What the CRaC...

Coordinated Restore at Checkpoint
on the Java Virtual Machine

azul

ABOUTME.

Gerrit Grunwald | Developer Advocate | Azul

SLIDES...

azul

JAVA IS GREAT...

azul

VIBRANT

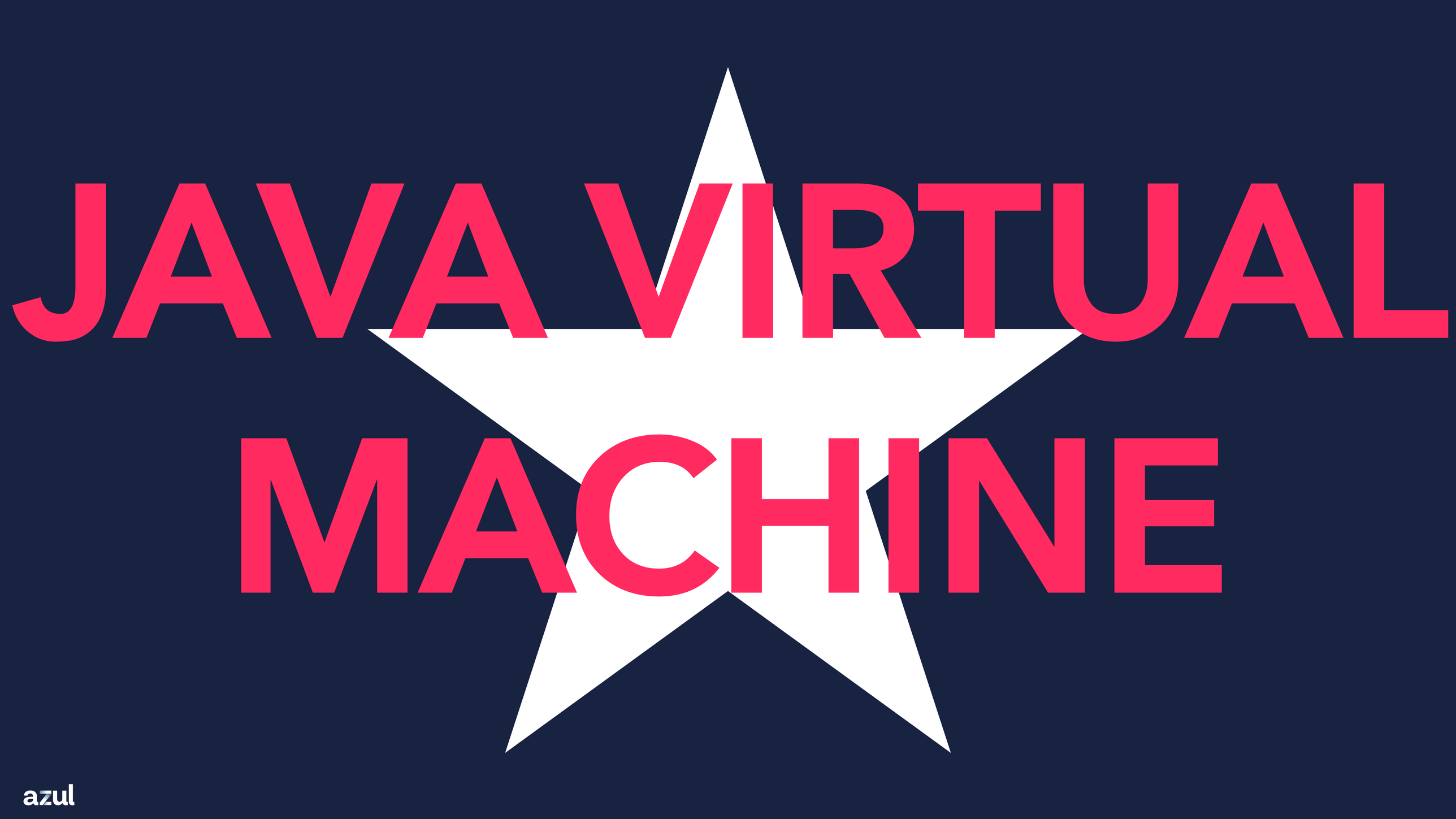
COMMUNITY...

azul

HUNDREDS OF JUGs...

azul

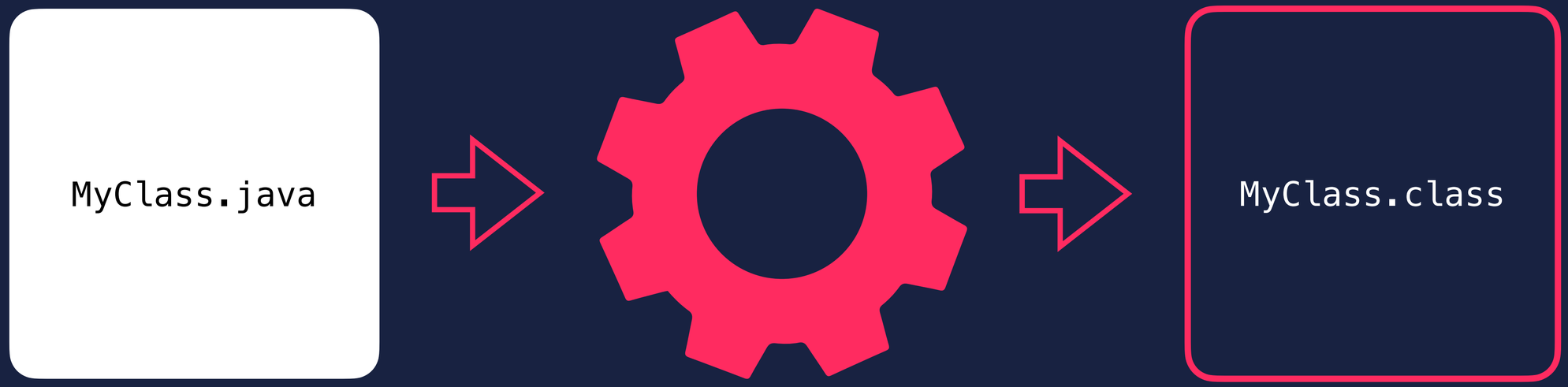
THOUSANDS OF FOSS PROJECTS...



azul

HOW DOES IT WORK...

azul



SOURCE CODE COMPILER BYTE CODE

BYTE CODE

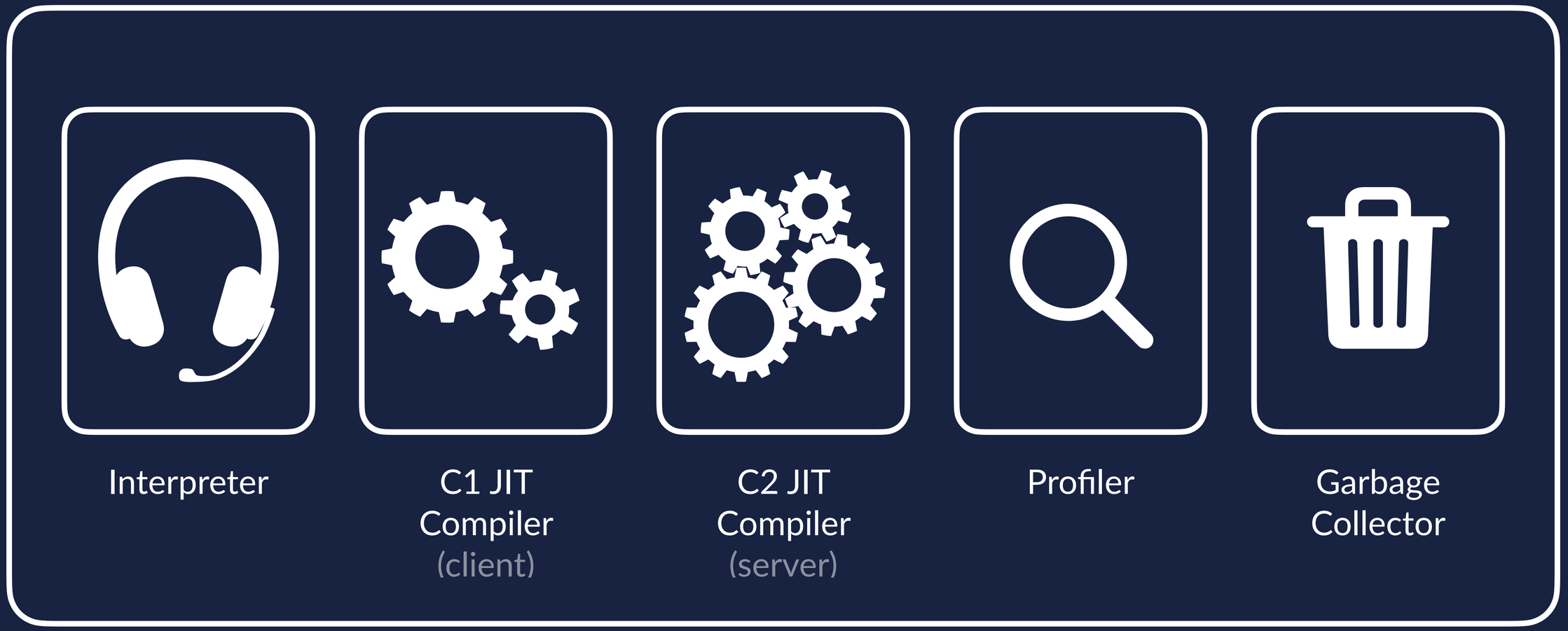
CLASS LOADER

JVM MEMORY

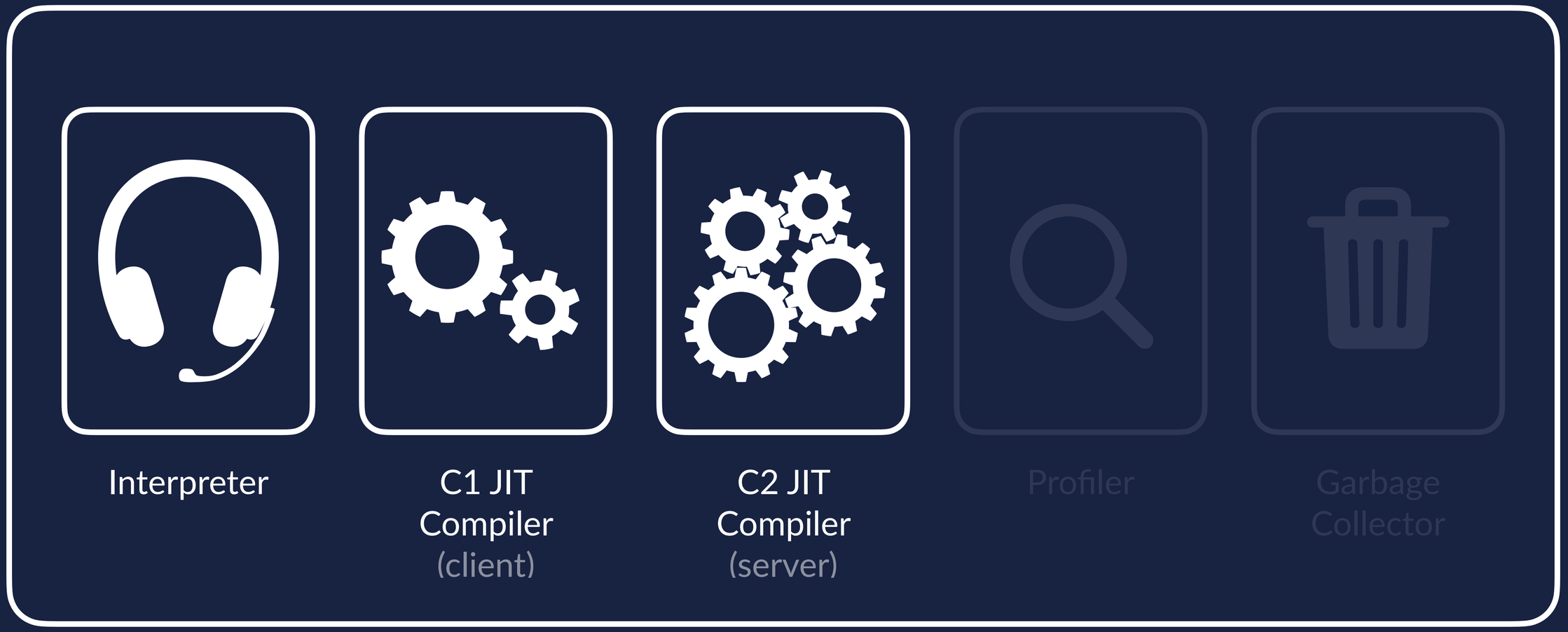
JVM MEMORY

EXECUTION ENGINE

EXECUTION ENGINE



EXECUTION ENGINE



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

JVM

Compiles code as quickly
as possible with low optimisation

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

JVM

Compiles code with best
optimisation possible (slower)

C2 JIT
COMPILER

TIERED COMPILATION

azul

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, ~35% overhead)
- Level 4 - C2 compiled code (uses profile data from previous steps)

LEVELS OF EXECUTION

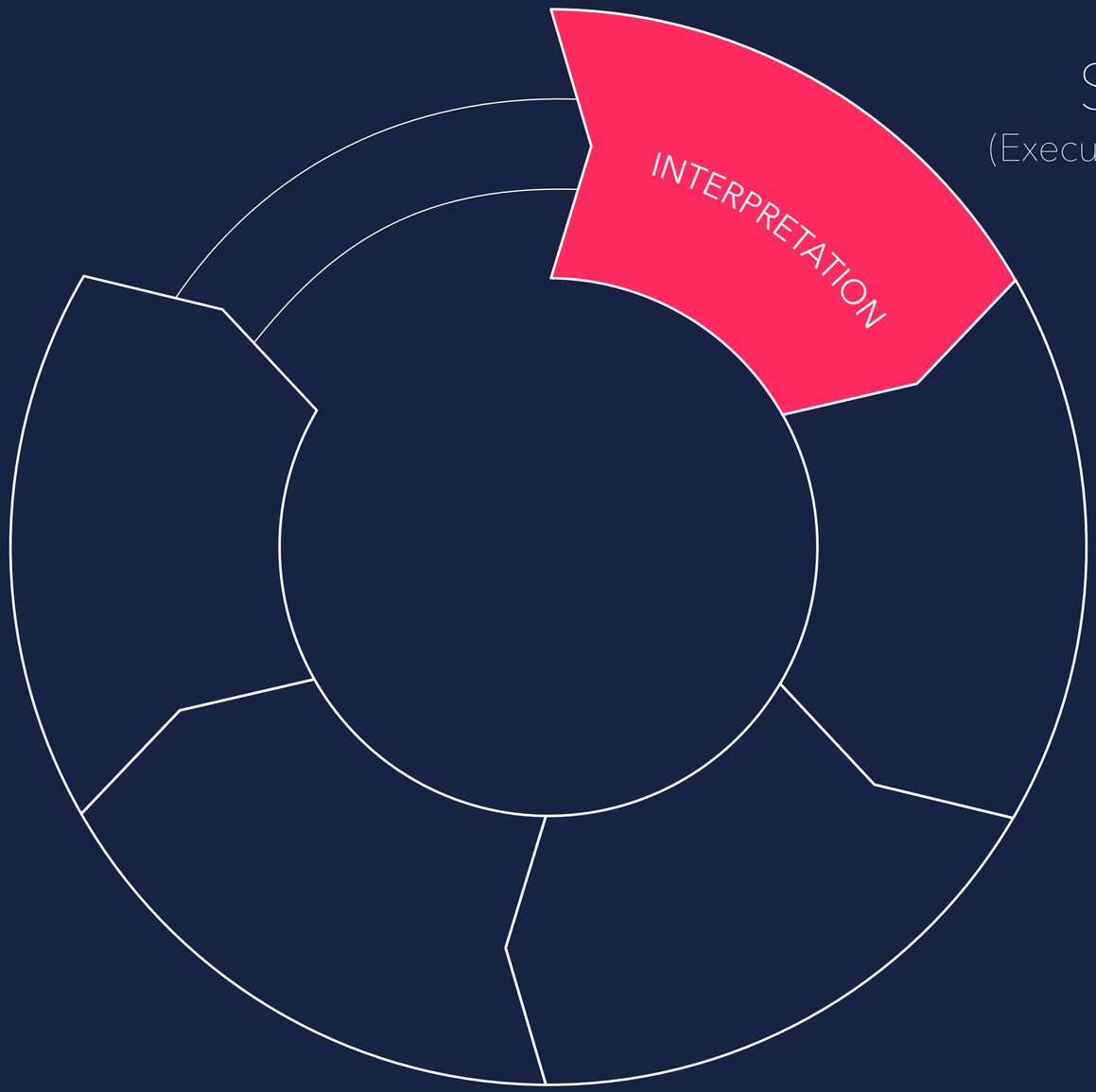
- Level 0 - Interpreted code
- Level 1 - C1 compiled code (no profiling)
- Level 2 - C1 compiled code (basic profiling)
- Level 3 - C1 compiled code (full profiling, ~35% overhead)
- Level 4 - C2 compiled code (uses profile data from previous steps)

PREFERRED

EXECUTION CYCLE

azul

EXECUTION CYCLE

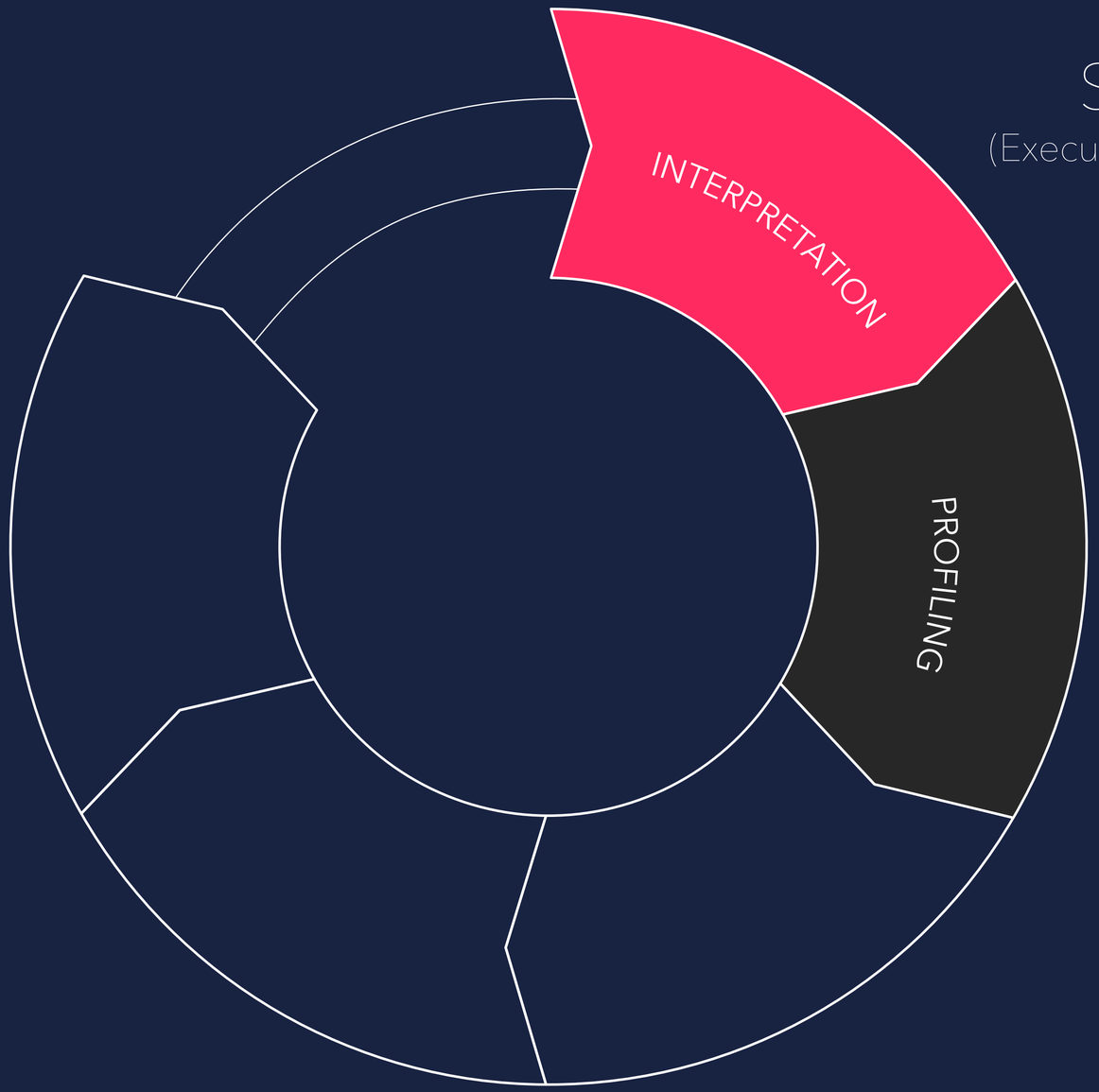


Slow

(Execution Level 0)

azul

EXECUTION CYCLE

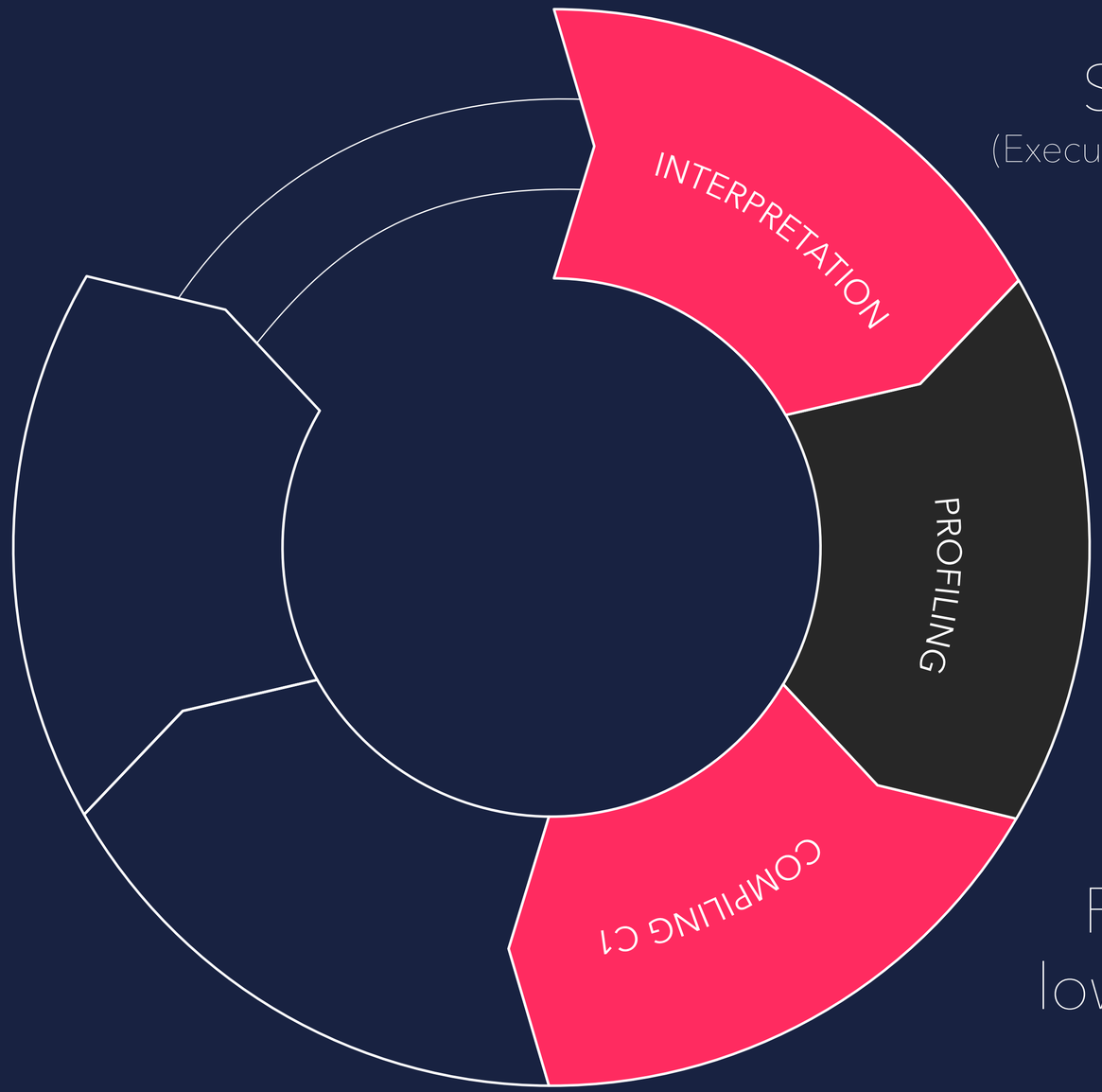


Slow
(Execution Level 0)

Finding
"hotspots"

azul

EXECUTION CYCLE



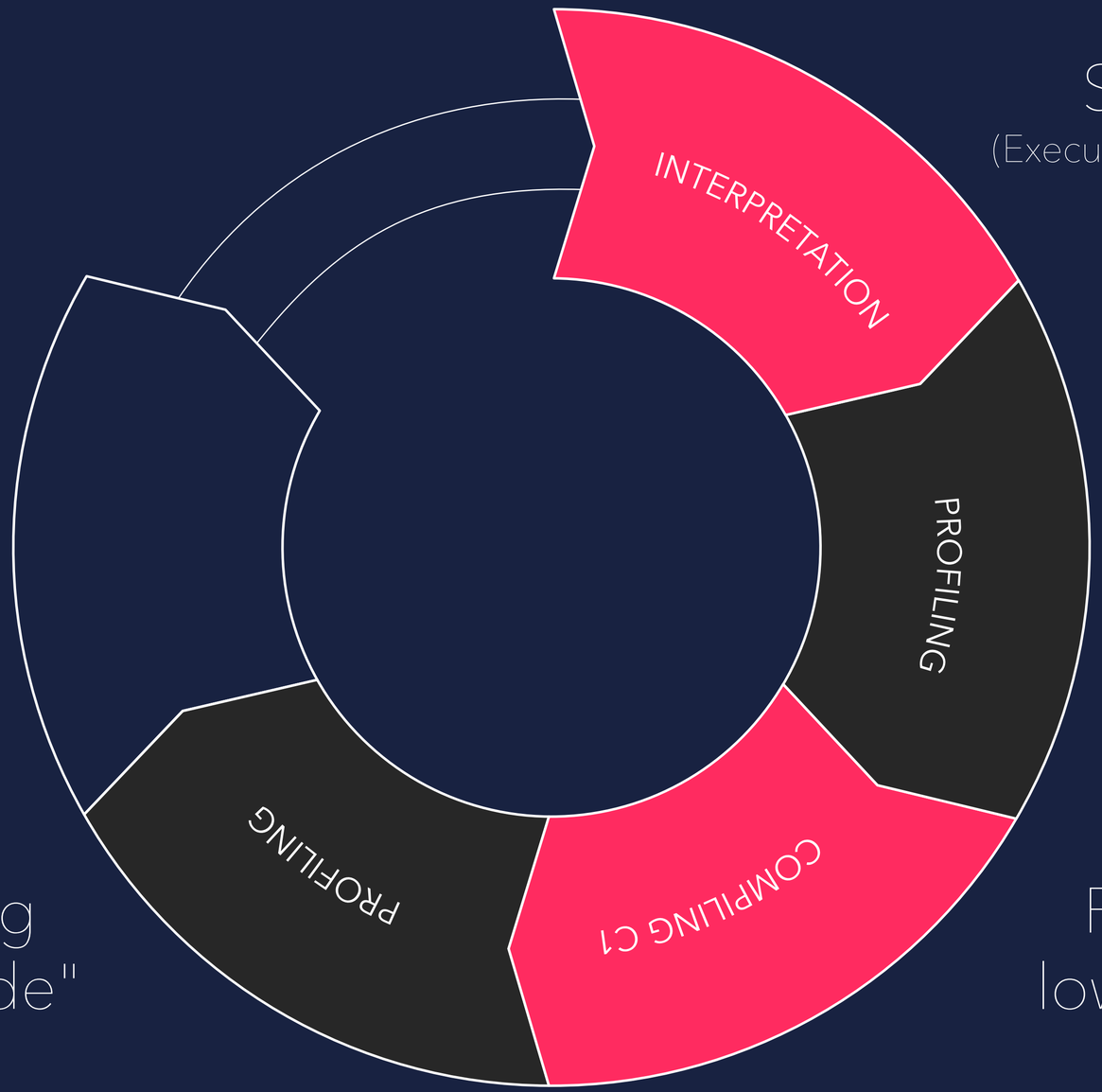
Slow
(Execution Level 0)

Finding
"hotspots"

Fast compile,
low optimisation
(Execution Level 3)

azul

EXECUTION CYCLE



Slow
(Execution Level 0)

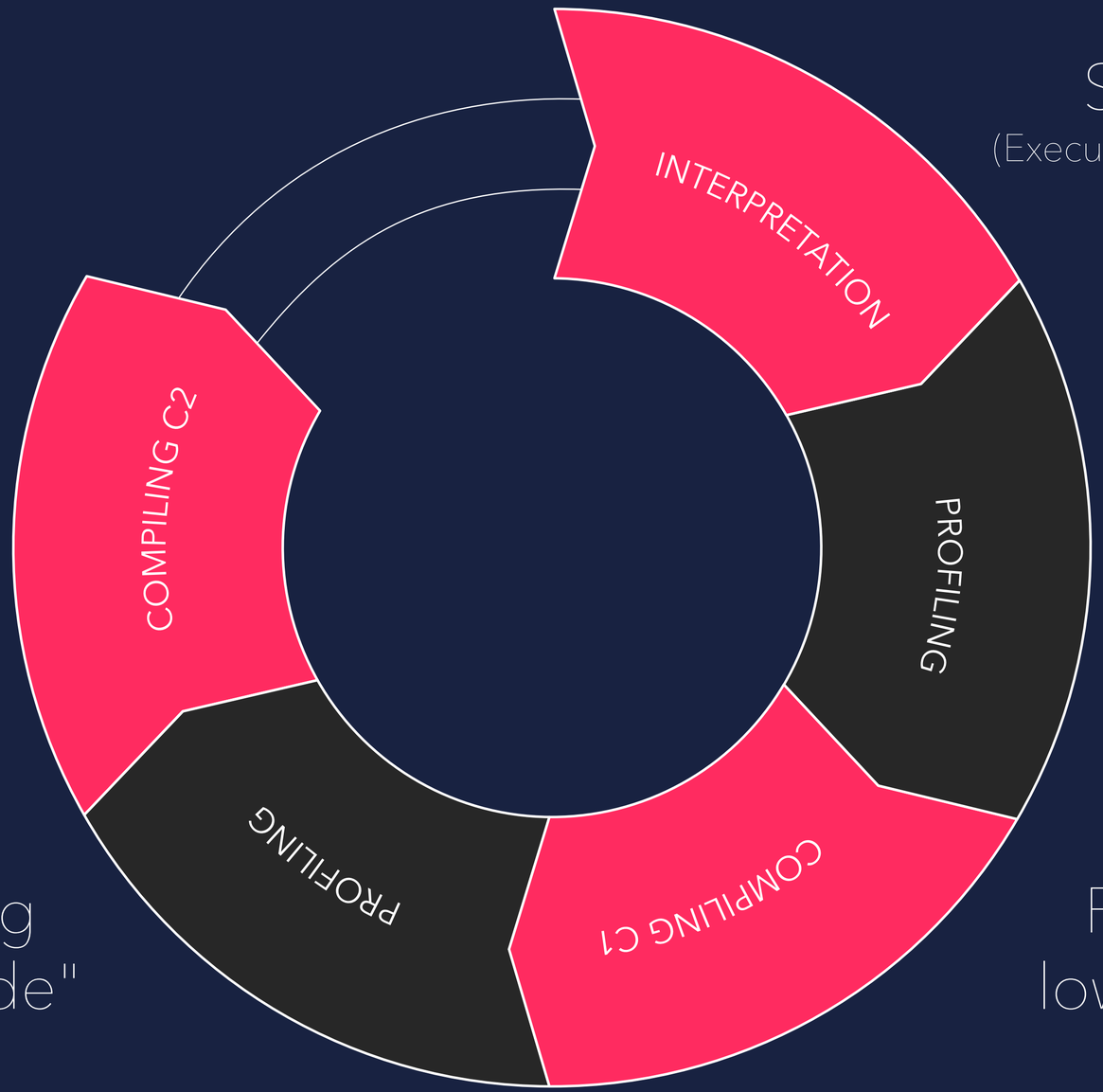
Finding
"hotspots"

Fast compile,
low optimisation
(Execution Level 3)

Finding
"hot code"

azul

EXECUTION CYCLE



Slow
(Execution Level 0)

Finding
"hotspots"

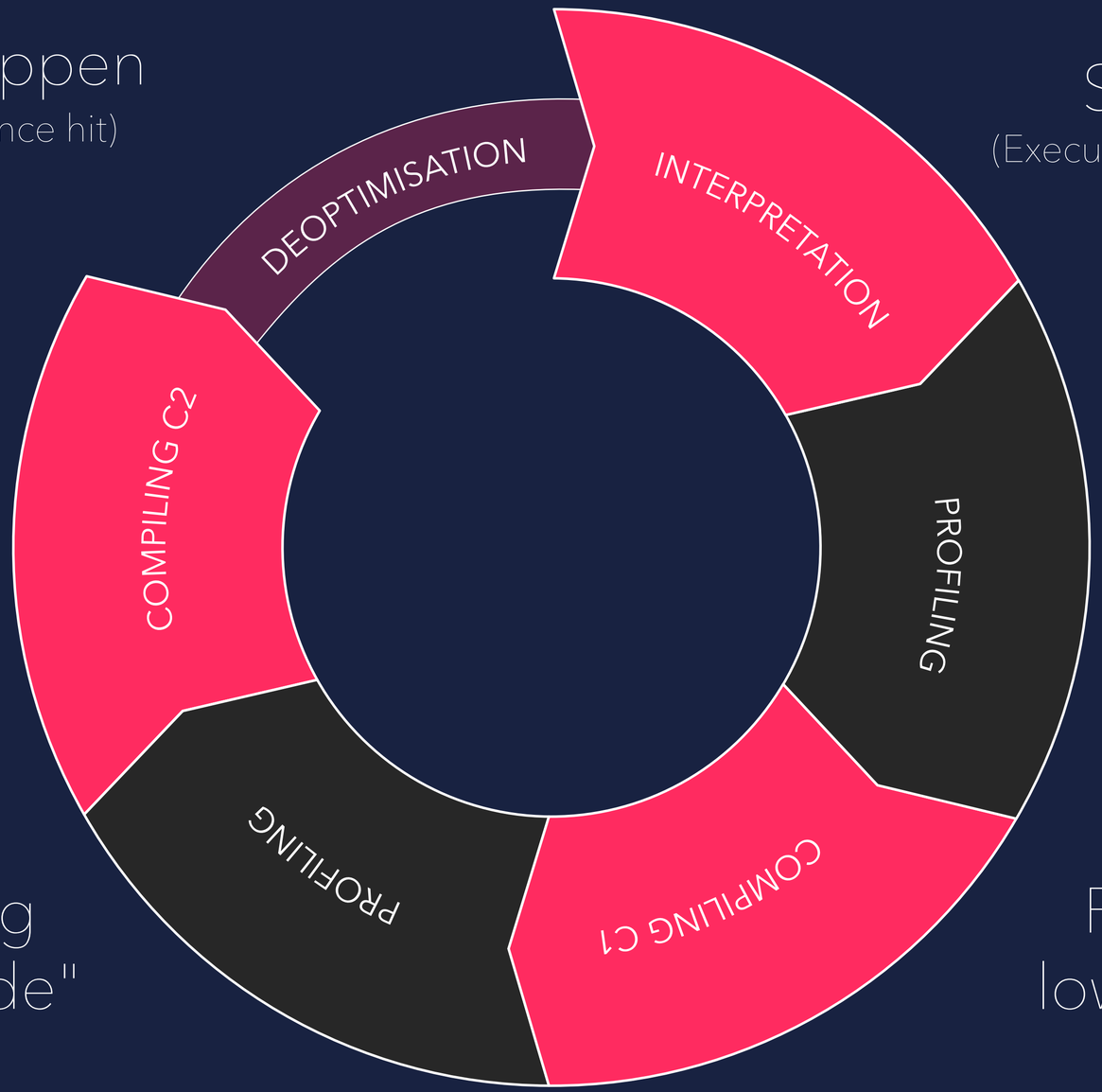
Fast compile,
low optimisation
(Execution Level 3)

Finding
"hot code"

Slower compile,
high optimisation
(Execution Level 4)

azul

EXECUTION CYCLE



Slow
(Execution Level 0)

Finding
"hotspots"

Fast compile,
low optimisation
(Execution Level 3)

Finding
"hot code"

Slower compile,
high optimisation
(Execution Level 4)

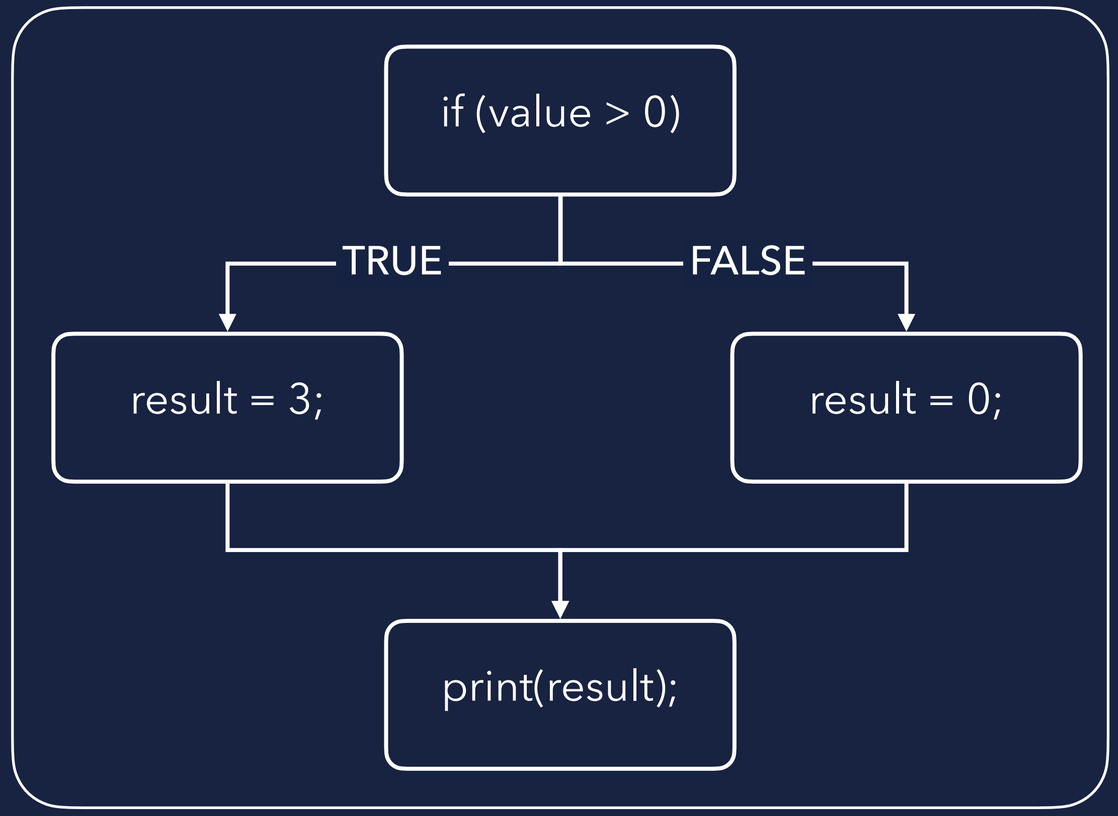
Can happen
(performance hit)

azul

DEOPTIMISATION

DEOPTIMISATION

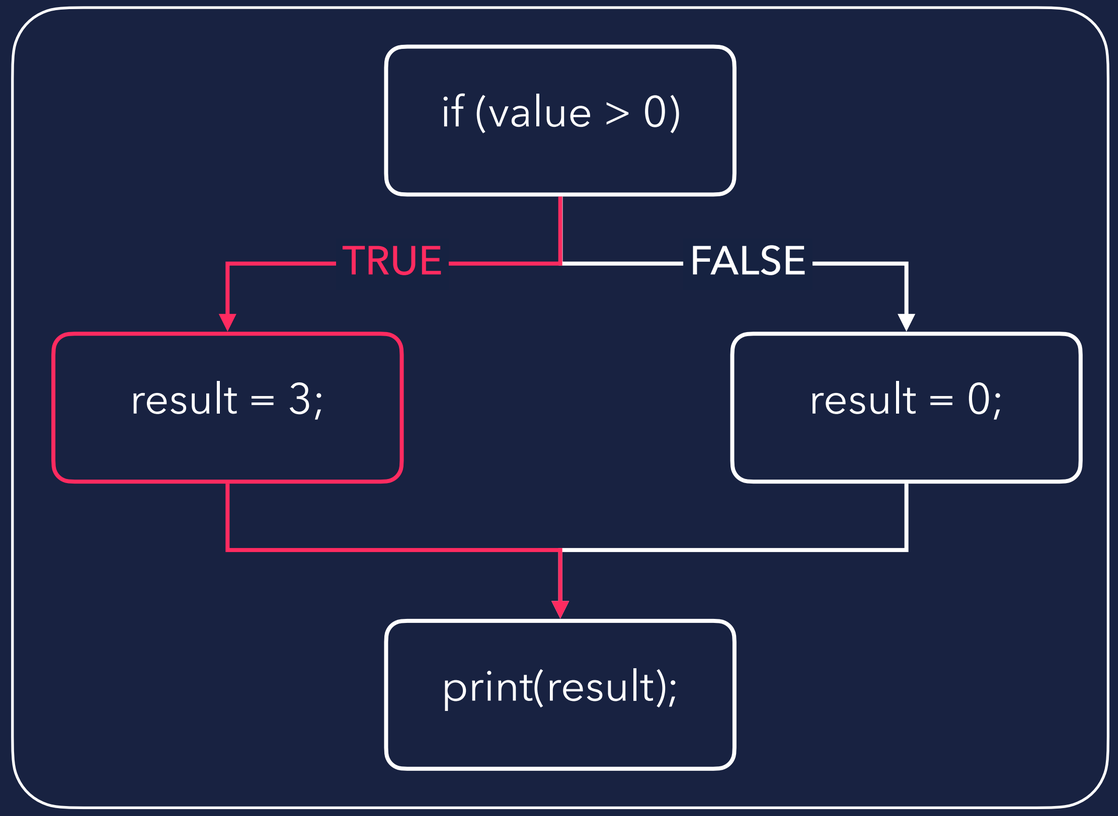
CODE



DEOPTIMISATION

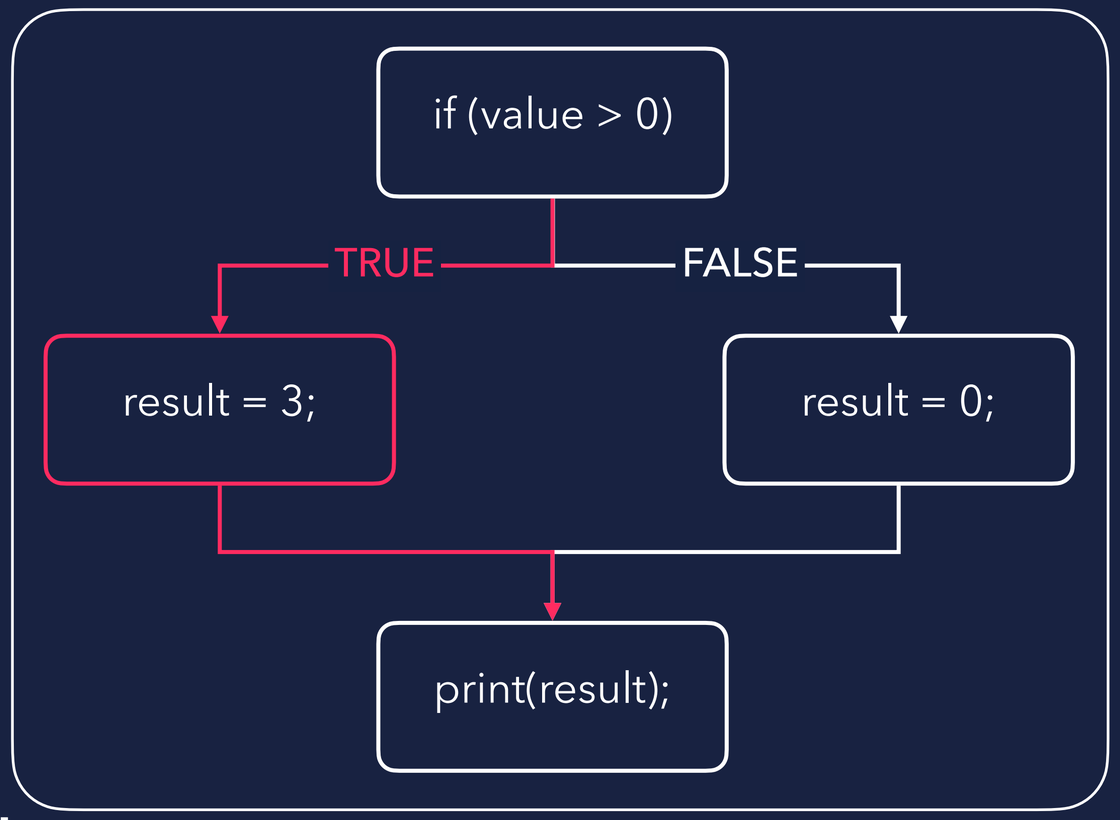
CODE

Hot Path (profiled)

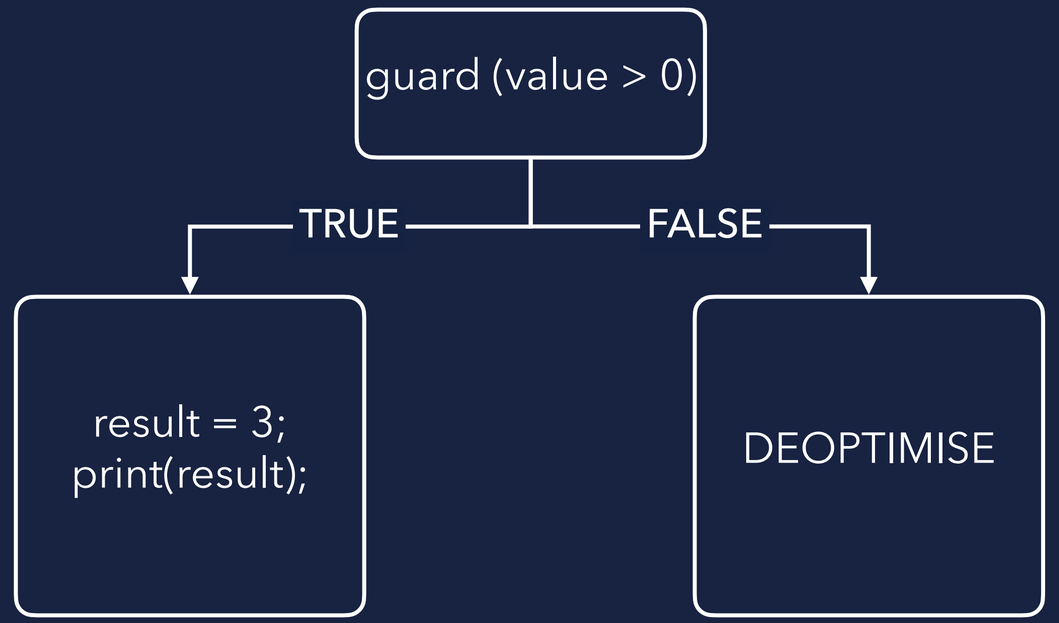


DEOPTIMISATION

CODE

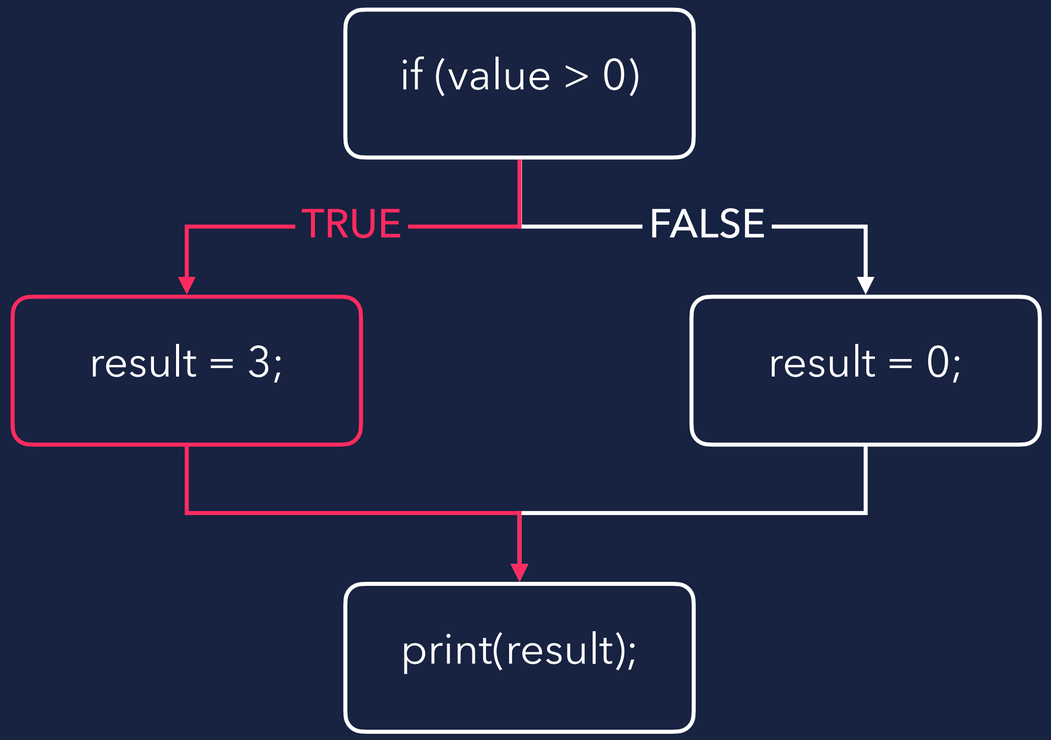


OPTIMIZED CODE



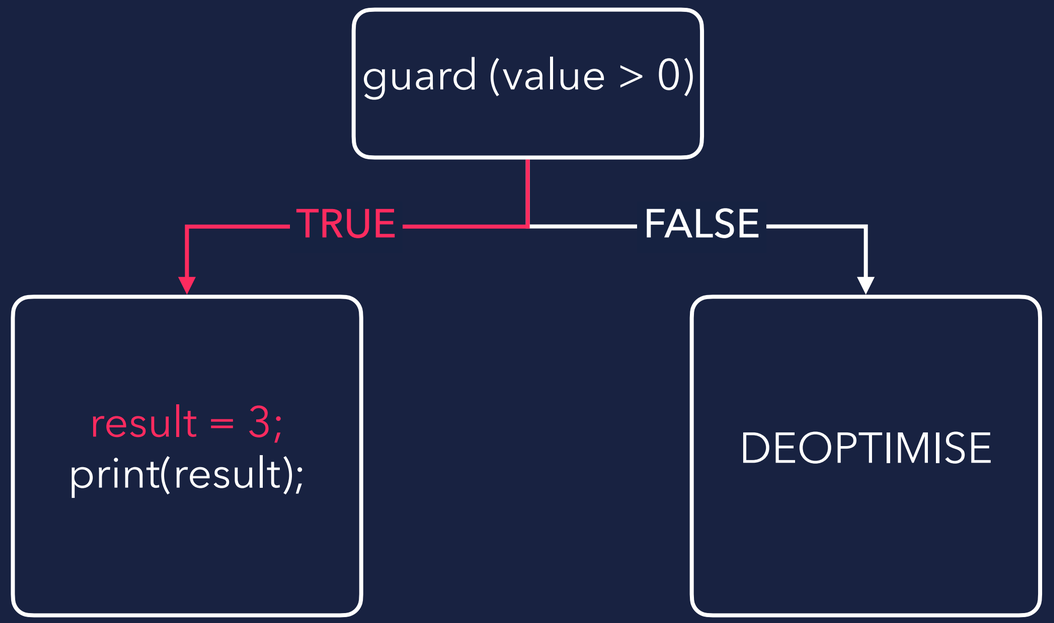
DEOPTIMISATION

CODE



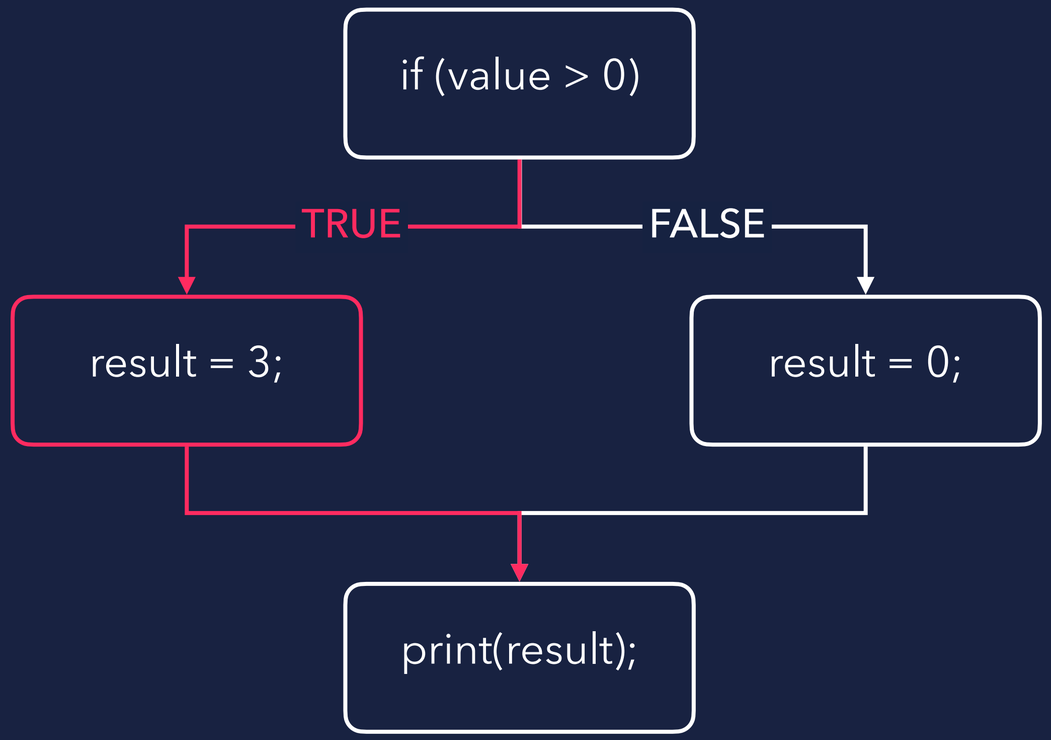
OPTIMIZED CODE

value = 3



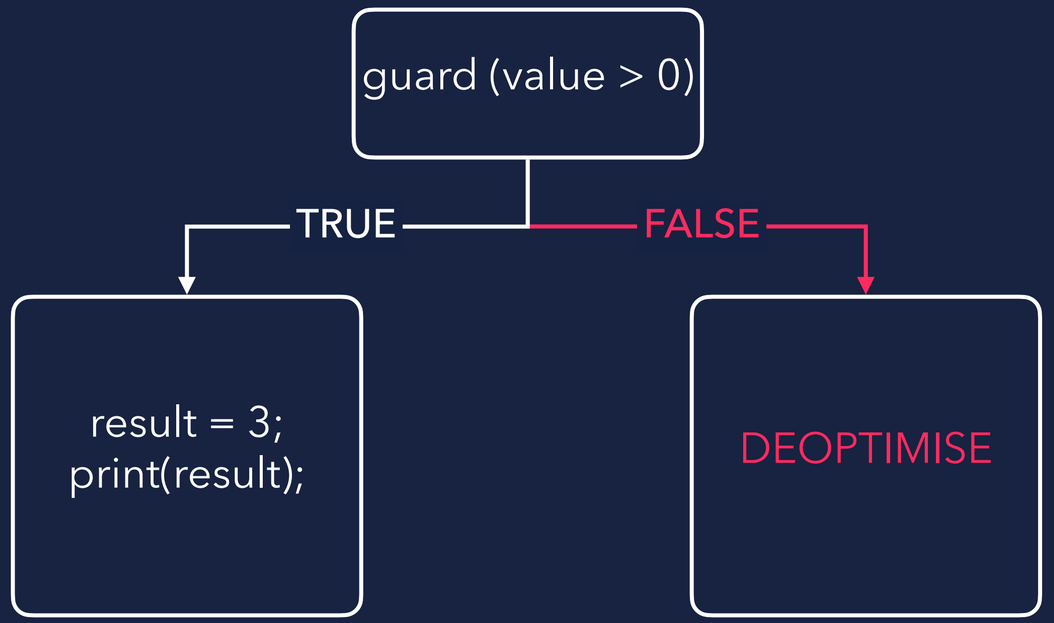
DEOPTIMISATION

CODE



OPTIMIZED CODE

value = 0



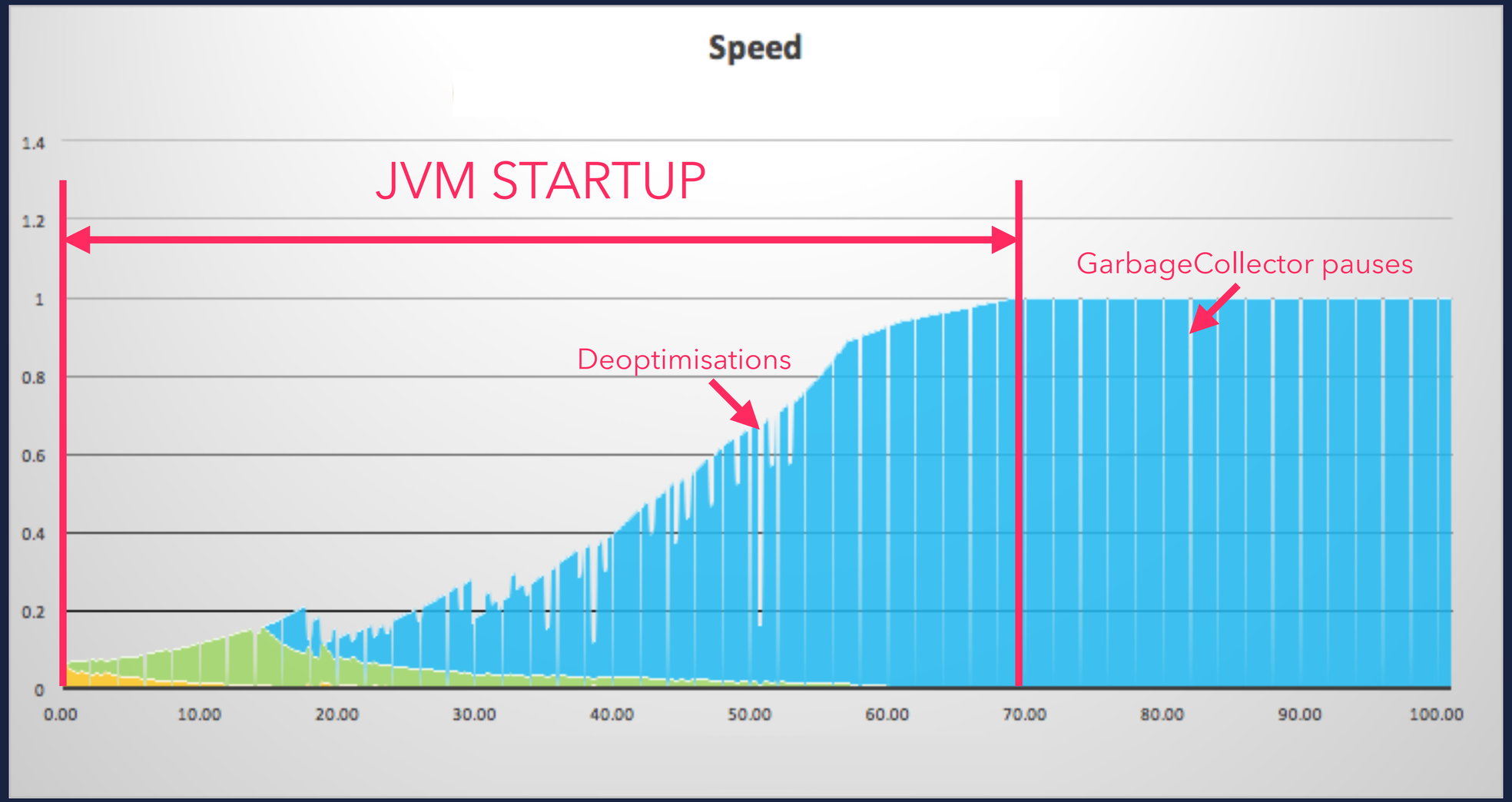
THAT'S GREAT...

azul

...BUT...

...IT TAKES TIME !

JVM PERFORMANCE GRAPH

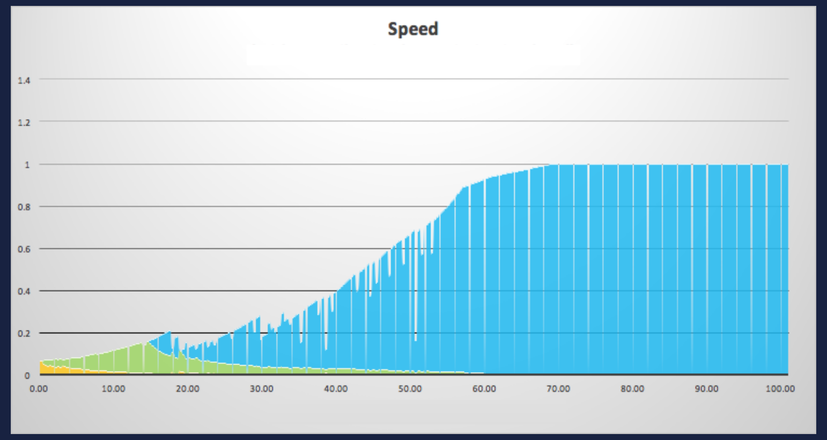


MICROSERVICE ENVIRONMENT

azul

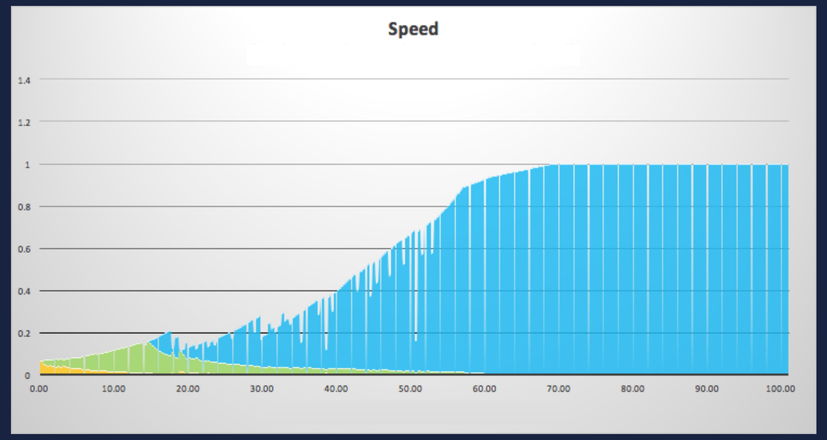
MICROSERVICE ENVIRONMENT

FIRST RUN



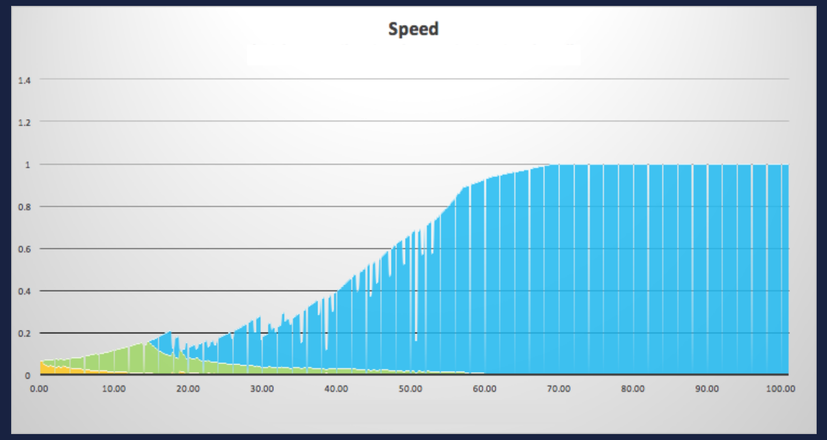
JVM STARTUP

SECOND RUN



JVM STARTUP

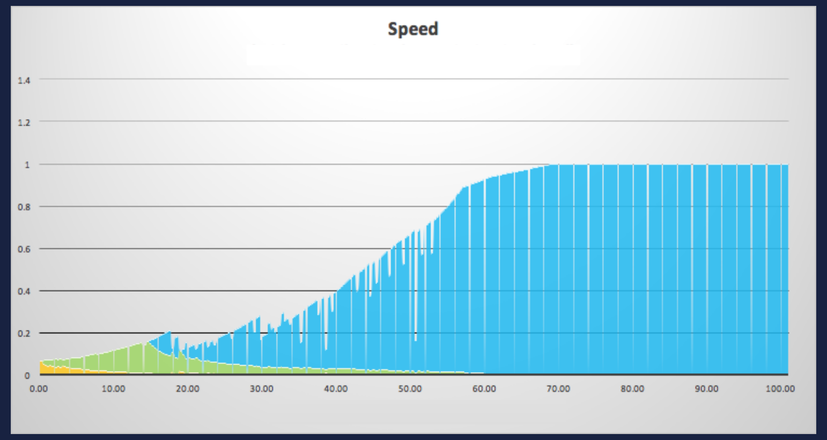
THIRD RUN



JVM STARTUP

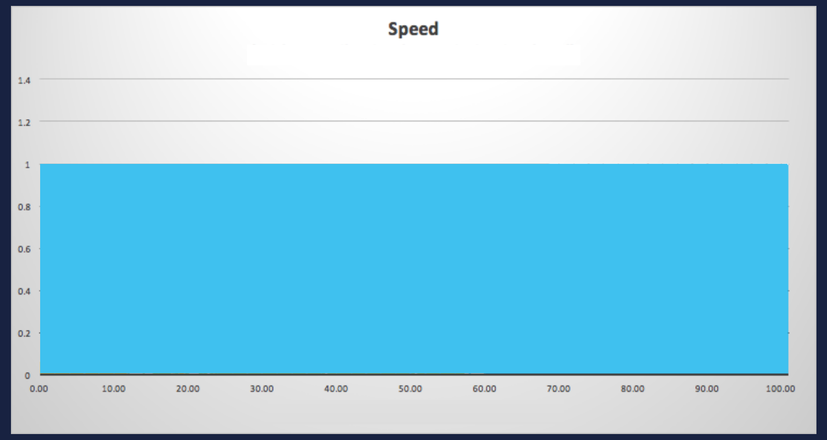
WOULDN'T IT BE GREAT...?

FIRST RUN



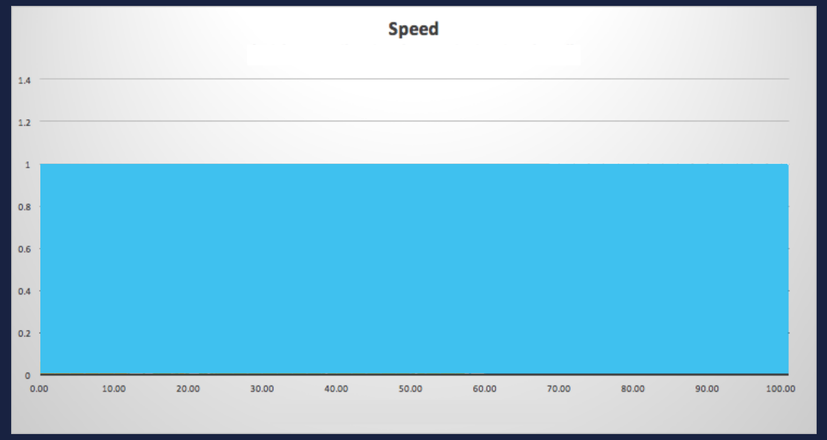
JVM STARTUP

SECOND RUN



NO STARTUP OVERHEAD

THIRD RUN



NO STARTUP OVERHEAD

azul

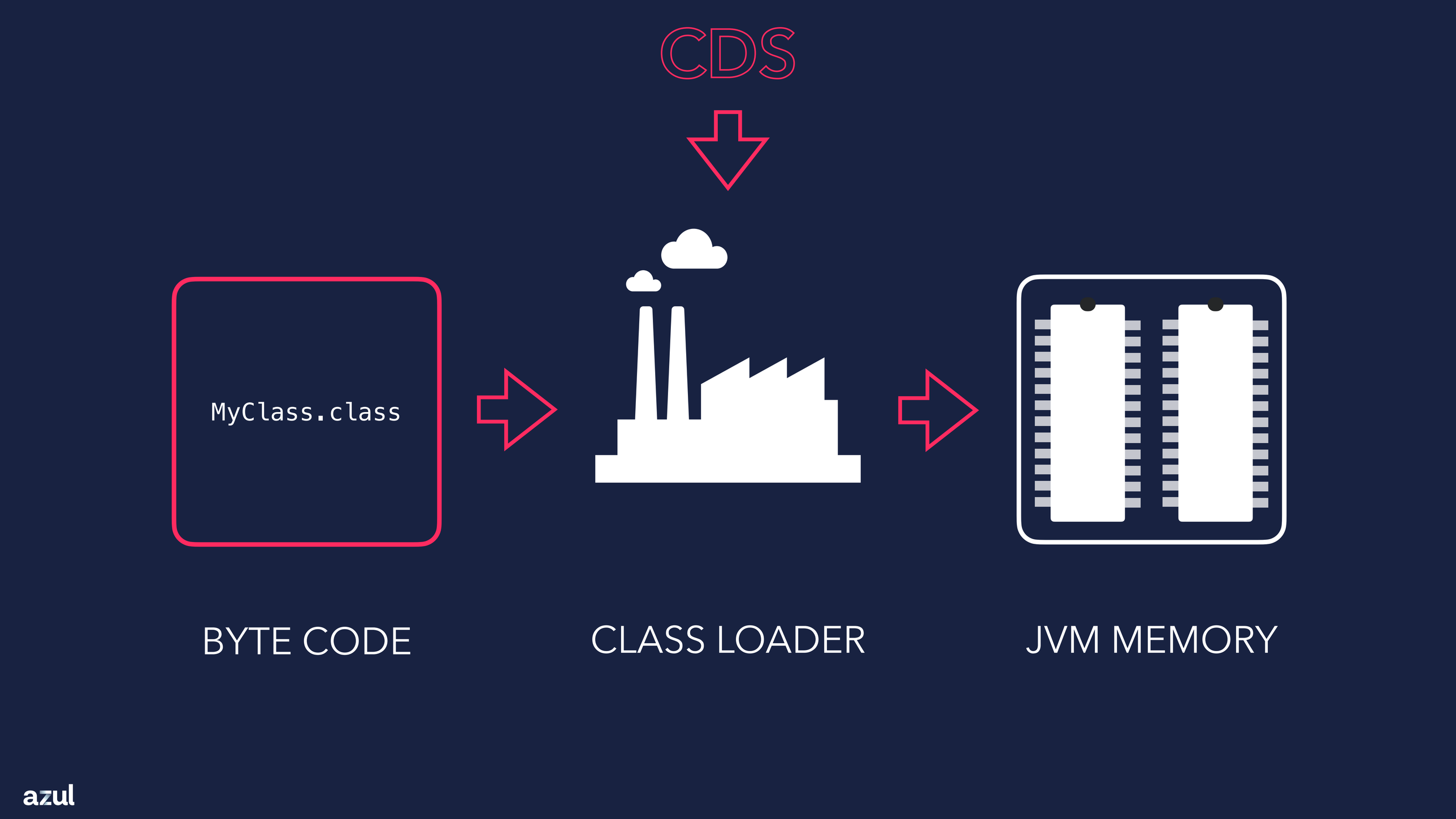
SOLUTIONS...?

CLASS DATA SHARING

azul

WHAT ABOUT CDS?

- Class Data Sharing (App/Dynamic)
- Dump internal class representation into file
- Shared on each JVM start (CDS)
- No optimization or hotspot detection
- Only reduces loading time of classes
- Start up could be up to 1-2 seconds faster


CDS
MyClass.class
BYTE CODE
CLASS LOADER
JVM MEMORY

AHEAD OF TIME COMPILATION

azul

WHY NOT USE AOT?

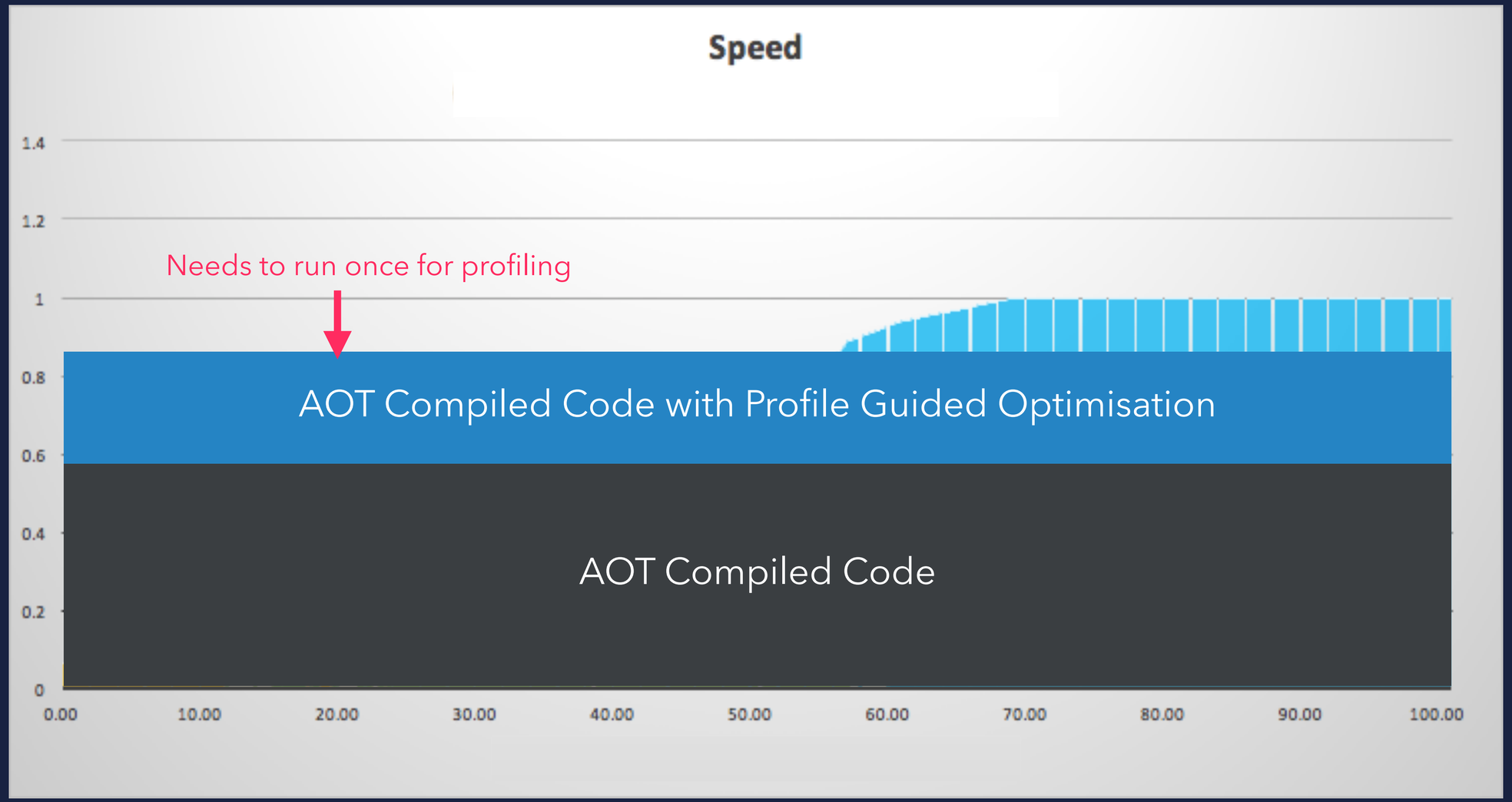
- No interpreting bytecodes
- No analysis of hotspots
- No runtime compilation of code
- Start at full speed, straight away
- GraalVM native image does that

PROBLEM SOLVED...?

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

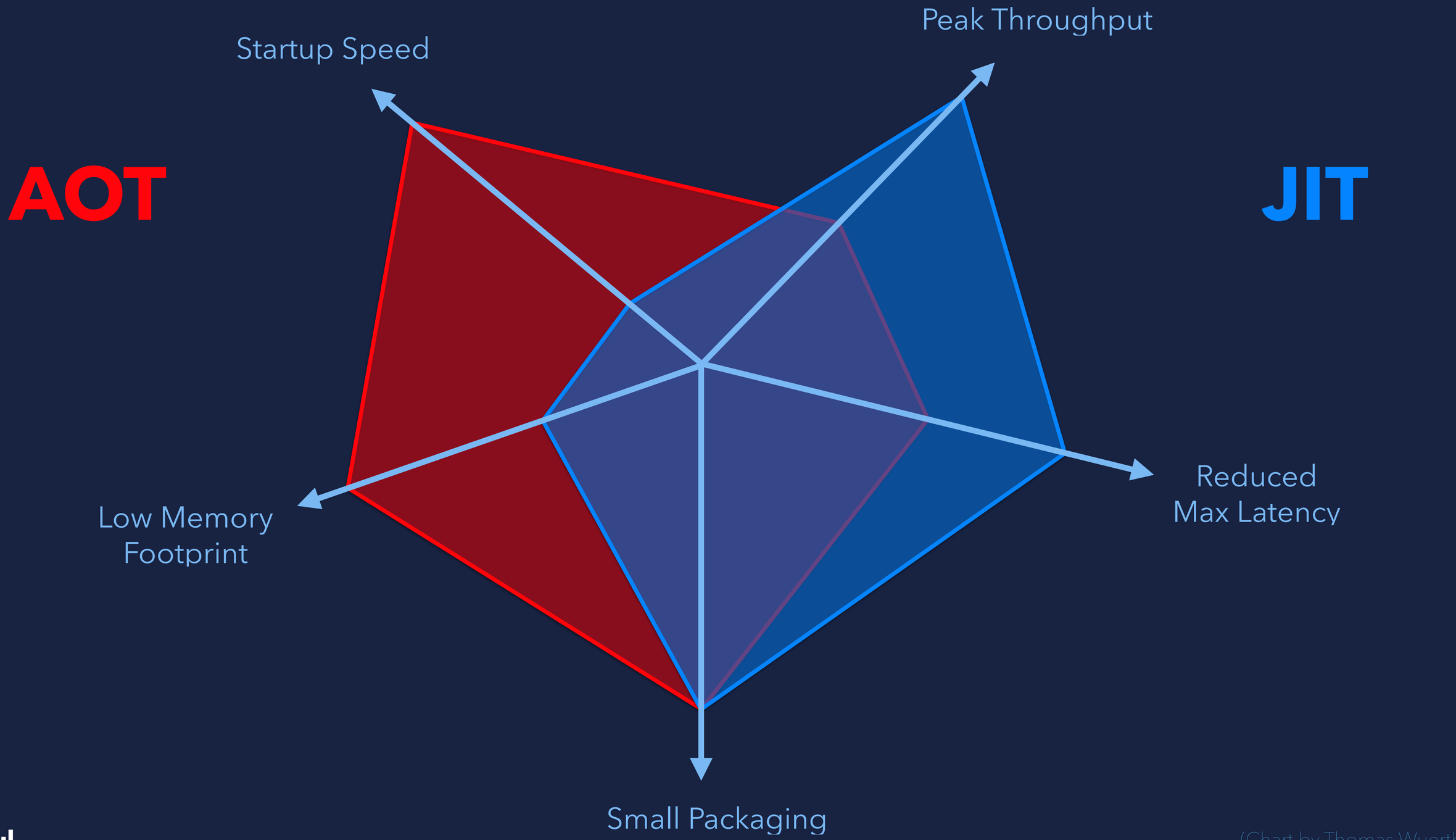
JIT

- 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

JIT DISADVANTAGES

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



azul

(Chart by Thomas Wuerthinger)

A DIFFERENT APPROACH

azul

CRIU

CHECKPOINT RESTORE IN USERSPACE

azul

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

- Mainly implemented in userspace (not kernel)
- Main goal is migration of containers for microservices
- 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 without need to exchange packets

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

azul

CRaC

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

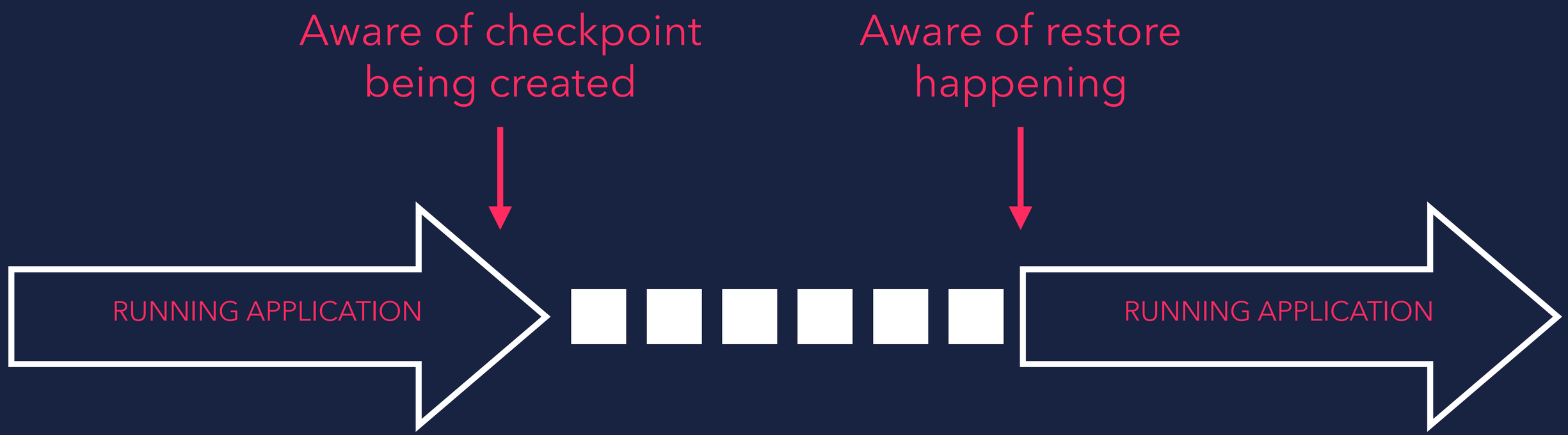
openjdk.org/projects/crac

Lead by Anton Kozlov (Azul)

azul

CRaC

CRaC has more restrictions (e.g. no open files, sockets etc.)



CRaC

START

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

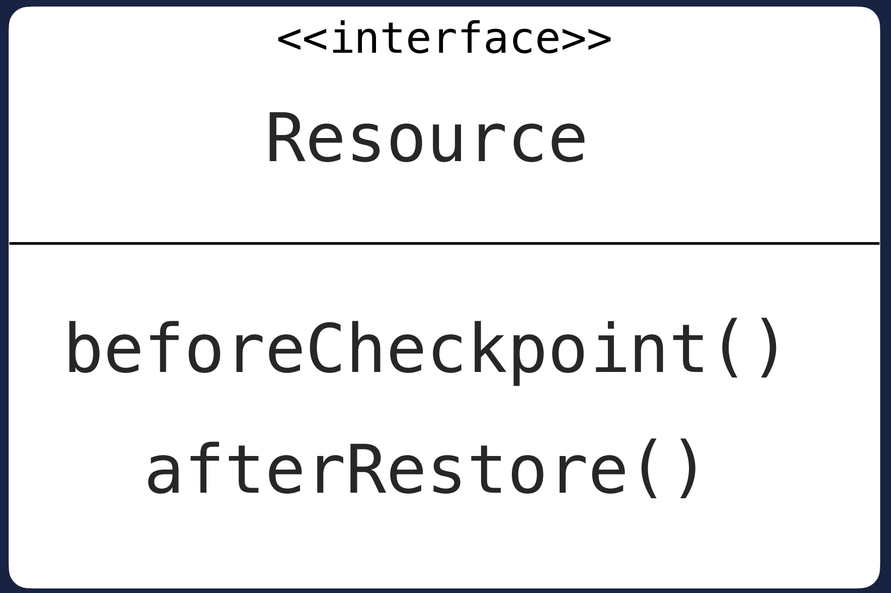
RESTORE

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

CRaC API

CRaC API

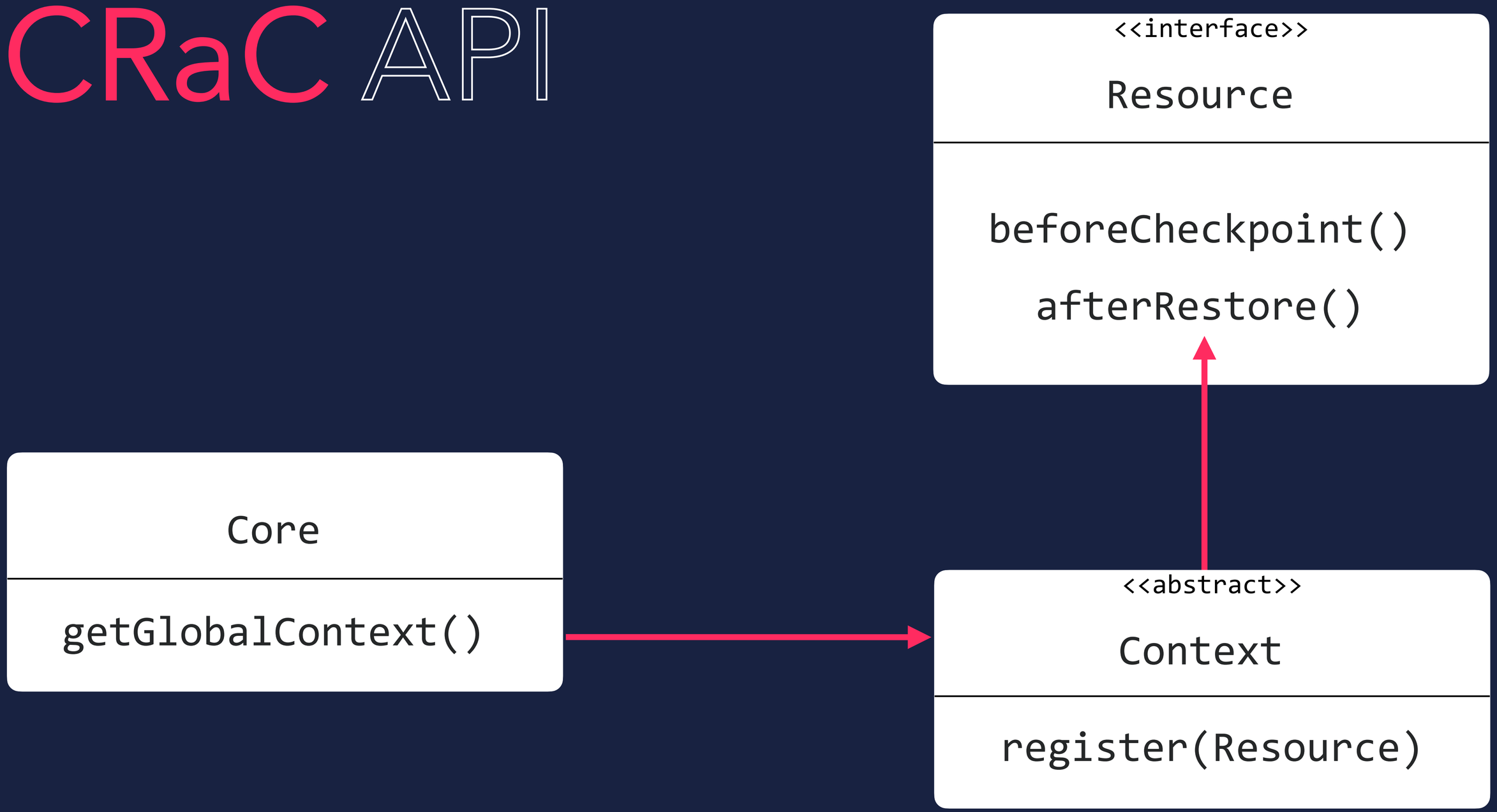
- CRaC uses Resources that can be notified about a Checkpoint and Restore
- Classes in application code implement the Resource interface
- The application receives callbacks during checkpointing and restoring
- Makes it possible to close/restore resources (e.g. open files, sockets)



CRaC API

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

CRaC API


<<interface>>
Resource
beforeCheckpoint()
afterRestore()
Core
getGlobalContext()
<<abstract>>
Context
register(Resource)

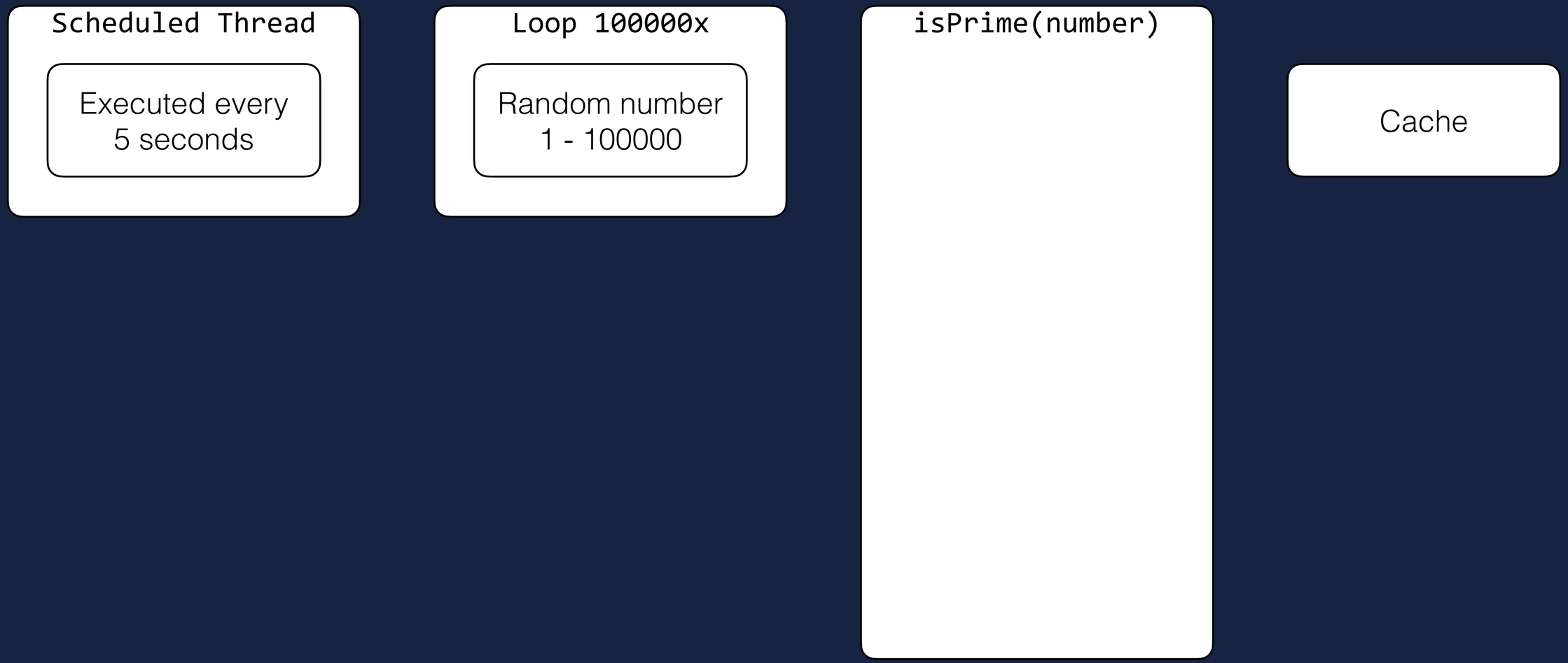
CRaC API

- The global Context maintains a list of Resource objects
- The beforeCheckpoint() methods are called in the reverse order the Resource objects were added
- The afterRestore() methods are called in the order the Resource objects were added
- Checkpoint and restore can be done programmatically by using Core.checkpointRestore()
- Checkpoint can also be created by jcmd

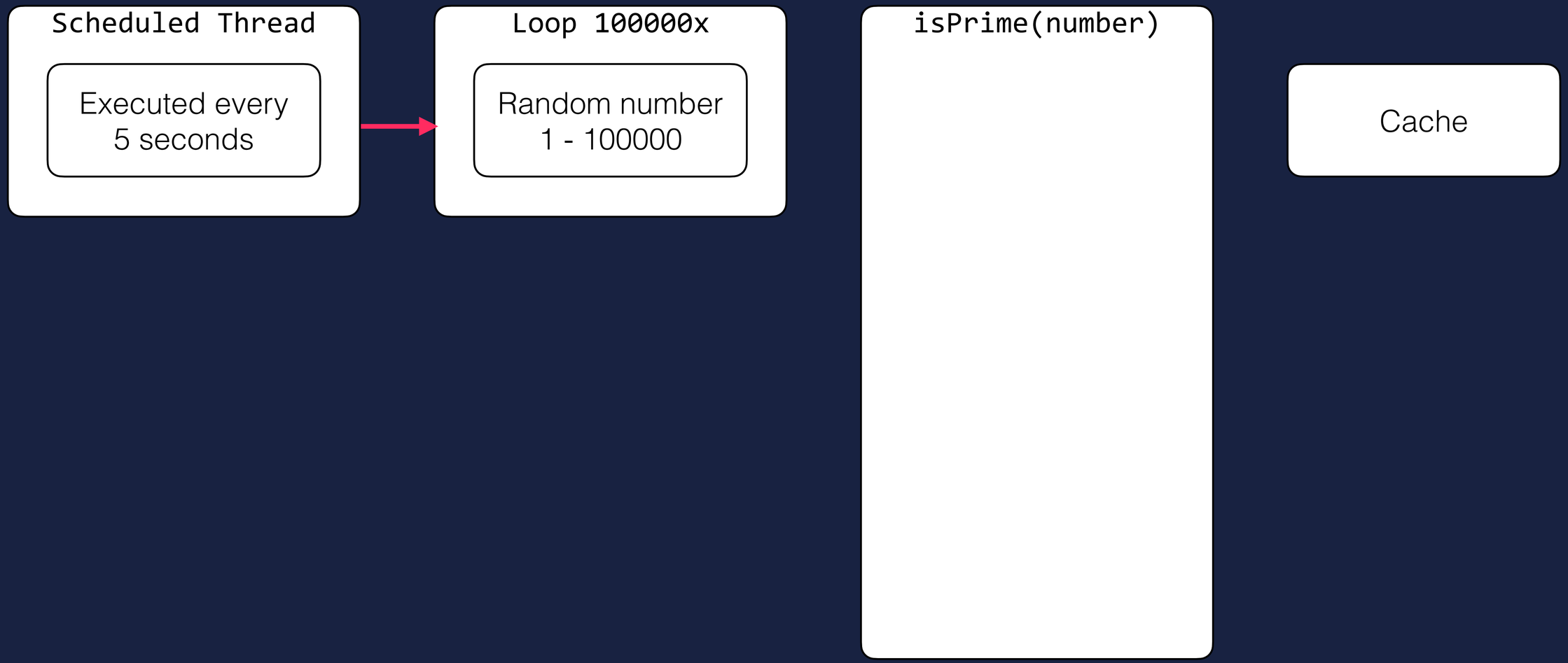
SIMPLE

EXAMPLE

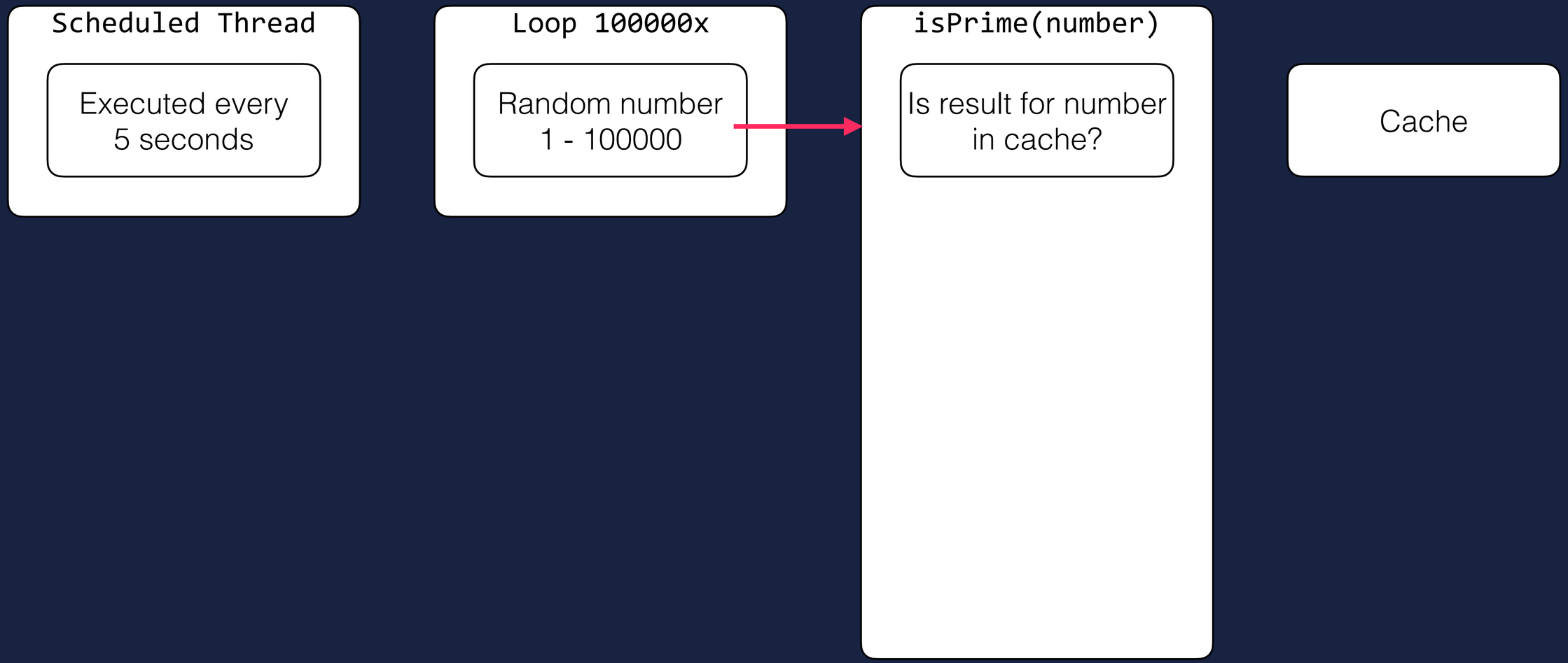
SIMPLE EXAMPLE



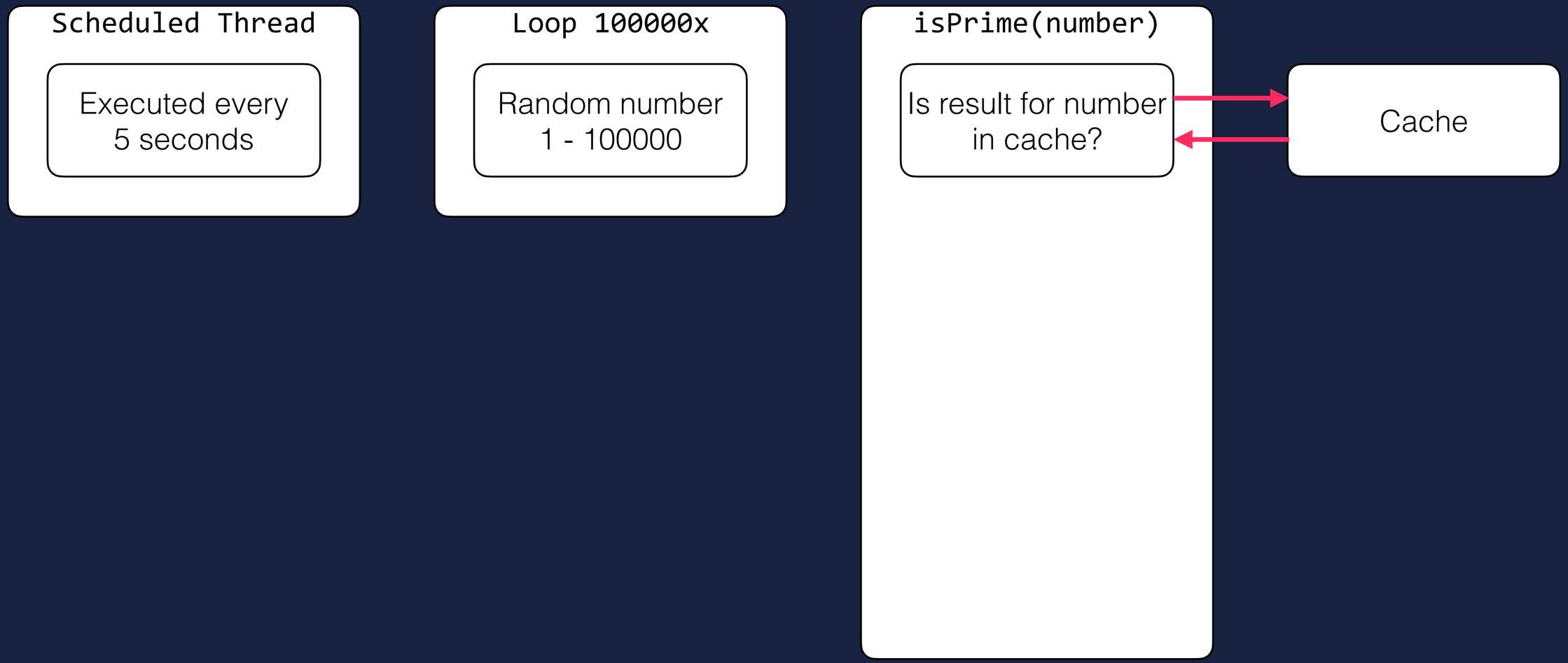
SIMPLE EXAMPLE



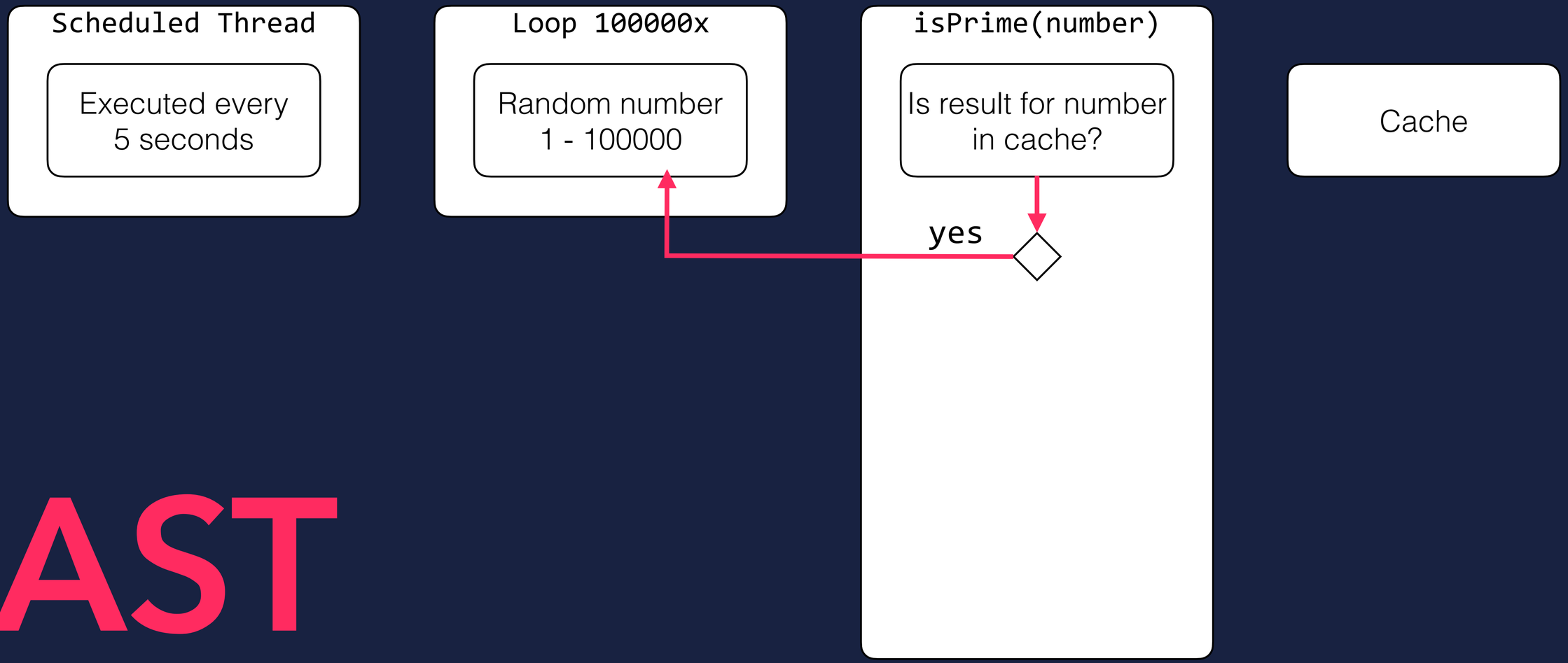
SIMPLE EXAMPLE



SIMPLE EXAMPLE

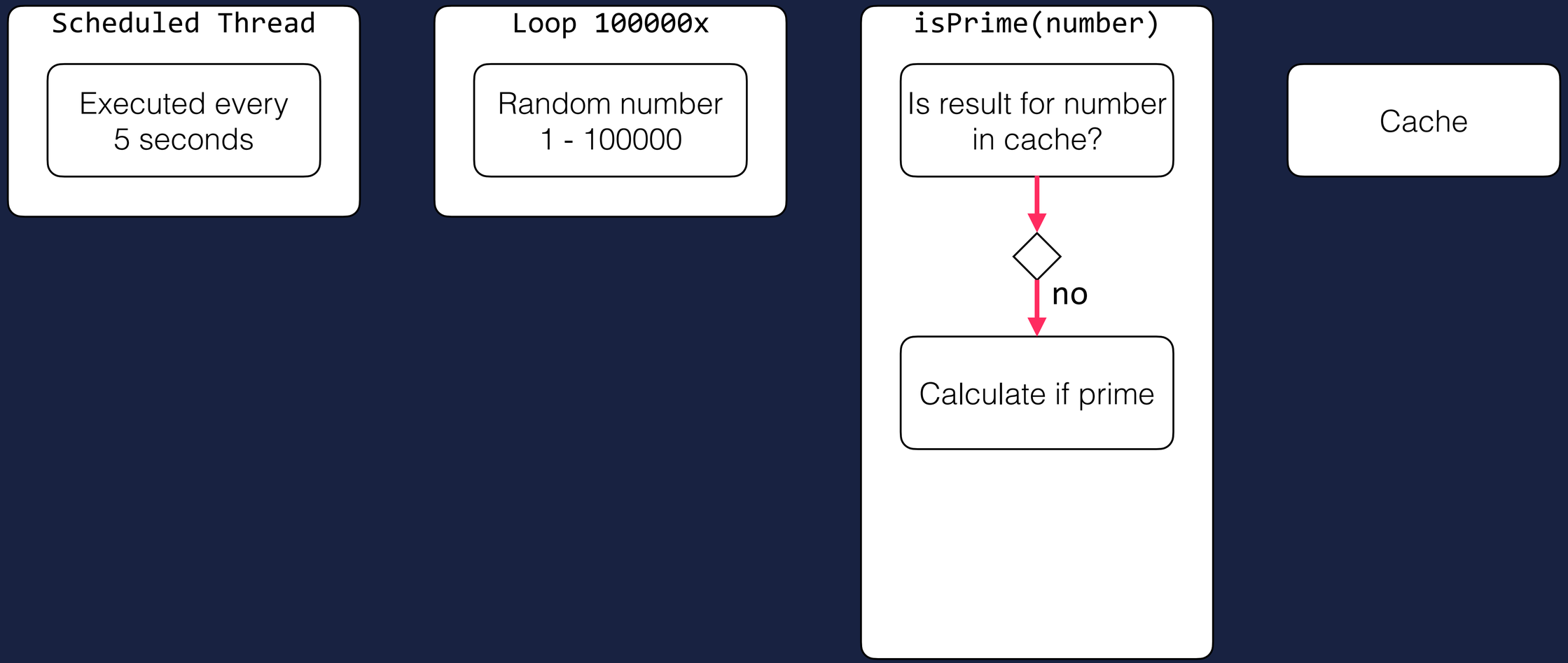


SIMPLE EXAMPLE

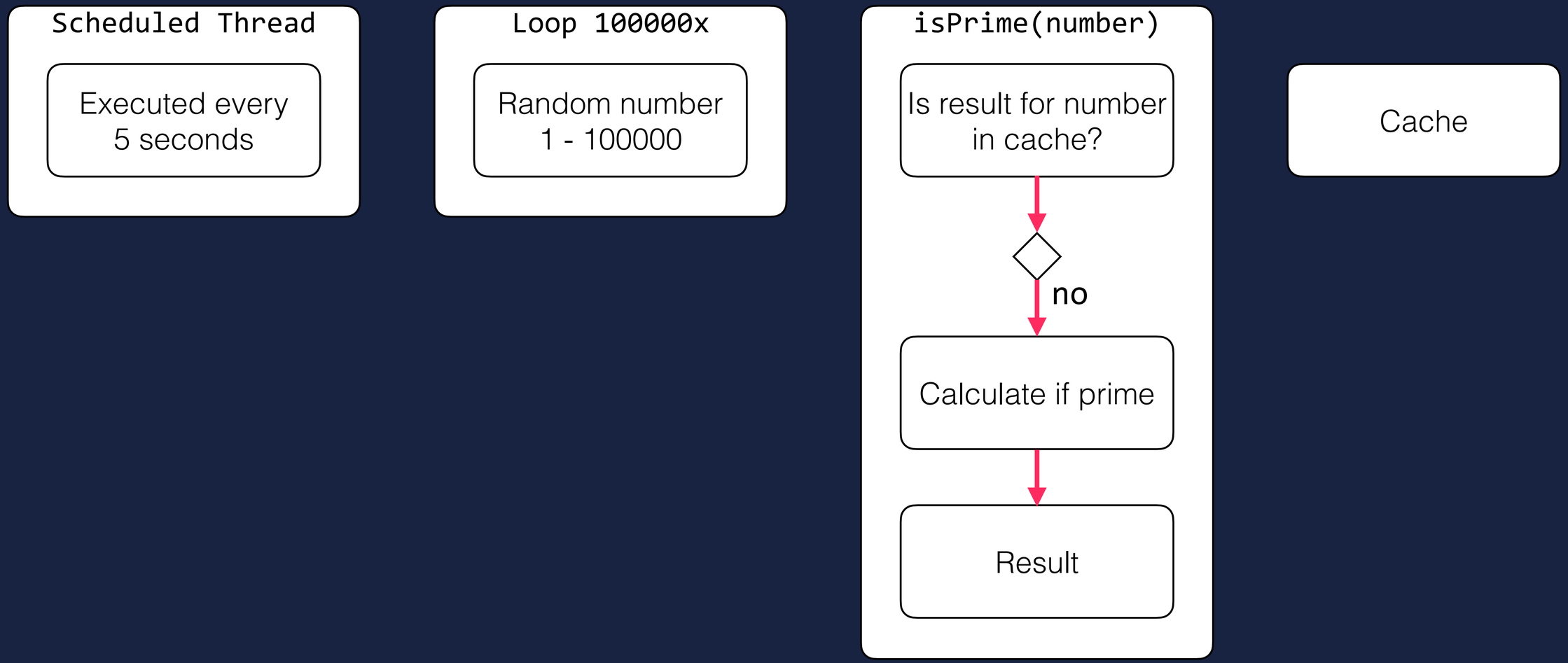


FAST

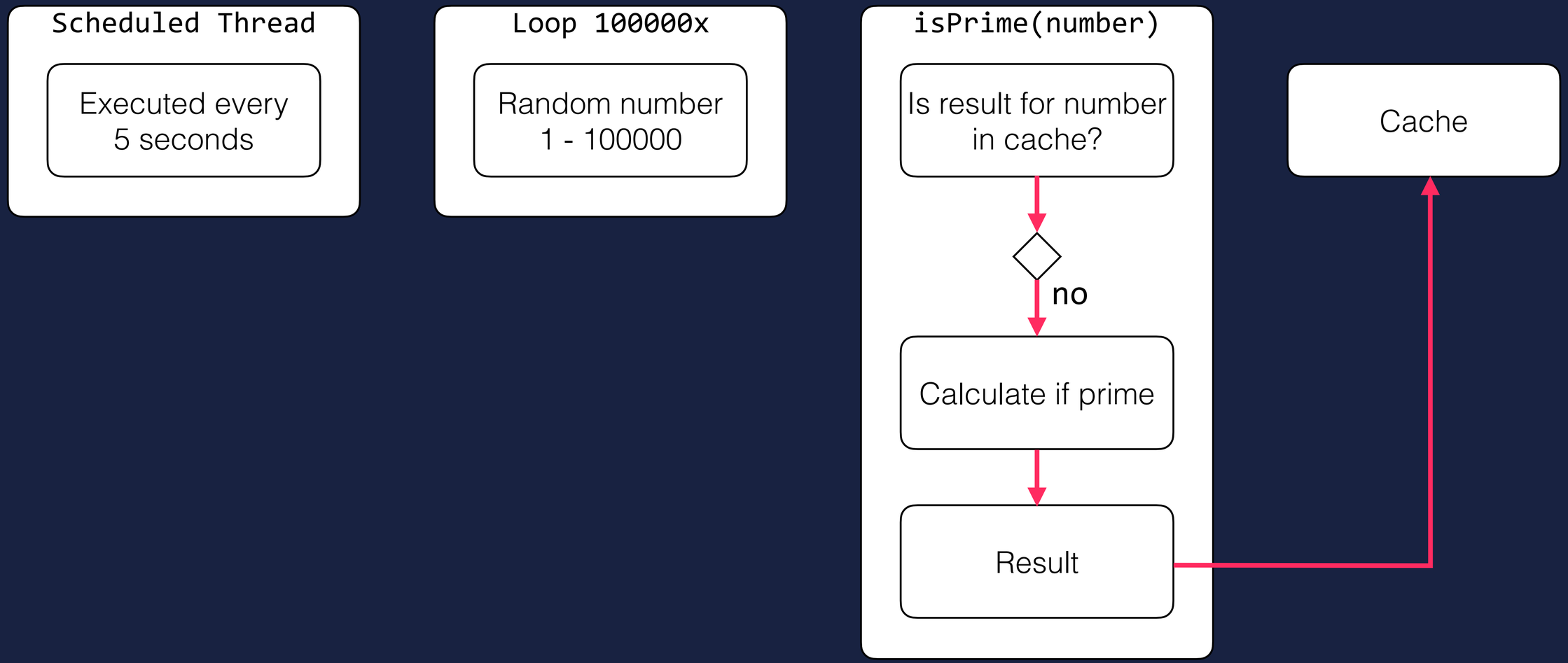
SIMPLE EXAMPLE



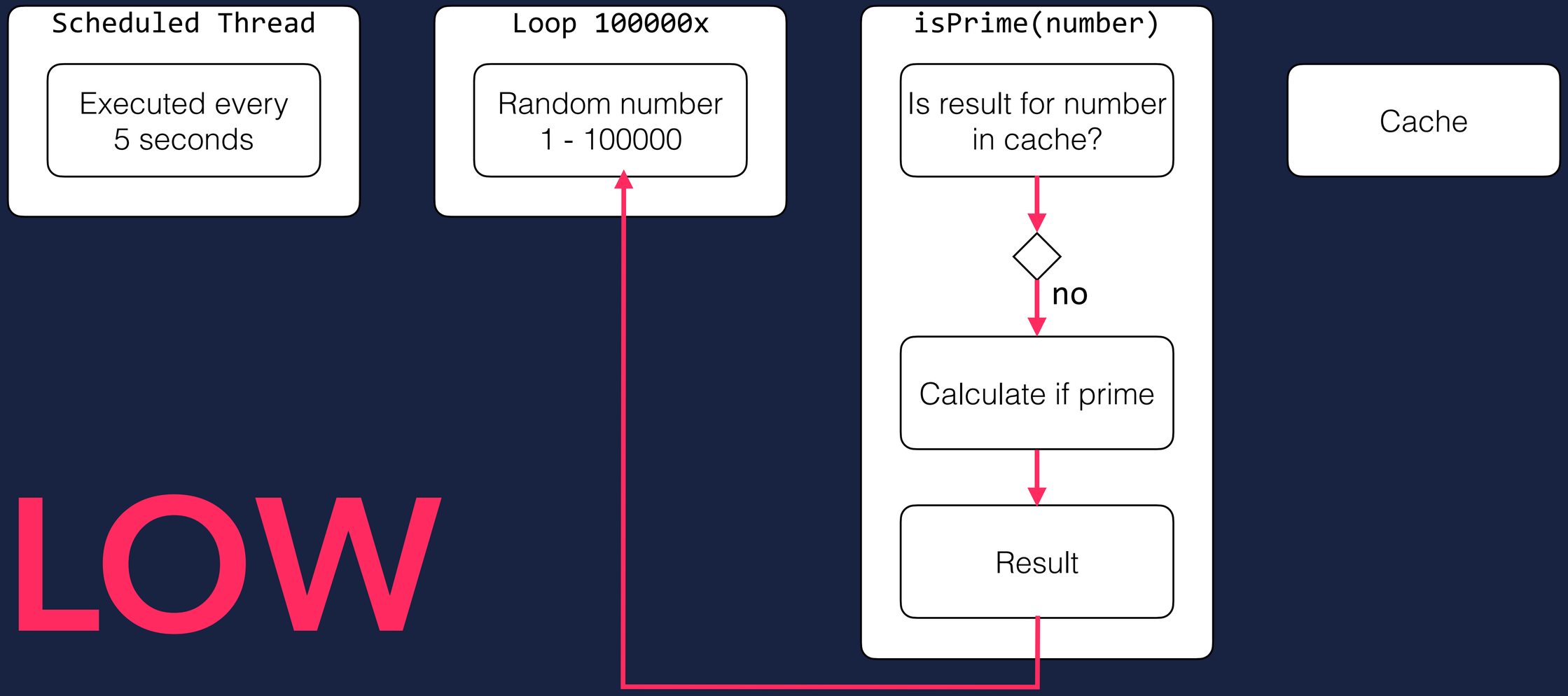
SIMPLE EXAMPLE



SIMPLE EXAMPLE



SIMPLE EXAMPLE



SLOW

SIMPLE EXAMPLE

`Scheduled Thread`

Executed every
5 seconds

`Loop 100000x`

Random number
1 - 100000

`isPrime(number)`

Is result for number
in cache?

no

Calculate if prime

Result

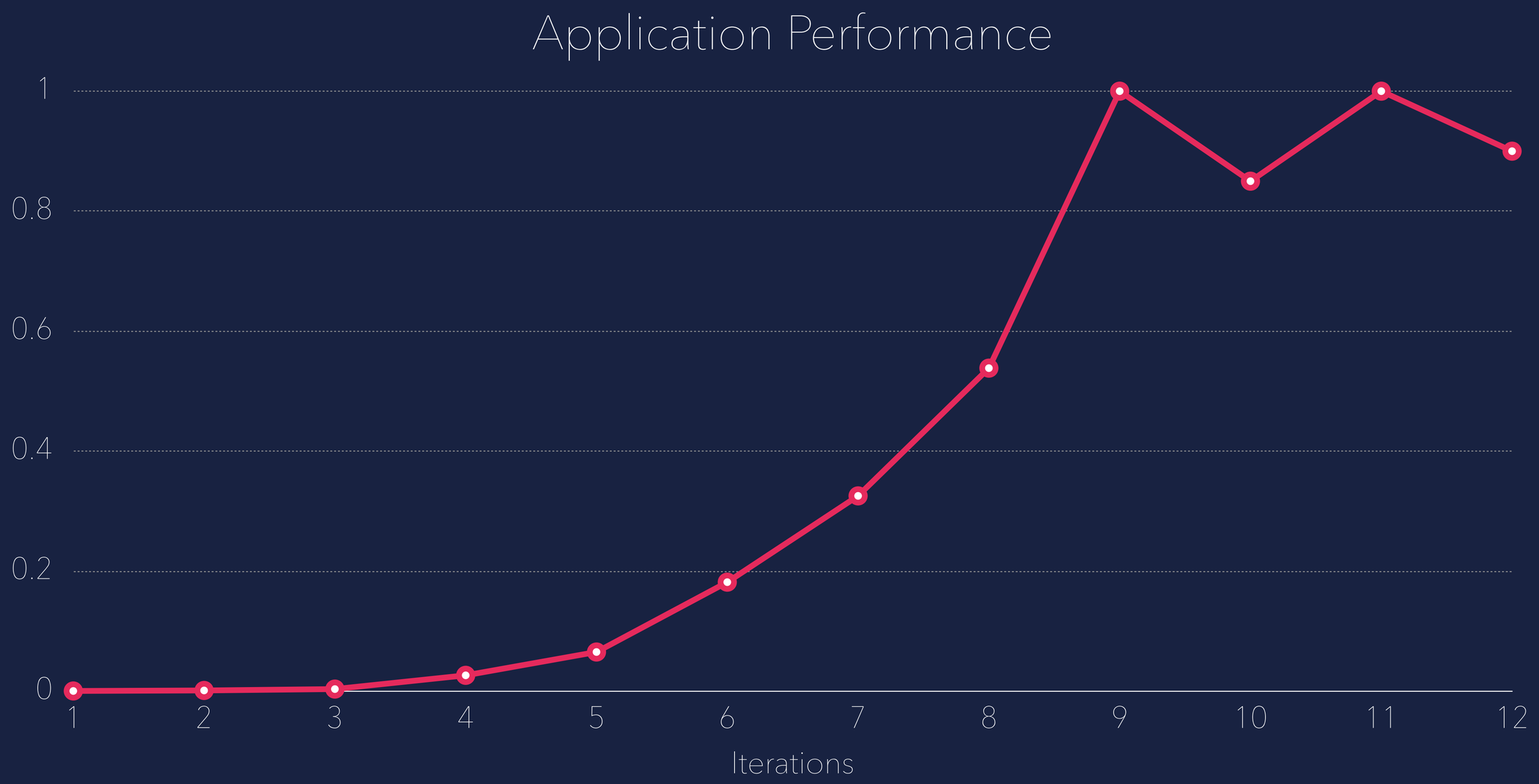
Cache

```
inital delay 50 sec
timeout 10 sec
```

SIMPLE EXAMPLE

- First run is slow because the cache is empty
- Time to check 100 000 random numbers for prime should decrease with every run

SIMPLE EXAMPLE



azul

DEMO...

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
```

Defines where to store files
for the checkpoint

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
```

Slow because cache is not filled yet

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
11. Run: 17 ms (99999 elements cached, 100.0%)
```

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
11. Run: 17 ms (99999 elements cached, 100.0%)
```

Fast because cache is filled

SHELL 2

```
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
11. Run: 17 ms (99999 elements cached, 100.0%)
```

SHELL 2

```
>jcmd crac4.jar JDK.checkpoint
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
11. Run: 17 ms (99999 elements cached, 100.0%)
```

SHELL 2

```
>jcmd crac4.jar JDK.checkpoint
```

CREATE CHECKPOINT

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
11. Run: 17 ms (99999 elements cached, 100.0%)
beforeCheckpoint() called in Main
beforeCheckpoint() called in GenericCache
CR: Checkpoint ...
Killed
>
```

SHELL 2

```
>jcmd crac4.jar JDK.checkpoint
4240:
Command executed successfully
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
11. Run: 17 ms (99999 elements cached, 100.0%)
beforeCheckpoint() called in Main
beforeCheckpoint() called in GenericCache
CR: Checkpoint ...
Killed
>
```

SHELL 2

```
>jcmd crac4.jar JDK.checkpoint
4240:
Command executed successfully
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
11. Run: 17 ms (99999 elements cached, 100.0%)
beforeCheckpoint() called in Main
beforeCheckpoint() called in GenericCache
CR: Checkpoint ...
Killed
>java -XX:CRaCRestoreFrom=/crac-files
```

SHELL 2

```
>jcmd crac4.jar JDK.checkpoint
4240:
Command executed successfully
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
11. Run: 17 ms (99999 elements cached, 100.0%)
beforeCheckpoint() called in Main
beforeCheckpoint() called in GenericCache
CR: Checkpoint ...
Killed
>java -XX:CRaCRestoreFrom=/crac-files
```

RESTORE

SHELL 2

```
>jcmd crac4.jar JDK.checkpoint
4240:
Command executed successfully
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
11. Run: 17 ms (99999 elements cached, 100.0%)
beforeCheckpoint() called in Main
beforeCheckpoint() called in GenericCache
CR: Checkpoint ...
Killed
>java -XX:CRaCRestoreFrom=/crac-files
afterRestore() called in GenericCache
afterRestore() called in Main
12. Run: 23 ms (99999 elements cached, 100.0%)
```

SHELL 2

```
>jcmd crac4.jar JDK.checkpoint
4240:
Command executed successfully
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
11. Run: 17 ms (99999 elements cached, 100.0%)
beforeCheckpoint() called in Main
beforeCheckpoint() called in GenericCache
CR: Checkpoint ...
Killed
>java -XX:CRaCRestoreFrom=/crac-files
afterRestore() called in GenericCache
afterRestore() called in Main
12. Run: 23 ms (99999 elements cached, 100.0%)
```

Continues to run...fast

SHELL 2

```
>jcmd crac4.jar JDK.checkpoint
4240:
Command executed successfully
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
11. Run: 17 ms (99999 elements cached, 100.0%)
beforeCheckpoint() called in Main
beforeCheckpoint() called in GenericCache
CR: Checkpoint ...
Killed
>java -XX:CRaCRestoreFrom=/crac-files
afterRestore() called in GenericCache
afterRestore() called in Main
12. Run: 23 ms (99999 elements cached, 100.0%)
13. Run: 21 ms (99995 elements cached, 100.0%)
```

SHELL 2

```
>jcmd crac4.jar JDK.checkpoint
4240:
Command executed successfully
>
```

SIMPLE EXAMPLE

SHELL 1

```
>java -XX:CRaCCheckpointTo=/crac-files -jar crac4.jar
Running on CRaC (PID 4240)
First run will take up to 30 seconds...
Register Resource: GenericCache
Register Resource: Main
1. Run: 27863 ms (63303 elements cached, 63.3%)
2. Run: 10611 ms (86580 elements cached, 86.6%)
3. Run: 3965 ms (95161 elements cached, 95.2%)
4. Run: 1428 ms (98245 elements cached, 98.2%)
5. Run: 533 ms (99340 elements cached, 99.3%)
6. Run: 207 ms (99750 elements cached, 99.8%)
7. Run: 87 ms (99898 elements cached, 99.9%)
8. Run: 42 ms (99960 elements cached, 100.0%)
9. Run: 25 ms (99986 elements cached, 100.0%)
10. Run: 20 ms (99992 elements cached, 100.0%)
11. Run: 17 ms (99999 elements cached, 100.0%)
beforeCheckpoint() called in Main
beforeCheckpoint() called in GenericCache
CR: Checkpoint ...
Killed
>java -XX:CRaCRestoreFrom=/crac-files
afterRestore() called in GenericCache
afterRestore() called in Main
12. Run: 23 ms (99999 elements cached, 100.0%)
13. Run: 21 ms (99995 elements cached, 100.0%)
14. Run: 24 ms (99997 elements cached, 100.0%)
```

SHELL 2

```
>jcmd crac4.jar JDK.checkpoint
4240:
Command executed successfully
>
```

CREATING A CHECKPOINT

FROM THE COMMAND LINE:

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

CREATING A CHECKPOINT

FROM CODE:

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

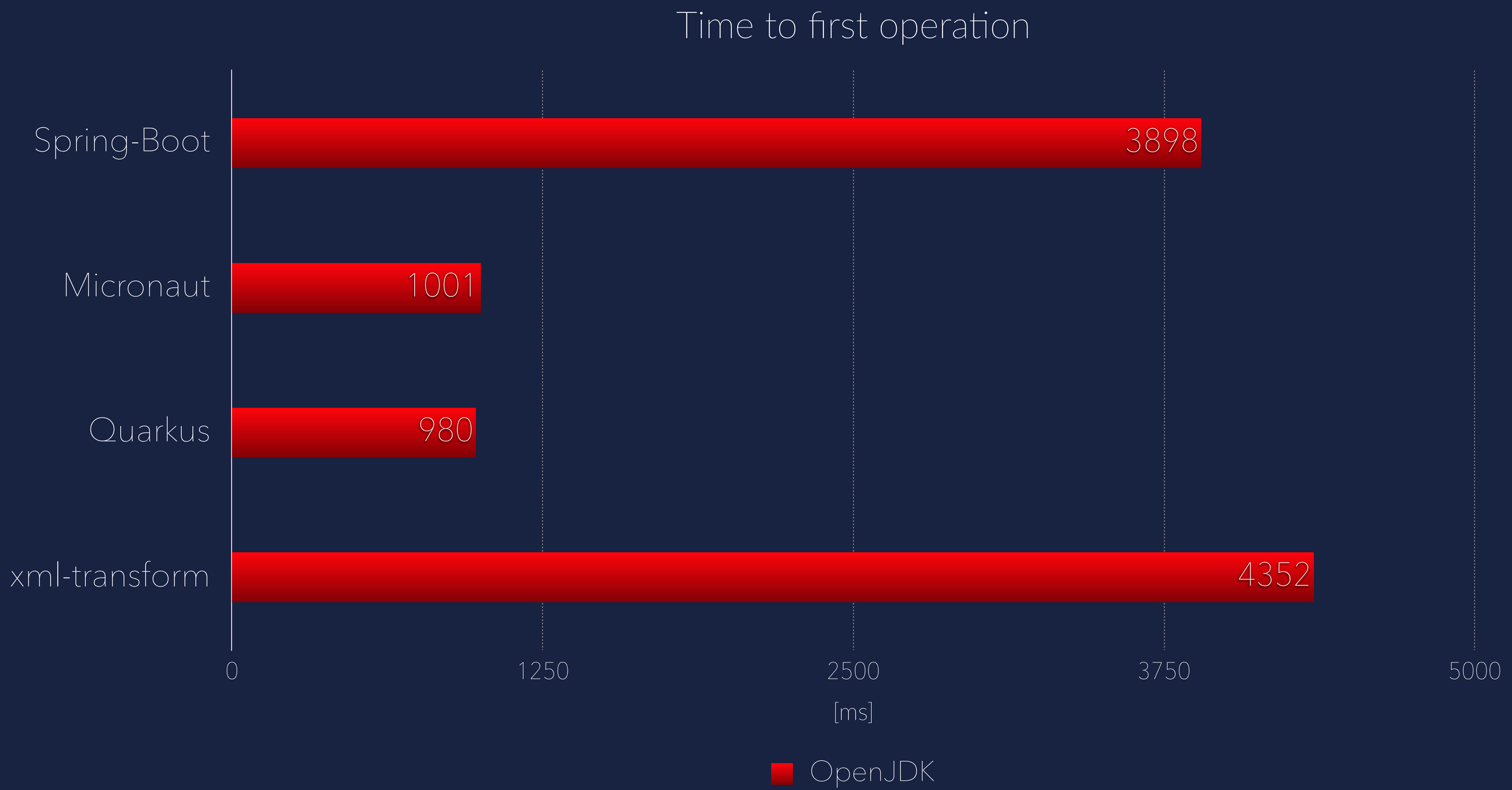
WHEN TO CHECKPOINT ?

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

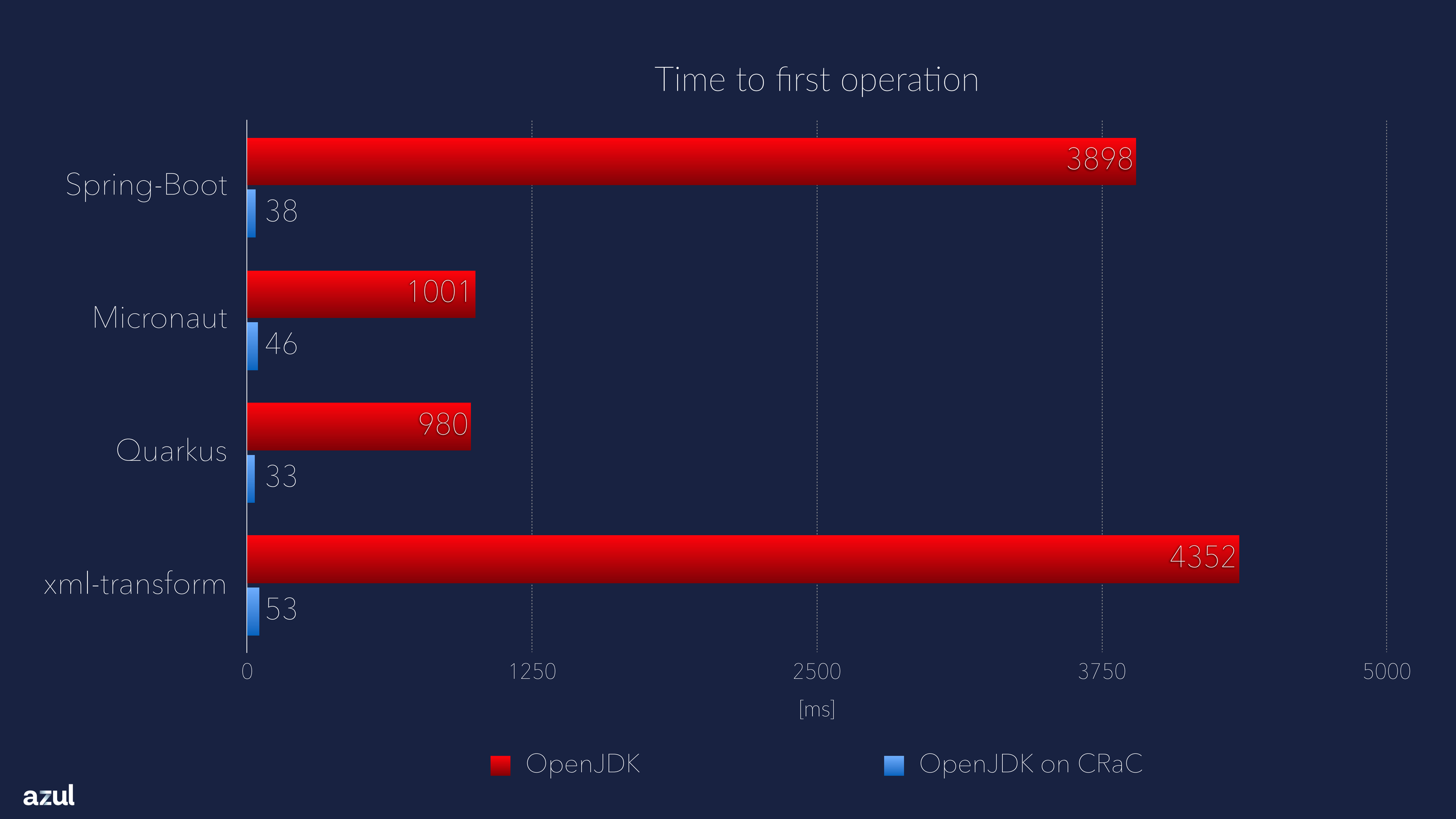
azul

OK...BUT

HOW GOOD IS IT...?


Time to first operation
Spring-Boot
3898
Micronaut
1001
Quarkus
980
xml-transform
4352
0
1250
2500
3750
5000
[ms]
OpenJDK

azul


Time to first operation
Spring-Boot
3898
38
Micronaut
1001
46
Quarkus
980
33
xml-transform
4352
53
0
1250
2500
3750
5000
[ms]
OpenJDK
OpenJDK on CRaC

azul

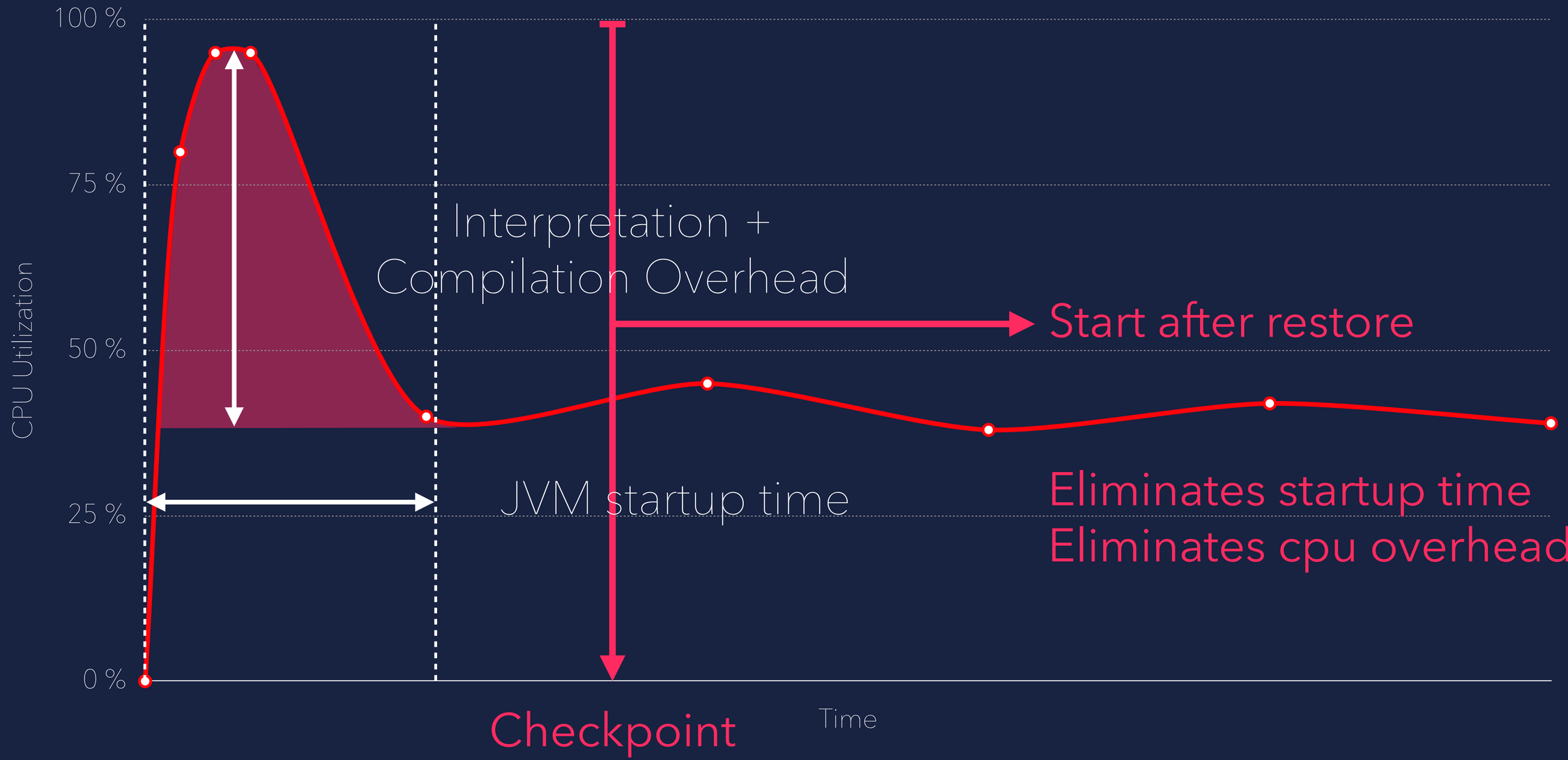
SUMMARY...

azul

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



azul

github.com/CRaC

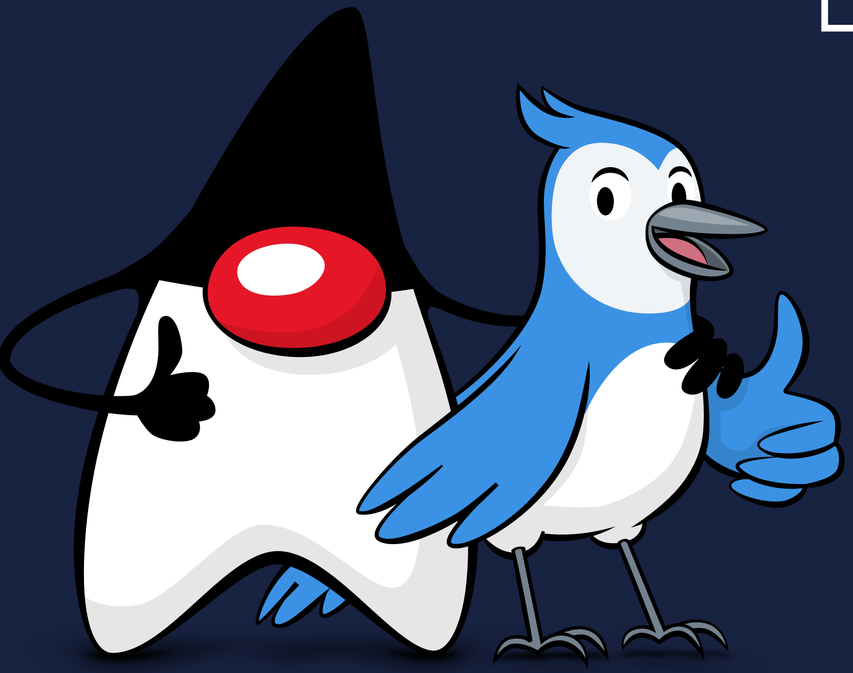
azul

wiki.openjdk.org/display/crac

azul

DEMO...

THANK YOU



azul