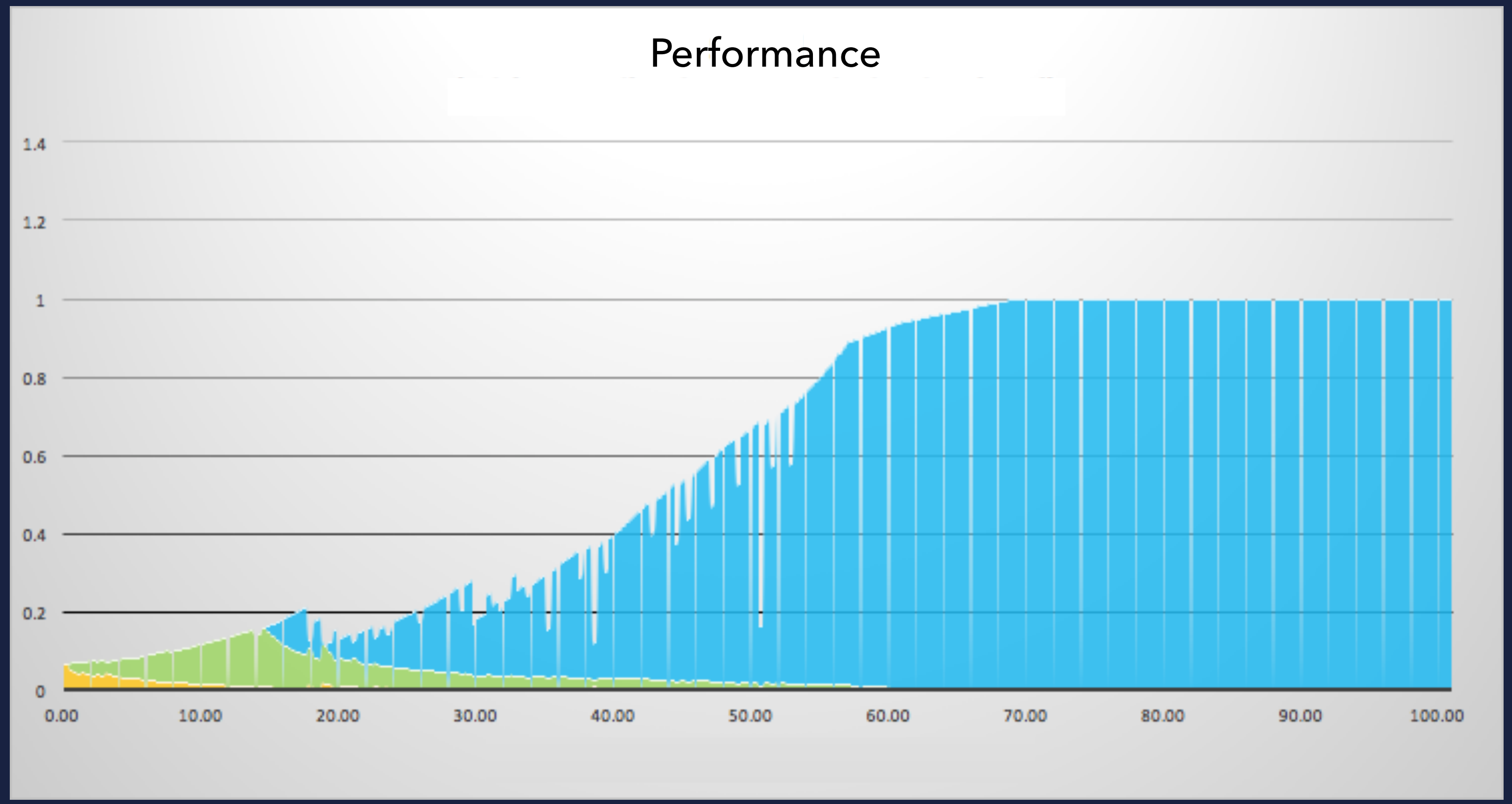
# MICROSERVICE ENVIRONMENT

FIRST RUN



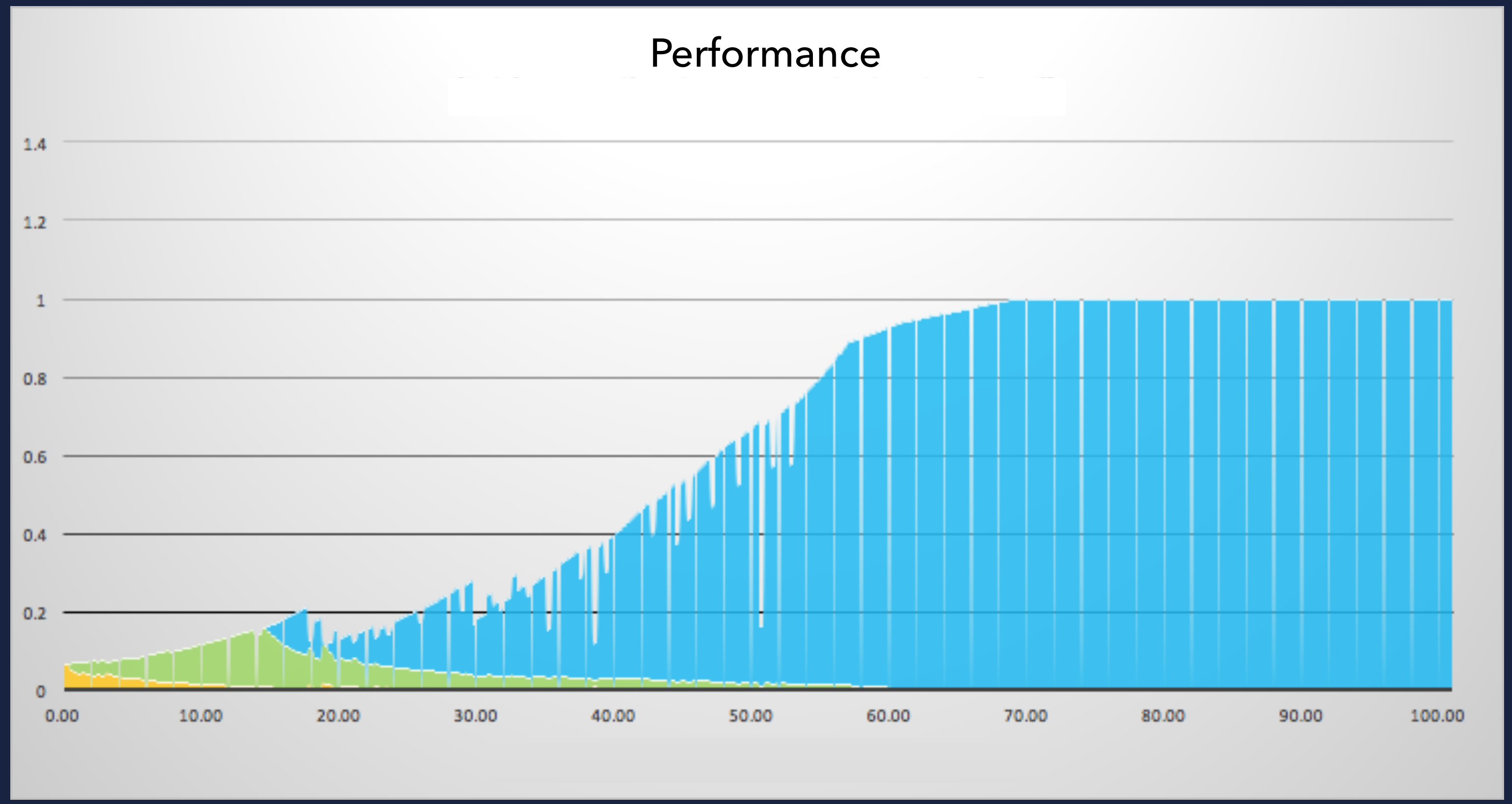
JVM STARTUP

SECOND RUN



JVM STARTUP

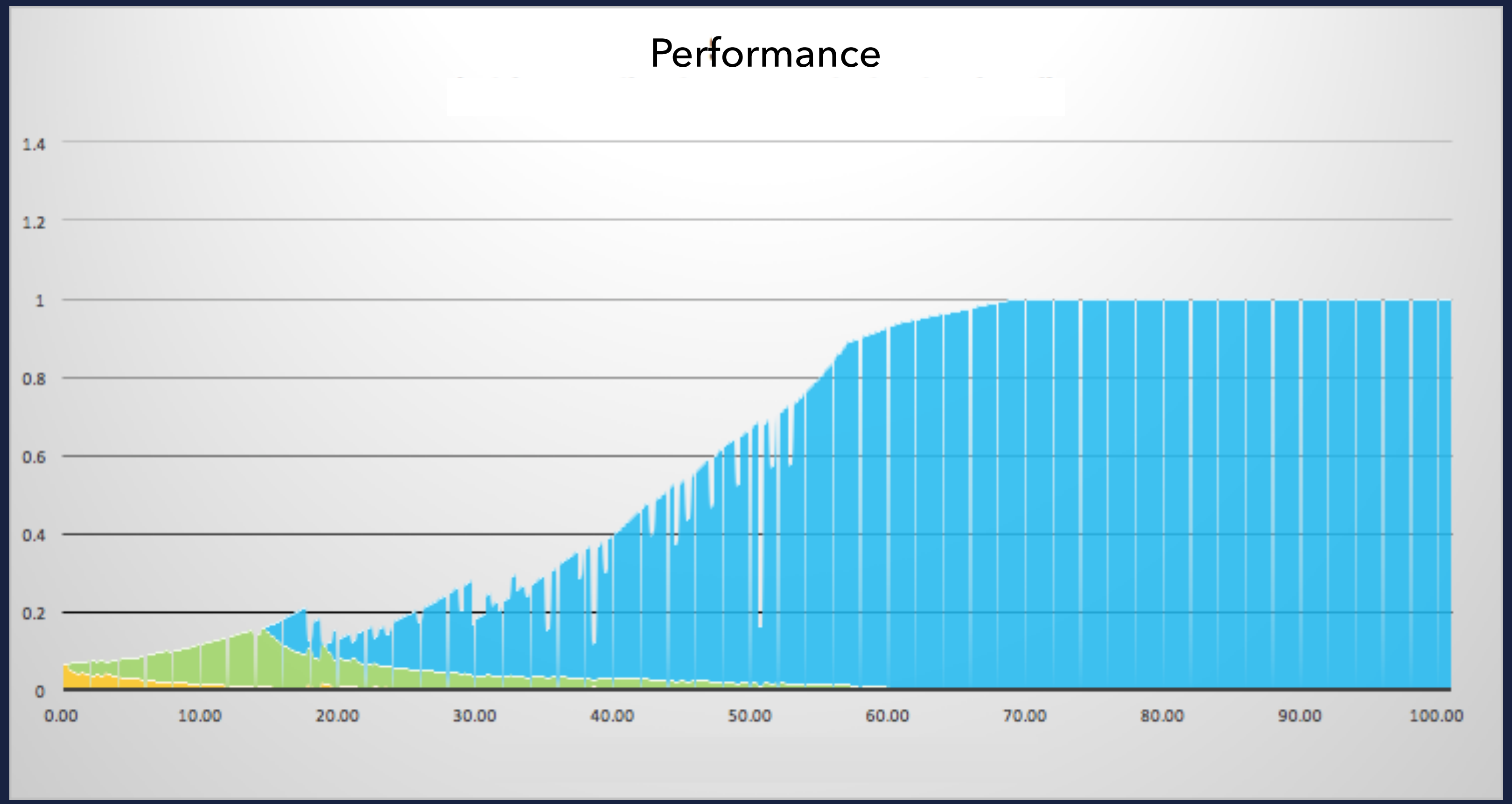
THIRD RUN



JVM STARTUP

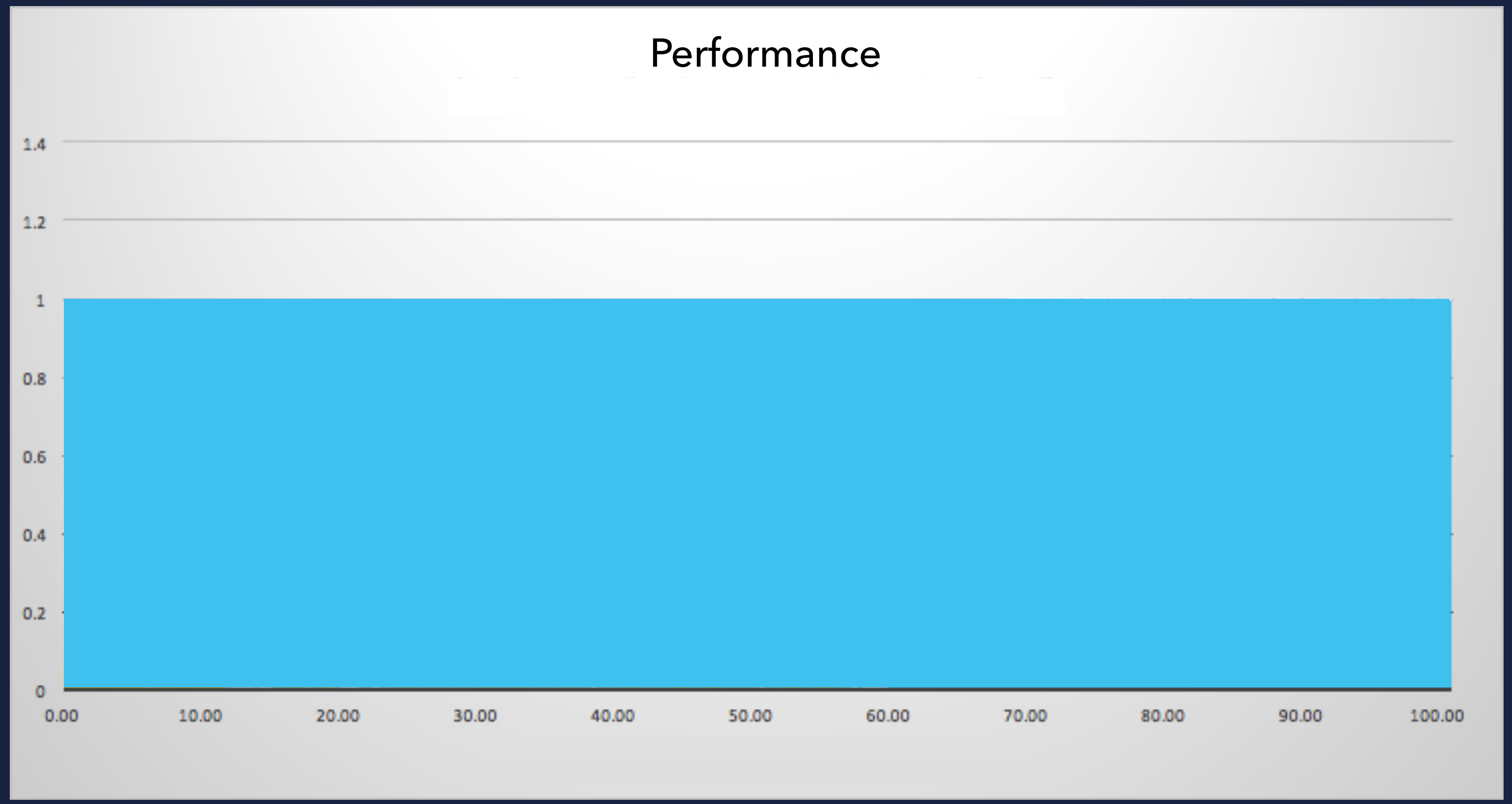
# WOULDN'T IT BE GREAT...?

## FIRST RUN



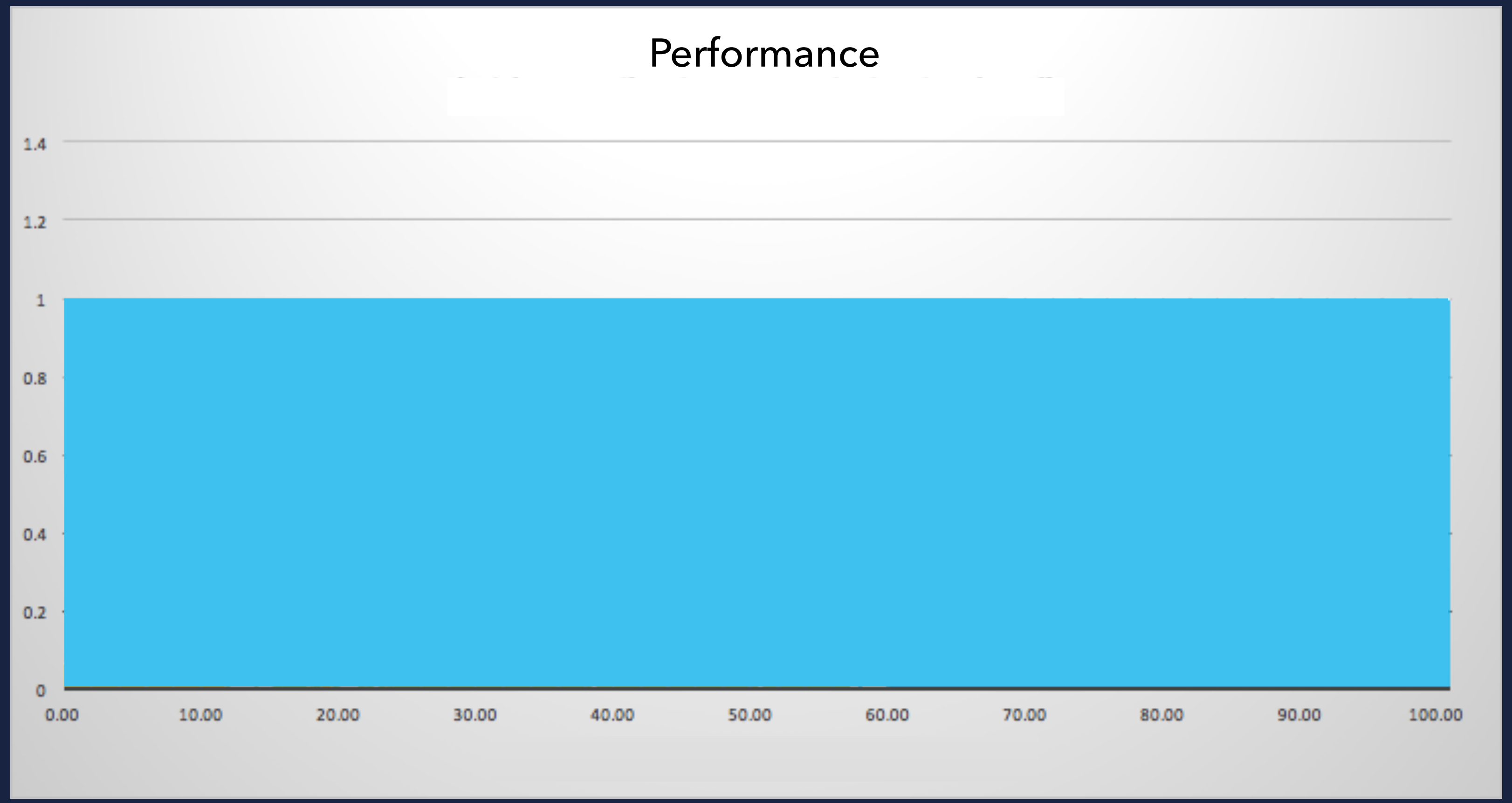
JVM STARTUP

## SECOND RUN



NO STARTUP OVERHEAD

## THIRD RUN



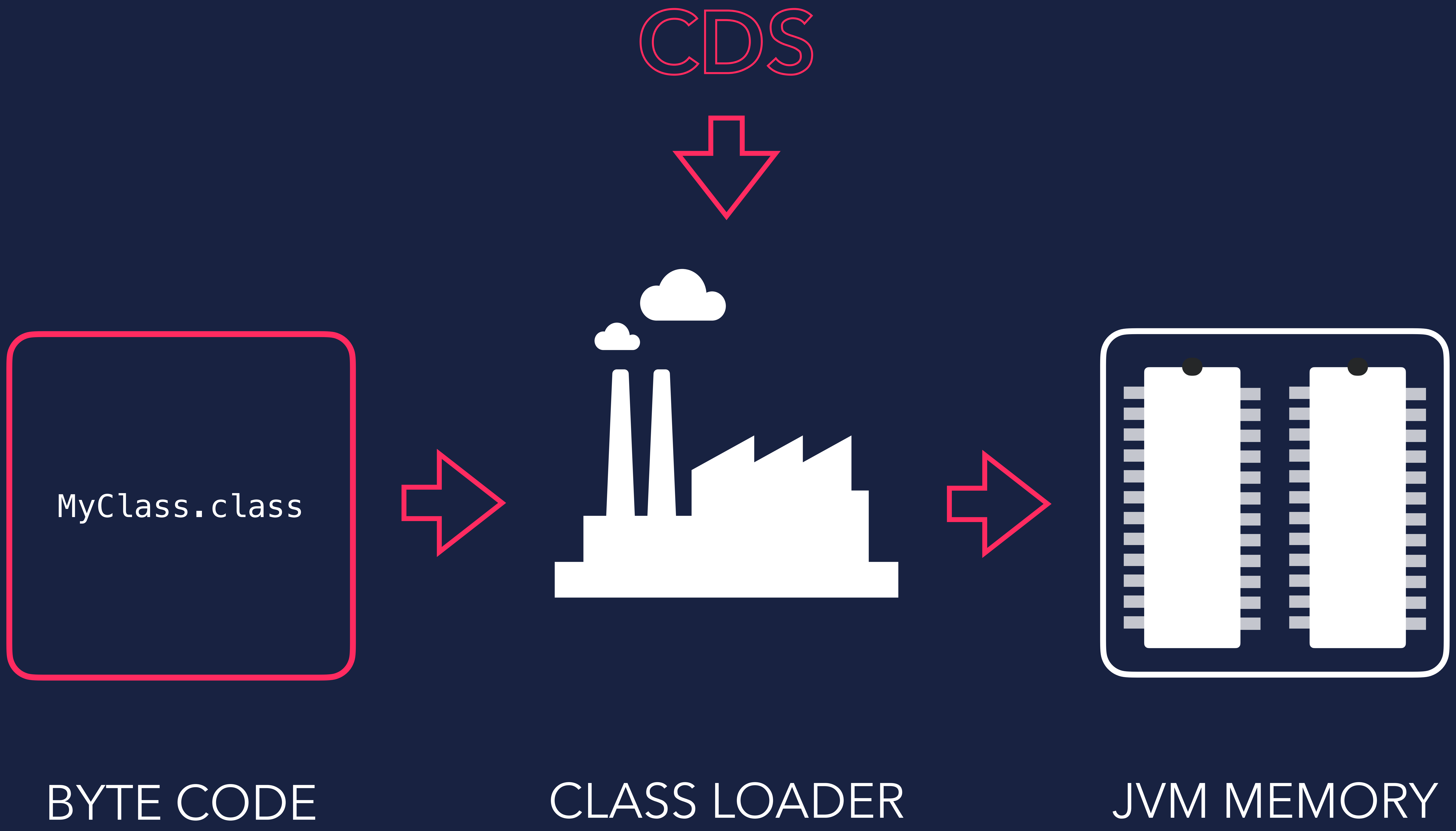
NO STARTUP OVERHEAD

# SOLUTIONS…?

# CLASS DATA SHARING

# WHAT ABOUT CDS?

- Dump internal class representations into file
- Shared on each JVM start (CDS)
- No optimization or hotspot detection
- Only reduces class loading time
- Startup up to 2 seconds faster
- Good info from Ionut Balosin


CDS
MyClass.class
BYTE CODE
CLASS LOADER
JVM MEMORY

# AHEAD OF TIME COMPILATION

# WHY NOT USE AOT?

- No interpreting bytecodes
- No analysis of hotspots
- No runtime compilation of code
- Start at full speed, straight away
- GraalVM native image does that

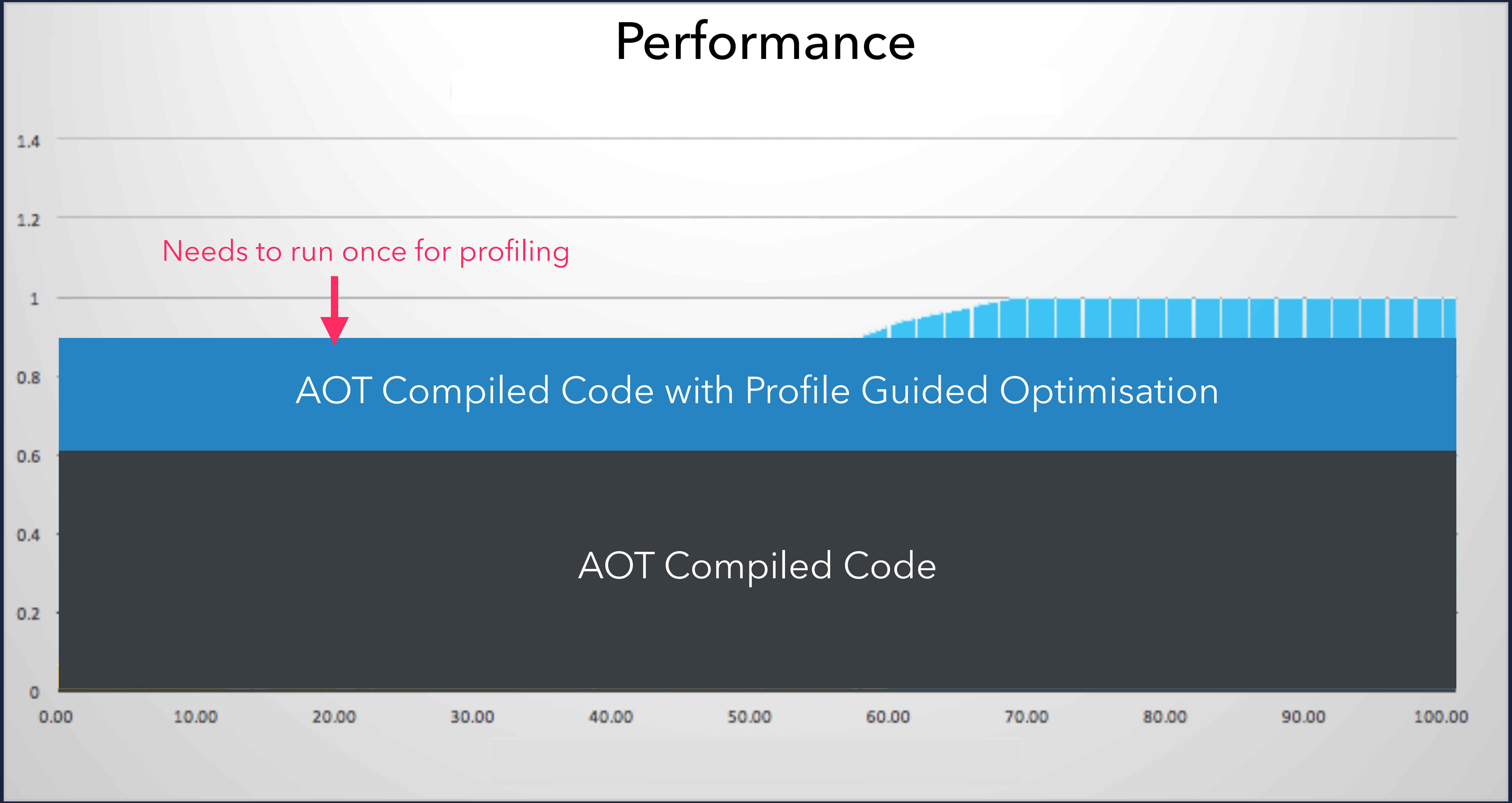
PROBLEM SOLVED...?

azul

# NOT SO FAST…

- AOT is, by definition, static
- Code is compiled before it is run
- Compiler has no knowledge of how the code will actually run
- Profile Guided Optimisation (PGO) can partially help

# JVM PERFORMANCE GRAPH



azul

# AOT VS JIT

## AOT

- Limited use of method inlining
- No runtime bytecode generation
- Reflection is possible but complicated
- Unable to use speculative optimisations
- Overall performance will typically be lower
- Deployed env != Development env.

- Full speed from the start
- No overhead to compile code at runtime
- Small memory footprint

## JIT

- Can use aggressive method inlining at runtime
- Can use runtime bytecode generation
- Reflection is simple
- Can use speculative optimisations
- Overall performance will typically be higher
- Deployed env. == Development env.

- Requires more time to start up
- Overhead to compile code at runtime
- Larger memory footprint

# A DIFFERENT APPROACH

azul

# CRIU

CHECKPOINT RESTORE IN USERSPACE

# CHECKPOINT RESTORE IN USERSPACE

- Linux project
- Part of kernel >= 3.11 (2013)
- Freeze a running container/application
- Checkpoint its state to disk
- Restore the container/application from the saved data.
- Used by/integrated in OpenVZ, LXC/LXD, Docker, Podman and others

# Checkpoint Restore In Userspace

- Heavily relies on /proc file system
- It can checkpoint:
  - Processes and threads
  - Application memory, memory mapped files and shared memory
  - Open files, pipes and FIFOs
  - Sockets
  - Interprocess communication channels
  - Timers and signals
- Can rebuild TCP connection from one side only

# CRIU CHALLENGES

- Restart from saved state on another machine
  (open files, shared memory etc.)
- Start multiple instances of same state on same machine
  (PID will be restored which will lead to problems)
- A Java Virtual Machine would assume it was continuing its tasks
  (very difficult to use effectively, e.g. running applications might have open files etc.)

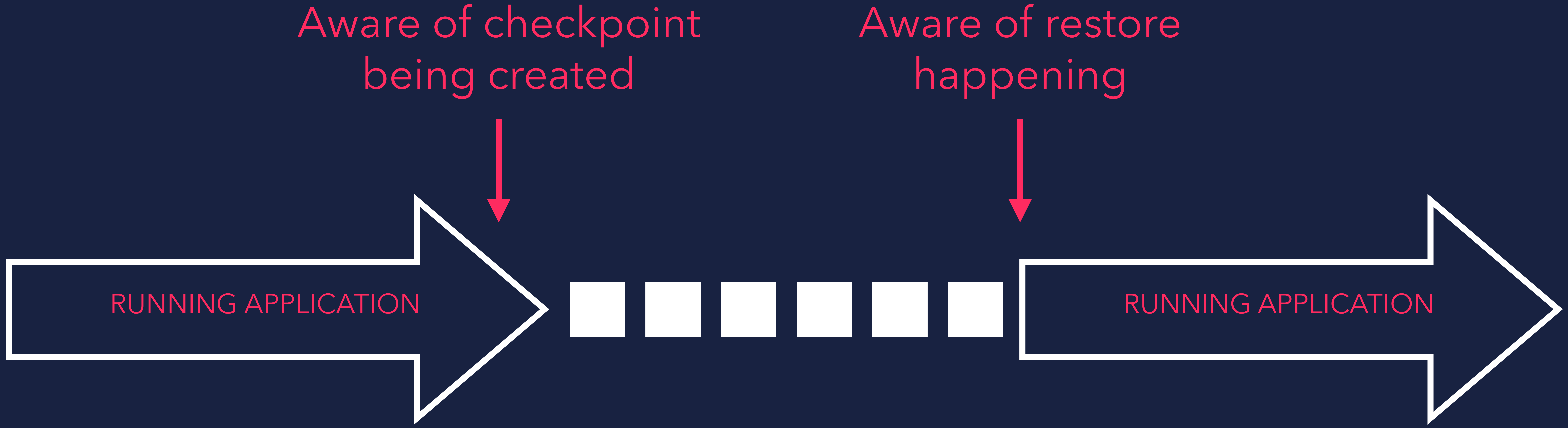
# CRaC

Coordinated Restore at Checkpoint

# CRaC

A way to solve the problems when checkpointing a JVM

(e.g. no open files, sockets etc.)



# CRaC

- CRIU comes bundled with the JDK
- Heap is cleaned, compacted
  (using JVM safepoint mechanism -> JVM is in a safe state)
- Comes with a simple API
- Creates checkpoints using code or jcmd
- Throws CheckpointException
  (in case of open files/sockets)

# CRaC

## Additional command line parameters

START

```
>java -XX:CRaCCheckpointTo=PATH -jar app.jar
```

RESTORE

```
>java -XX:CRaCRestoreFrom=PATH
```

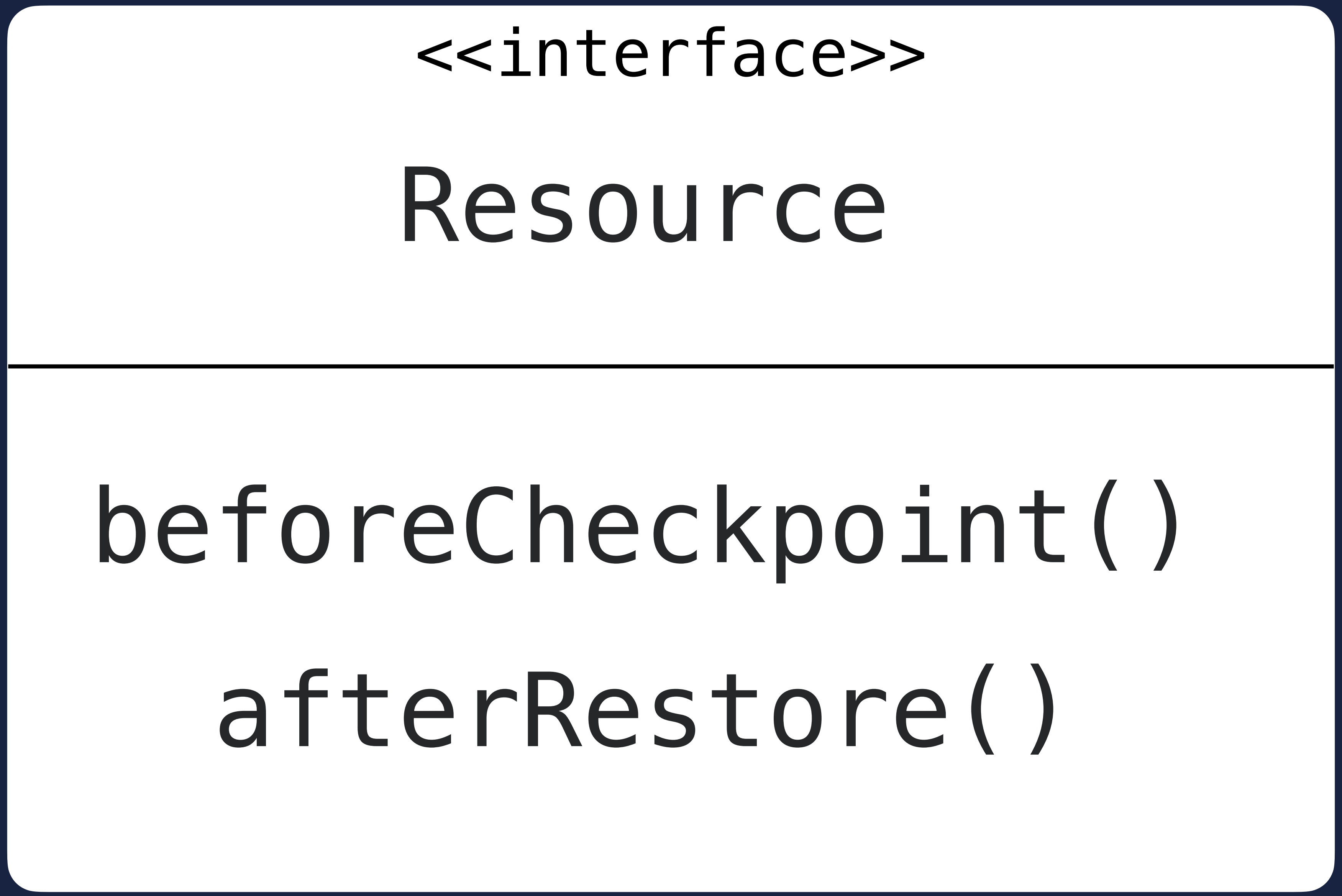
# openjdk.org/projects/crac

Lead by Anton Kozlov (Azul)

# CRaC API

# CRaC API

- CRaC uses Resources that can be notified about a Checkpoint and Restore
- Classes in application code implement the Resource interface
- The application receives callbacks during checkpointing and restoring
- Makes it possible to close/restore resources (e.g. open files, sockets)



<<interface>>

Resource

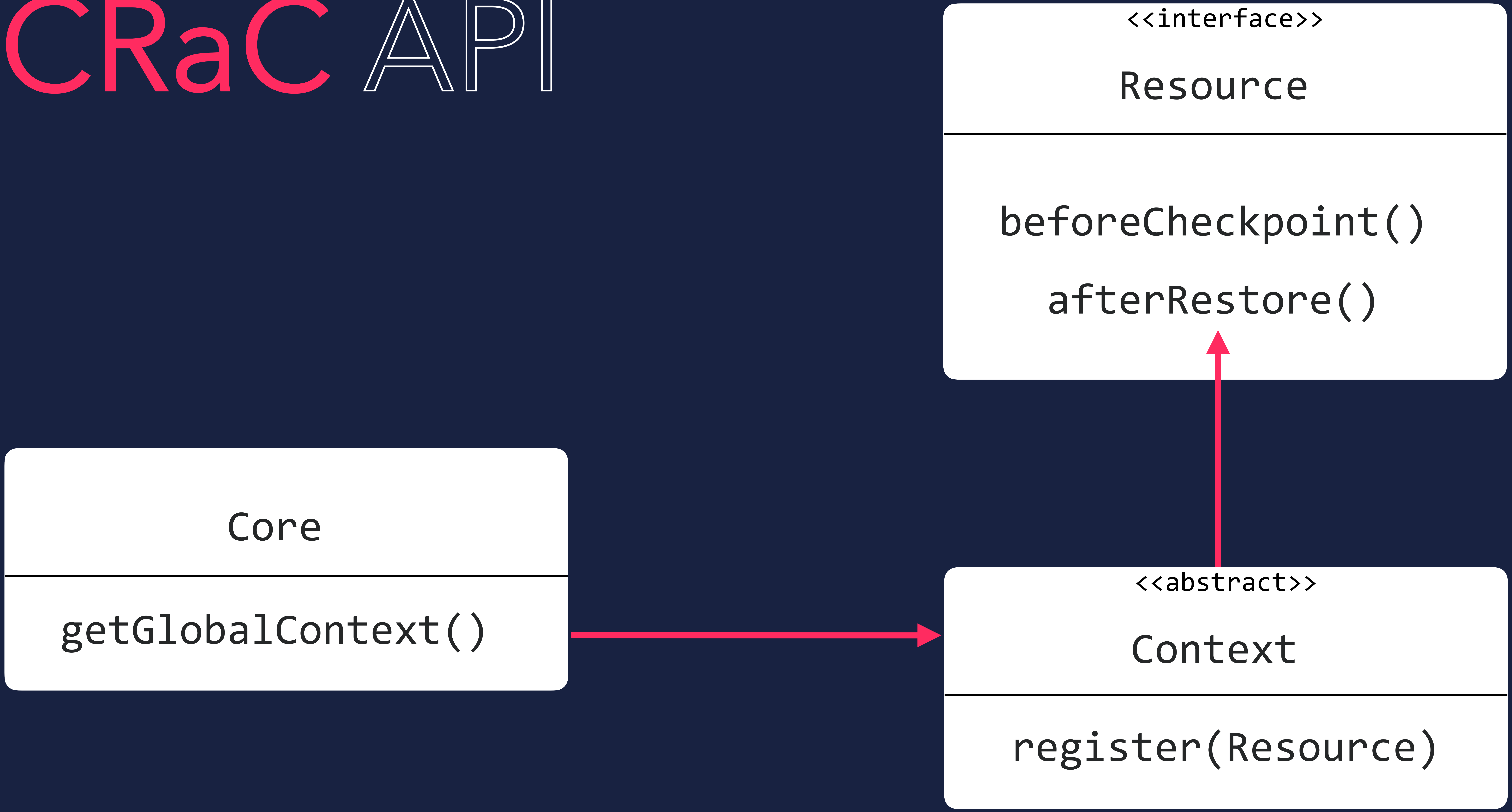
beforeCheckpoint()

afterRestore()

# CRaC API

- Resource objects need to be registered with a Context so that they can receive notifications
- There is a global Context accessible by via the static method Core.getGlobalContext()

# CRaC API



<<interface>>

Resource

beforeCheckpoint()

afterRestore()

Core

getGlobalContext()

<<abstract>>

Context

register(Resource)

azul

# CRaC API

- The global Context maintains a list of Resource objects
- The beforeCheckpoint() methods are called in the reverse order the Resource objects have been registered
- The afterRestore() methods are called in the order the Resource objects have been registered

# CREATING A CHECKPOINT

## FROM THE COMMAND LINE:

```
>jcmd YOUR_AWESOME.jar JDK.checkpoint
```

```
>jcmd PID JDK.checkpoint
```

# CREATING A CHECKPOINT

## FROM THE CODE:

```
Core.checkpointRestore();
```

# WHEN TO CHECKPOINT ?

- Run your app with your typical workload
- Use the parameter -XX:+PrintCompilation
- Observe the moment the compilations are ramped down
- Create the checkpoint

# TYPICAL USAGE…

- Run app in a docker container
- Create checkpoint in the docker container
- Commit the state of checkpointed container
- Start the container from checkpointed state

# COMPATIBILITY…

- Only on Linux x64 (at the moment, aarch64 would be possible)
- Upgrade (cp: Core i7 -> restore: Core i9)
- No downgrade (cp: Core i9 -> restore: Core i7)
- Usually node groups in cloud env. stick to same cpu architecture
- Using docker it works on linux, macos & windows

# DEMO...

## JVM STARTUP

# JVM STARTUP DEMO

```java
public Main() { ... }

@Override public void afterRestore(Context<? extends Resource> context) throws Exception { ... }

private boolean isPrime(final long number) {
    if (number < 1) { return false; }
    if (cache.containsKey(number)) { return cache.get(number); }
    boolean isPrime = true;
    for (long n = number ; n > 0 ; n--) {
        if (n != number && n != 1 && number % n == 0) {
            isPrime = false;
            break;
        }
    }
    cache.put(number, isPrime);
    return isPrime;
}
```

# JVM STARTUP DEMO

```java
public Main() { ... }

@Override public void afterRestore(Context<? extends Resource> context) throws Exception { ... }

private boolean isPrime(final long number) {
    if (number < 1) { return false; }
    if (cache.containsKey(number)) { return cache.get(number); }
    boolean isPrime = true;
    for (long n = number ; n > 0 ; n--) {
        if (n != number && n != 1 && number % n == 0) {
            isPrime = false;
            break;
        }
    }
    cache.put(number, isPrime);
    return isPrime;
}
```

# JVM STARTUP DEMO

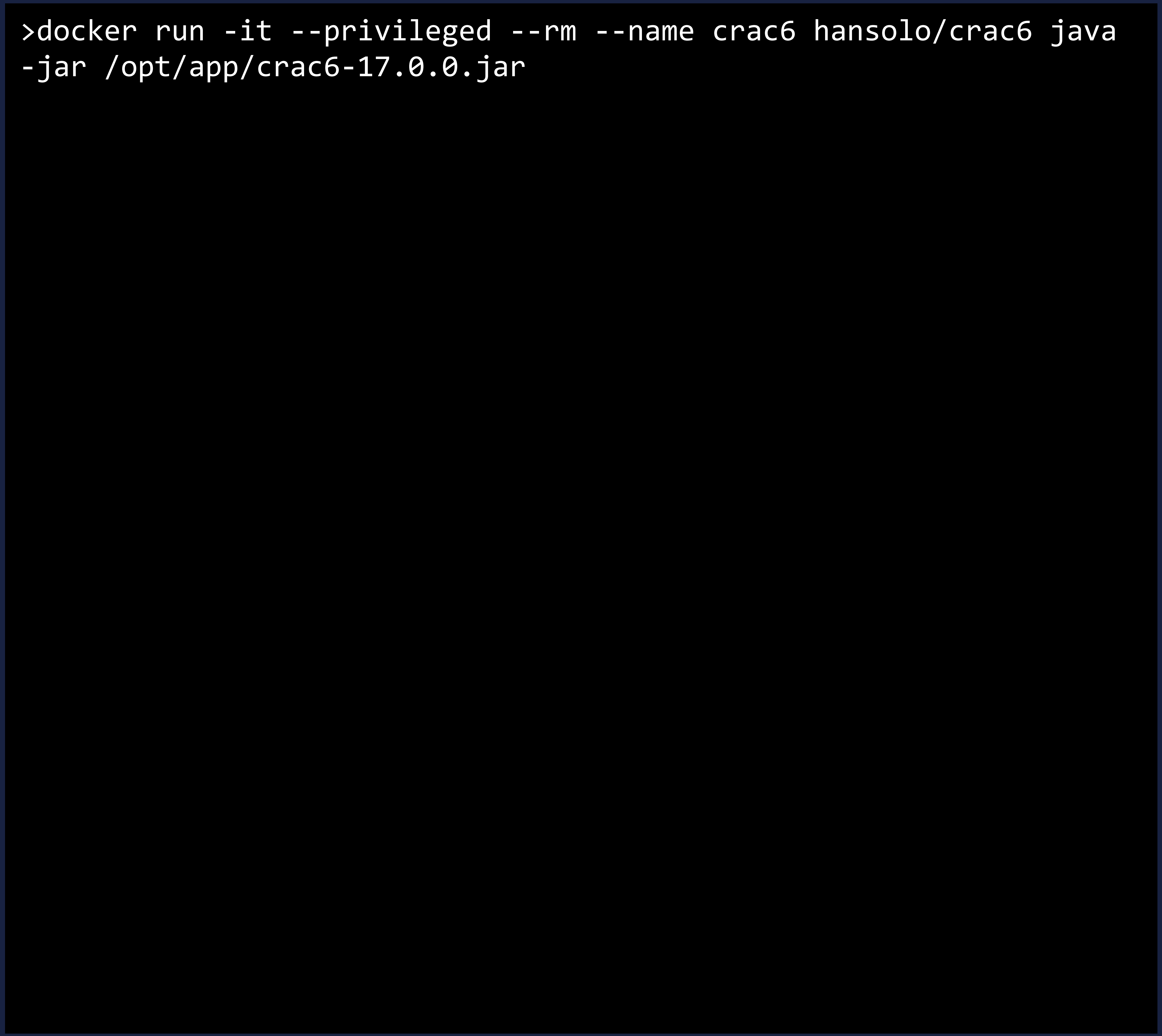
```
public Main() {
    Core.getGlobalContext().register(Main.this);
    final long start = System.nanoTime();

    // Loop emulates Application Startup and fills up the cache
    for (int i = 1 ; i < 50_000 ; i++) {
        isPrime(i);
    }

    isPrime(25000);
    System.out.println("Time to first response: " + ((System.nanoTime() - start) / 1_000_000) + " ms");
}

@Override public void afterRestore(Context<? extends Resource> context) throws Exception { ... }

private boolean isPrime(final long number) { ... }
```

# JVM STARTUP DEMO

```java
public Main() {
    Core.getGlobalContext().register(Main.this);
    final long start = System.nanoTime();

    // Loop emulates Application Startup and fills up the cache
    for (int i = 1 ; i < 50_000 ; i++) {
        isPrime(i);
    }

    isPrime(25000);
    System.out.println("Time to first response: " + ((System.nanoTime() – start) / 1_000_000) + " ms");
}

@Override public void afterRestore(Context<? extends Resource> context) throws Exception {
    System.out.println("afterRestore() called in Main");
    final long start = System.nanoTime();
    isPrime(25000);
    System.out.println("Time to first response: " + ((System.nanoTime() – start) / 1_000_000) + " ms");
}

private boolean isPrime(final long number) { ... }
```

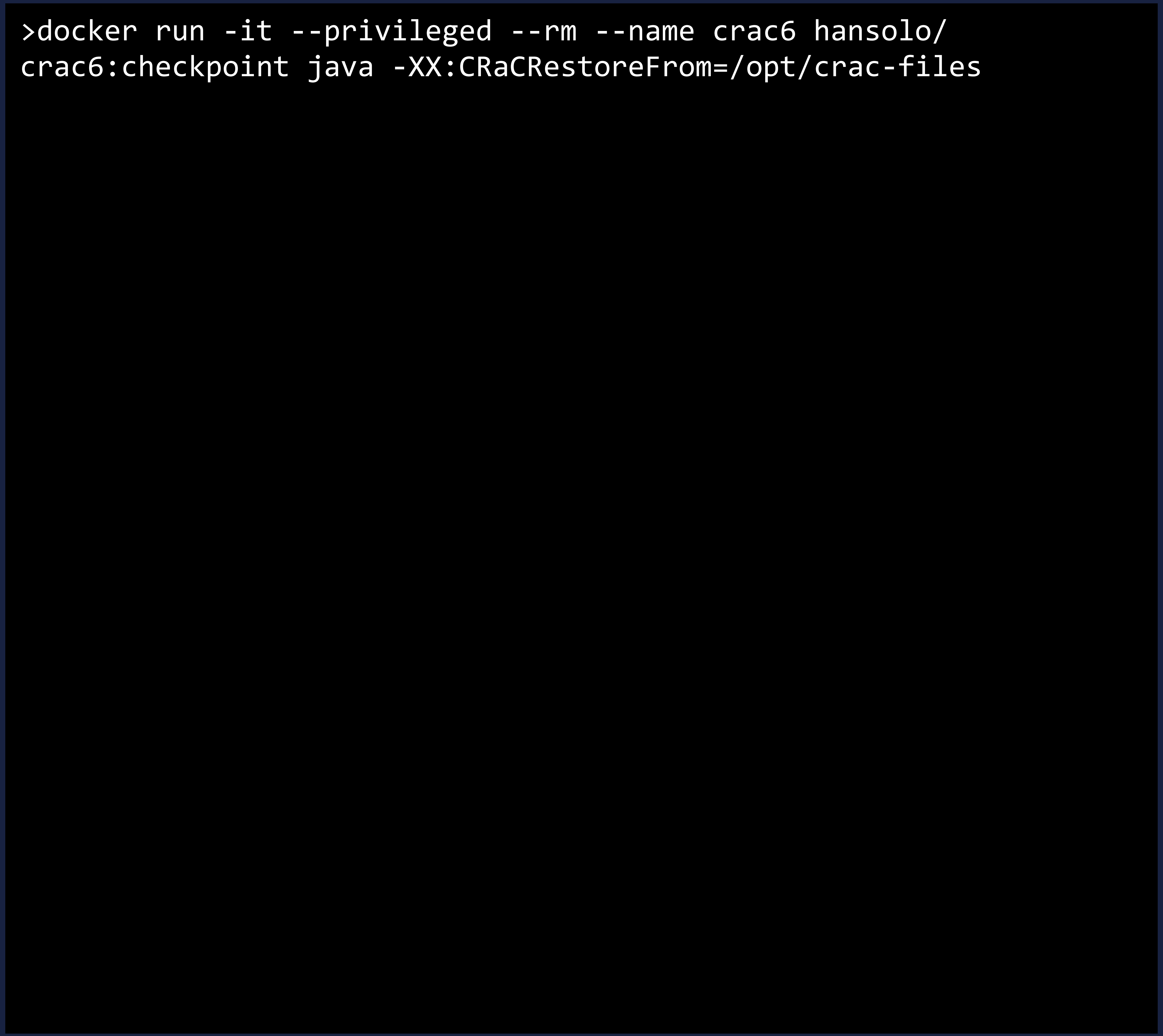
# JVM STARTUP DEMO

## SHELL 1



```
>docker run -it --privileged --rm --name crac6 hansolo/crac6 java
-jar /opt/app/crac6-17.0.0.jar
```

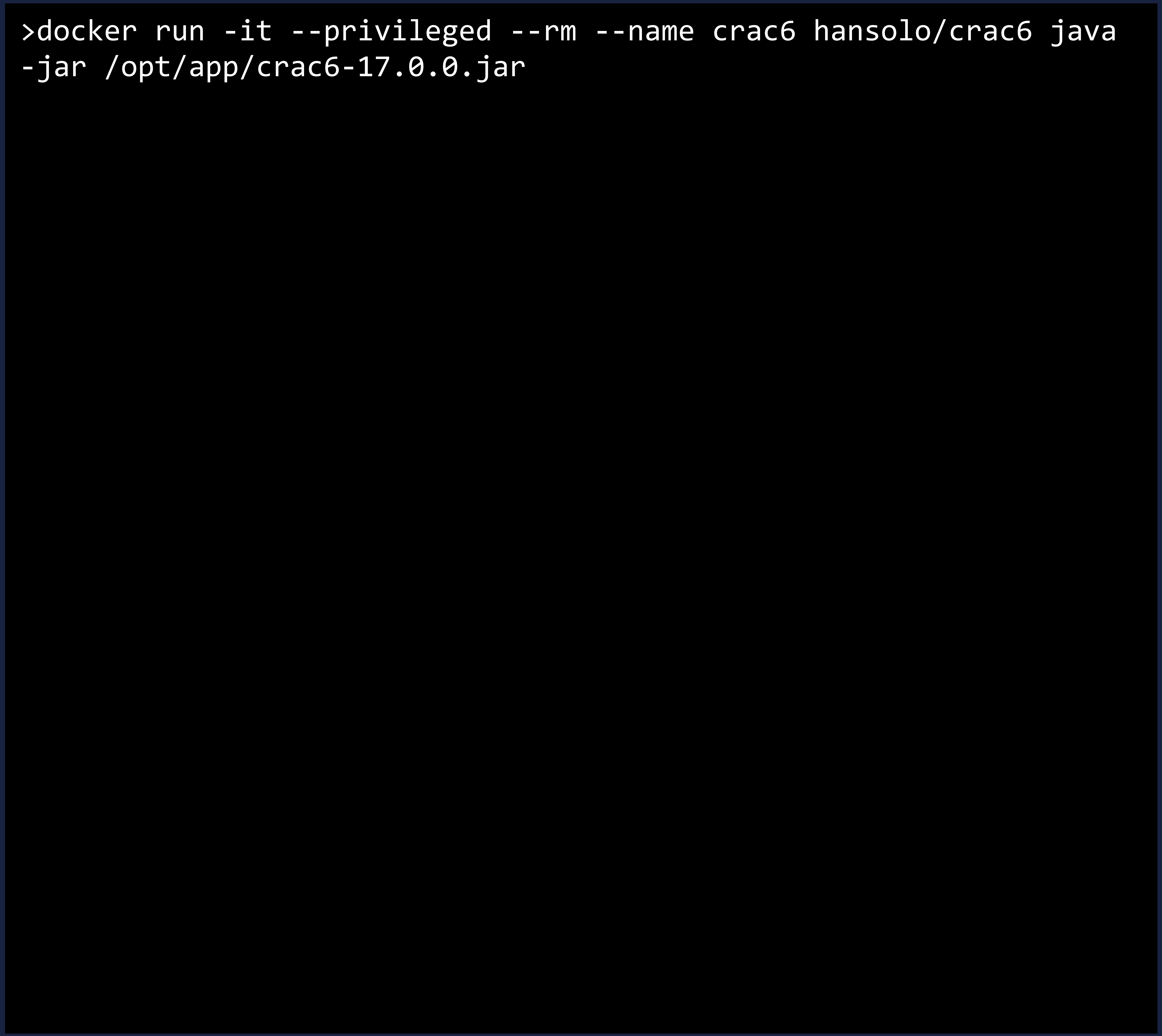
## SHELL 2



```
>docker run -it --privileged --rm --name crac6 hansolo/
crac6:checkpoint java -XX:CRaCRestoreFrom=/opt/crac-files
```

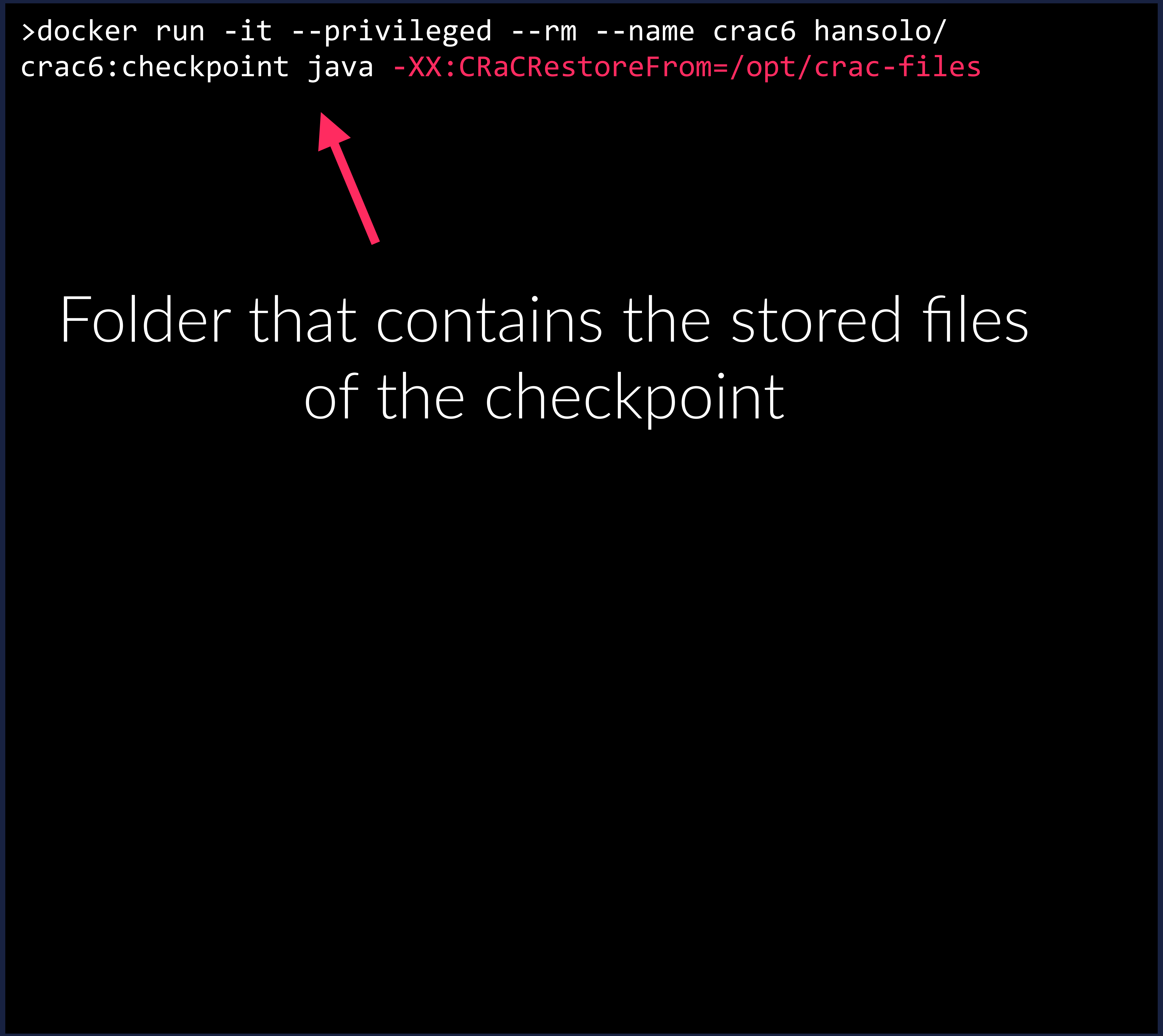
# JVM STARTUP DEMO

SHELL 1



```
>docker run -it --privileged --rm --name crac6 hansolo/crac6 java
-jar /opt/app/crac6-17.0.0.jar
```

SHELL 2



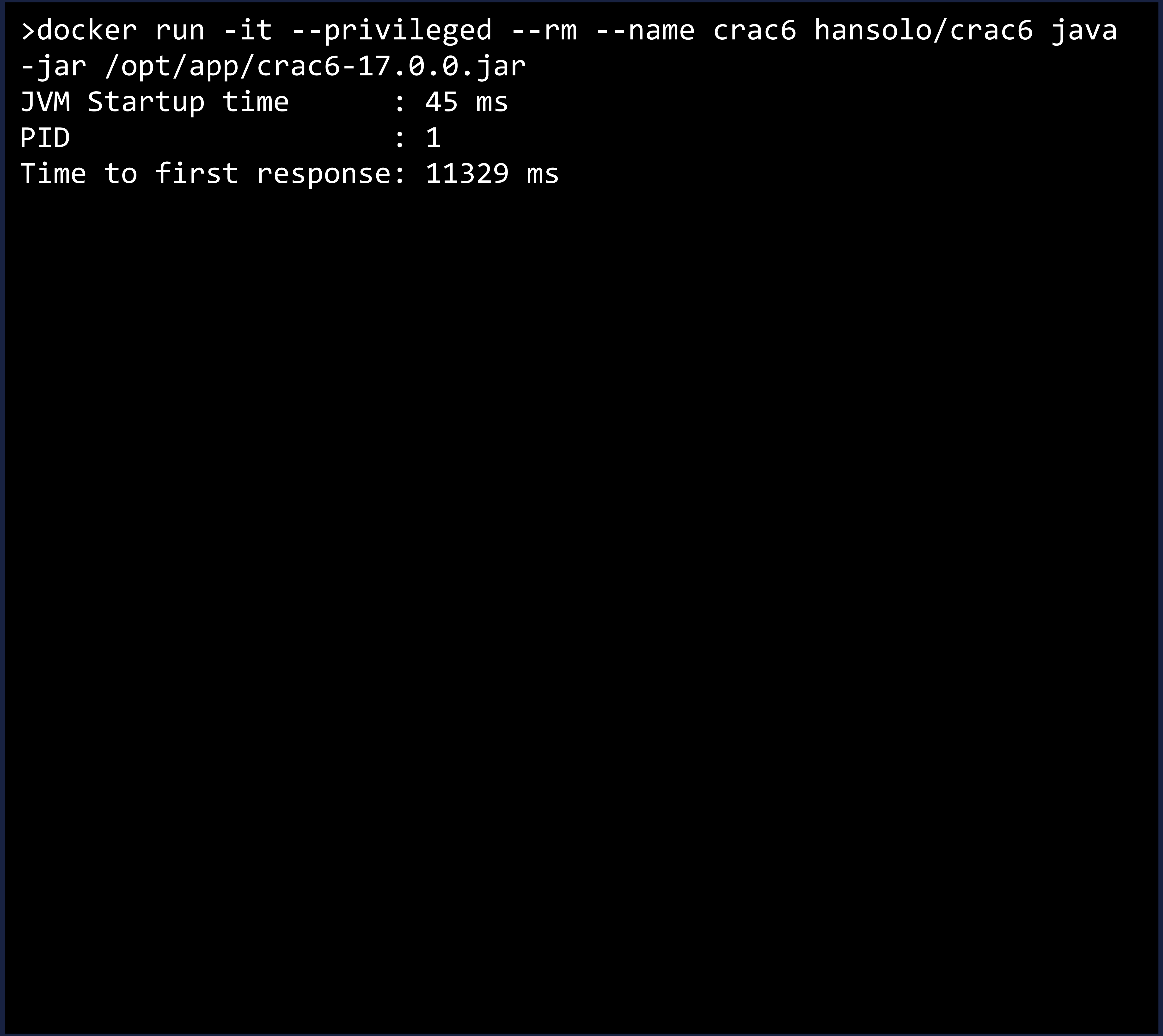
```
>docker run -it --privileged --rm --name crac6 hansolo/
crac6:checkpoint java -XX:CRaCRestoreFrom=/opt/crac-files
```

Folder that contains the stored files of the checkpoint

azul

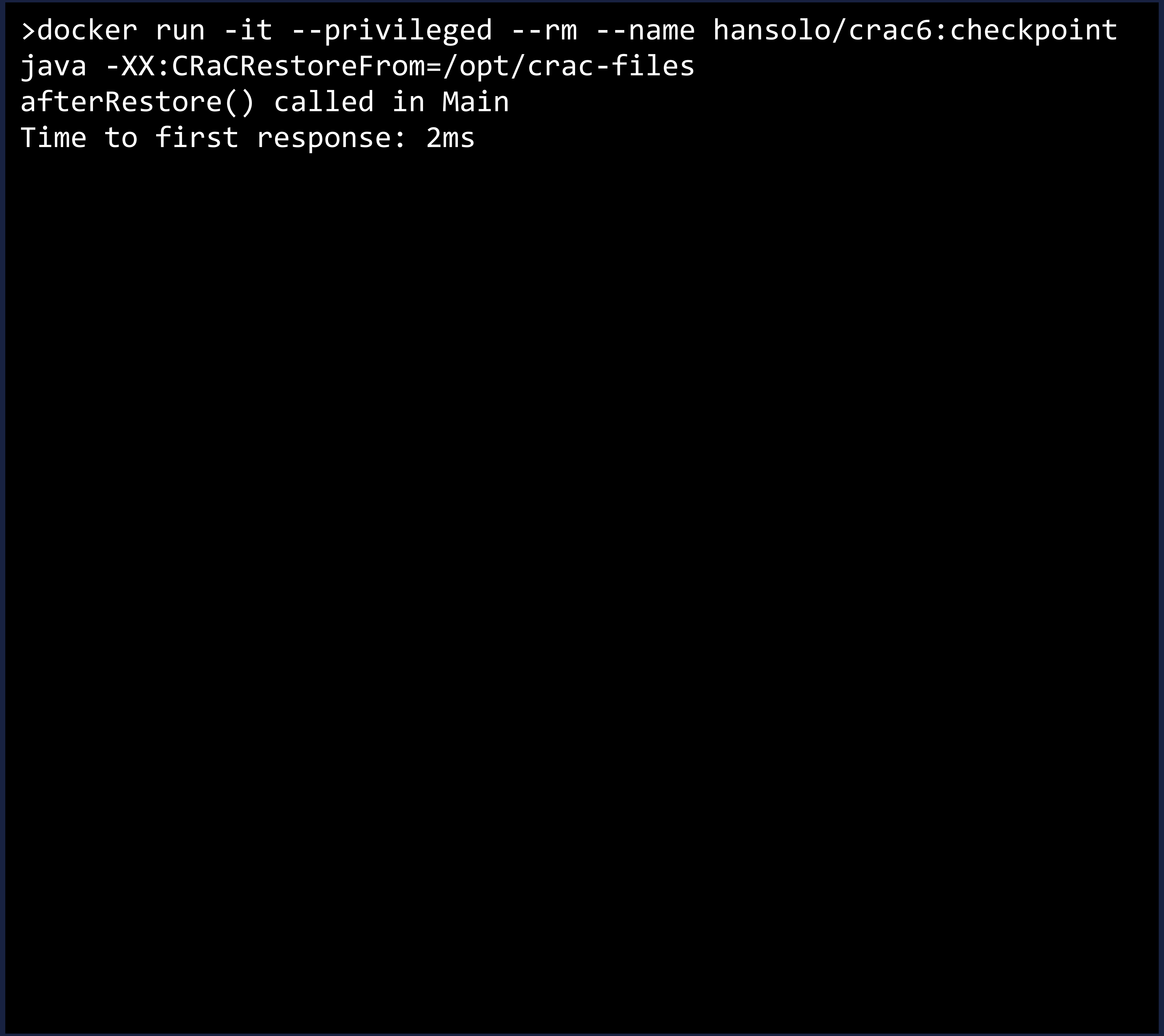
# JVM STARTUP DEMO

## SHELL 1



```
>docker run -it --privileged --rm --name crac6 hansolo/crac6 java
-jar /opt/app/crac6-17.0.0.jar
JVM Startup time       : 45 ms
PID                    : 1
Time to first response: 11329 ms
```

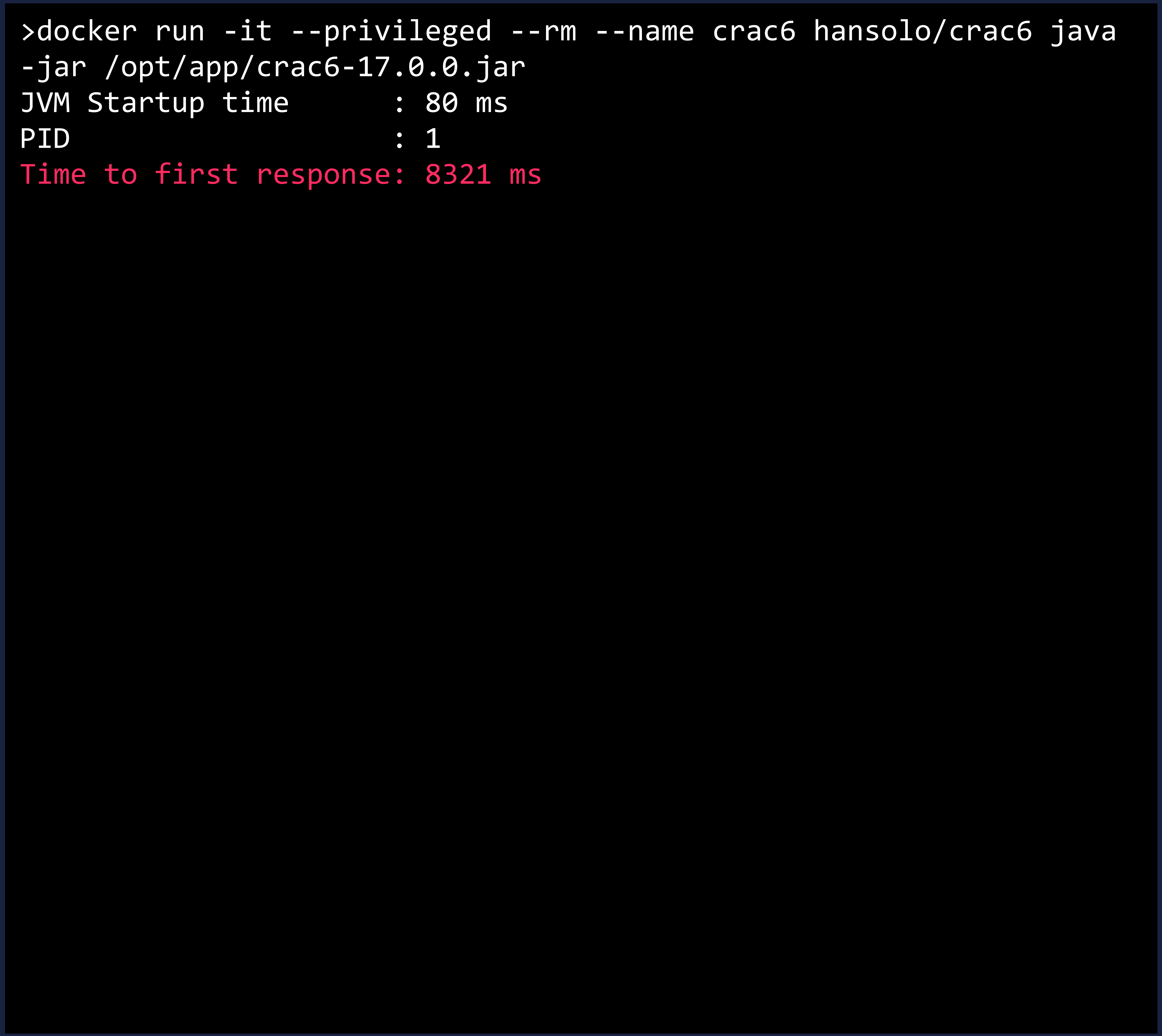
## SHELL 2



```
>docker run -it --privileged --rm --name hansolo/crac6:checkpoint
java -XX:CRaCRestoreFrom=/opt/crac-files
afterRestore() called in Main
Time to first response: 2ms
```

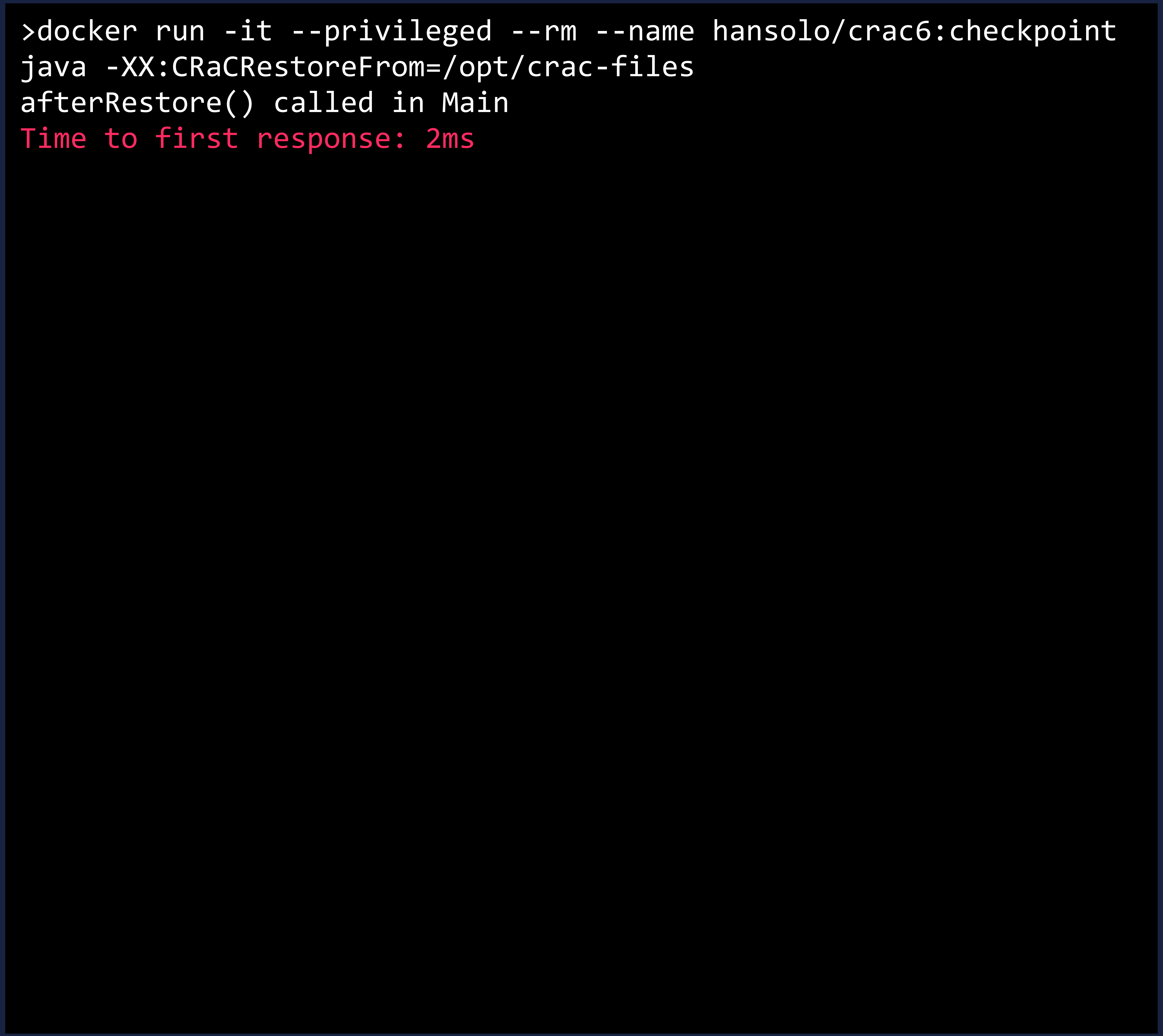
# JVM STARTUP DEMO

## SHELL 1



```
>docker run -it --privileged --rm --name crac6 hansolo/crac6 java
-jar /opt/app/crac6-17.0.0.jar
JVM Startup time       : 80 ms
PID                    : 1
Time to first response: 8321 ms
```

## SHELL 2



```
>docker run -it --privileged --rm --name hansolo/crac6:checkpoint
java -XX:CRaCRestoreFrom=/opt/crac-files
afterRestore() called in Main
Time to first response: 2ms
```

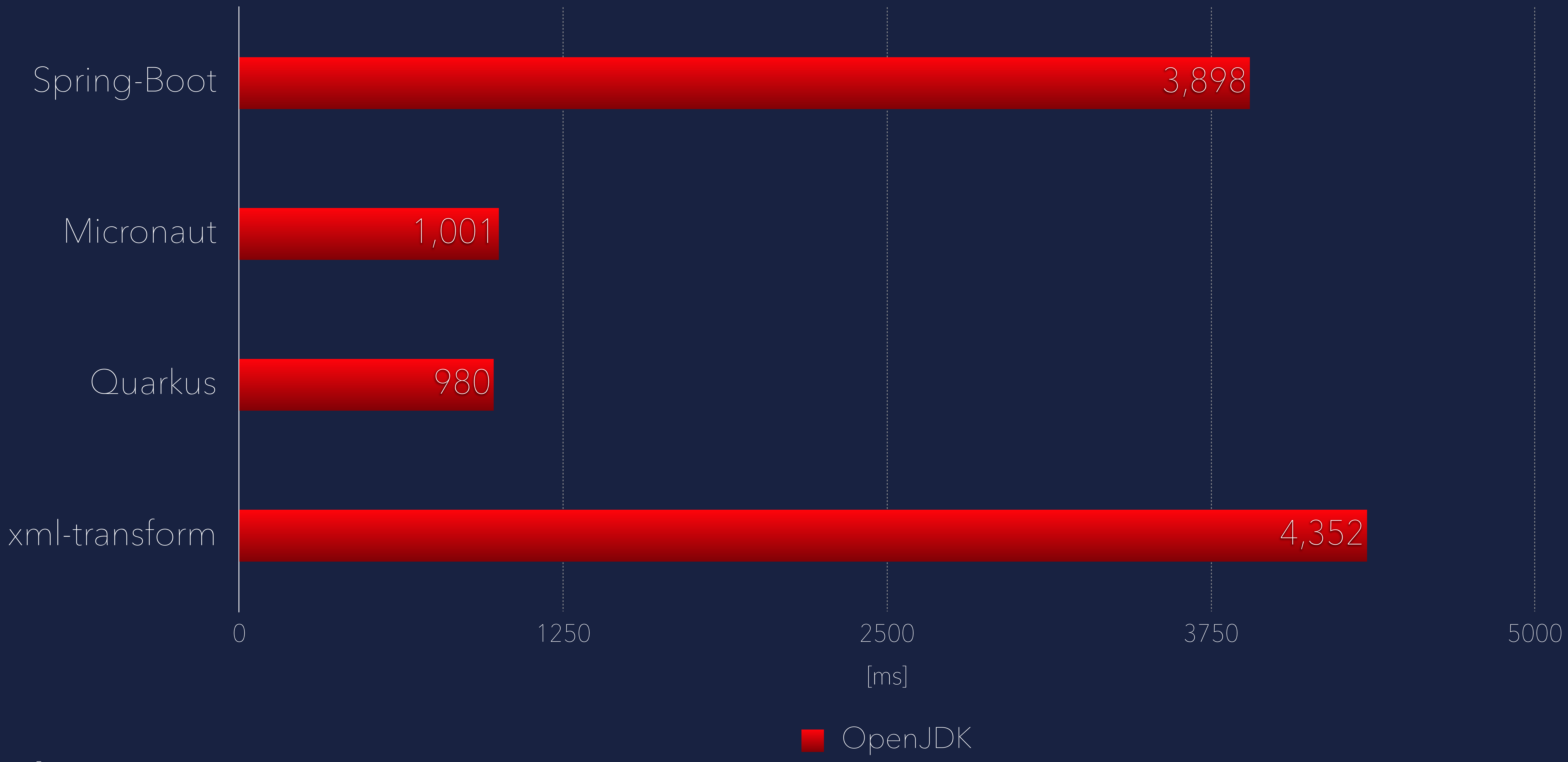
azul

github.com/HanSolo/crac6

# OK…BUT

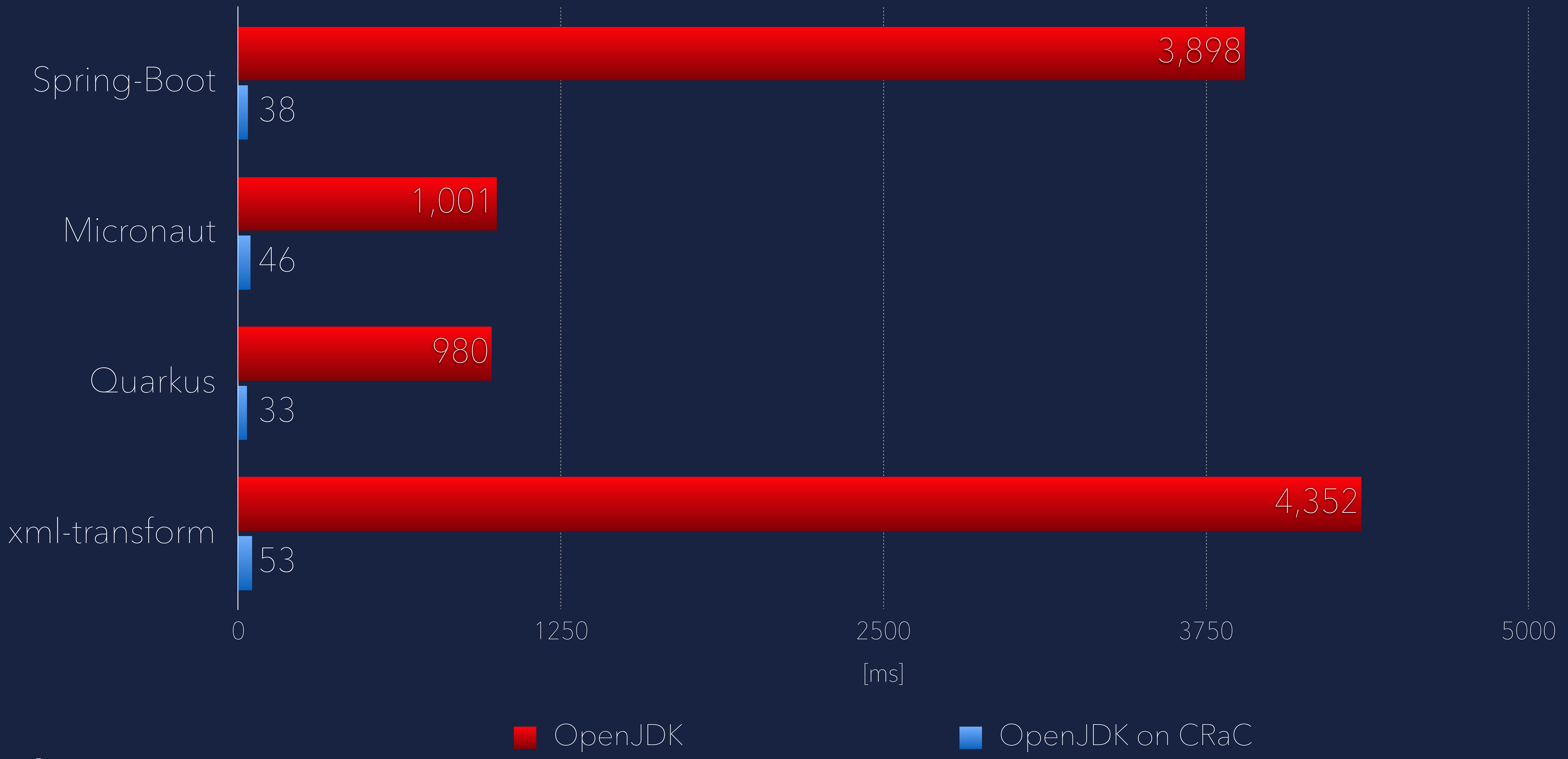
# HOW GOOD IS IT…?

# Time to first operation



azul

# Time to first operation

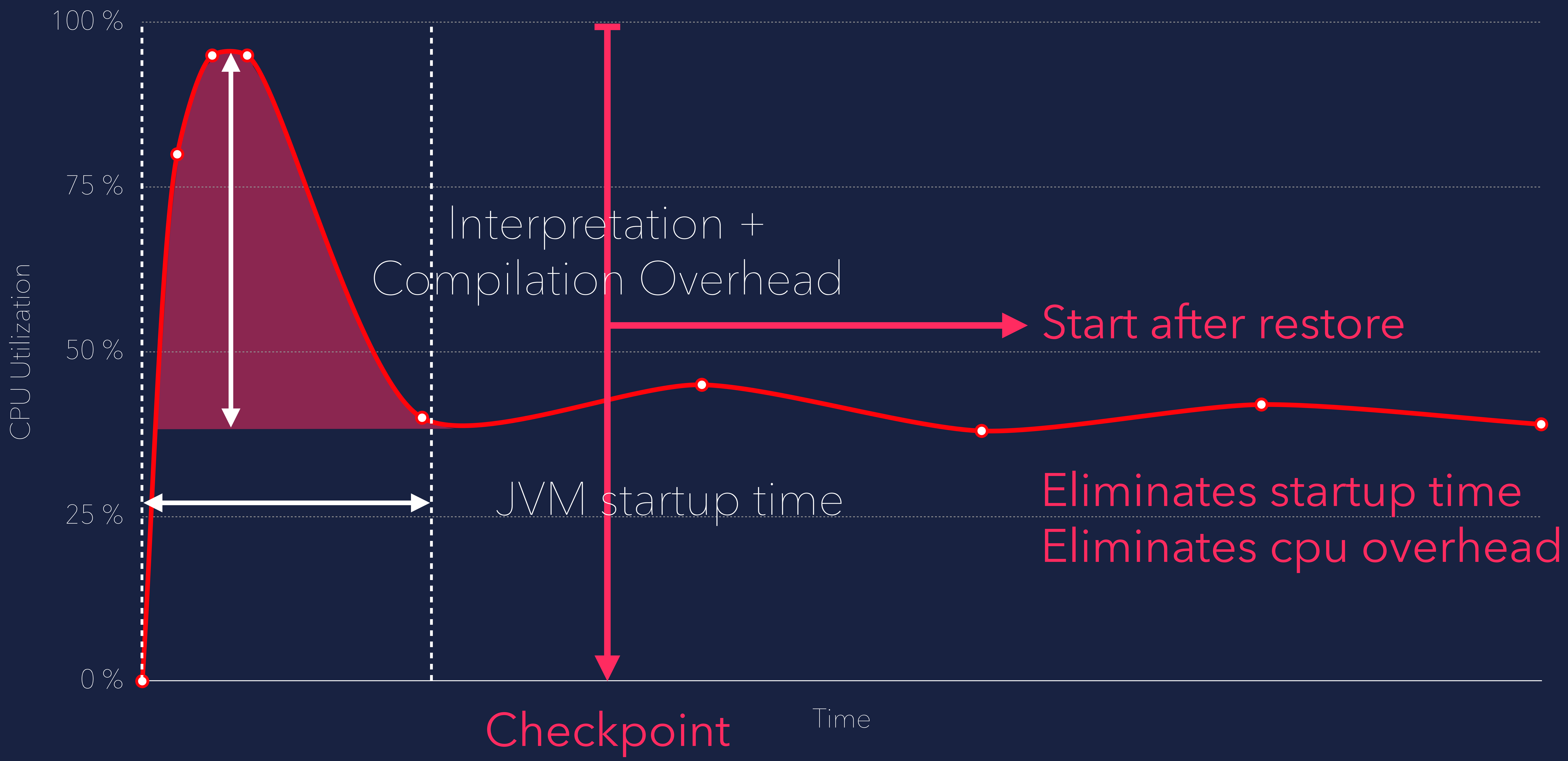


# SUMMARY…

# SUMMARY...

- CRaC is a way to pause and restore a JVM based application
- It doesn't require a closed world as with a native image
- Benefit is potentially extremely fast time to full performance level
- Eleminates the need for hotspot identification, method compiles, recompiles and deoptimisations
- Improved throughput from start
- CRaC is an OpenJDK project
- CRaC can save infrastructure cost

# INFRASTRUCTURE COST



Eliminates startup time

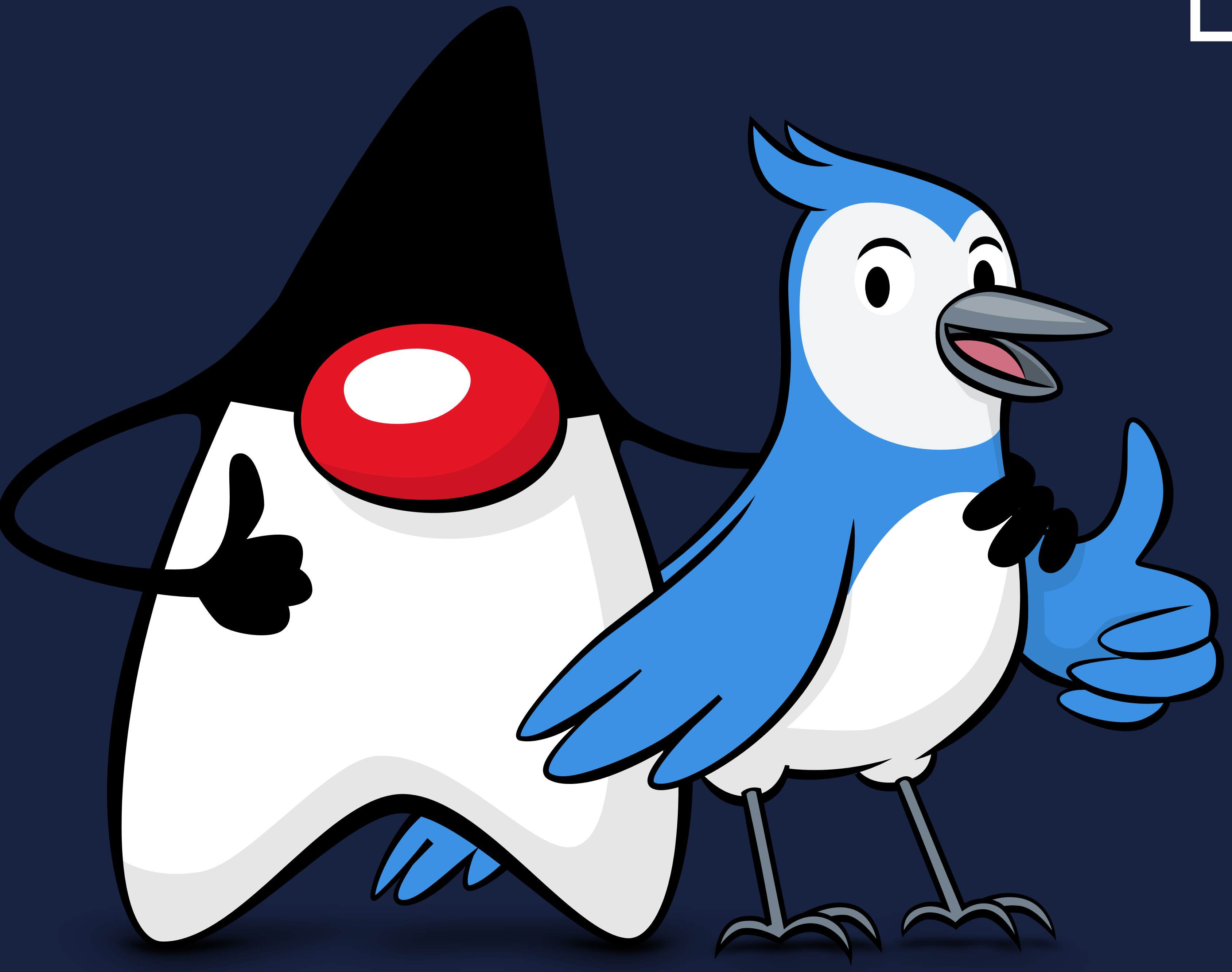
Eliminates cpu overhead

azul

# github.com/CRaC

# DEMO...

# THANK YOU



azul