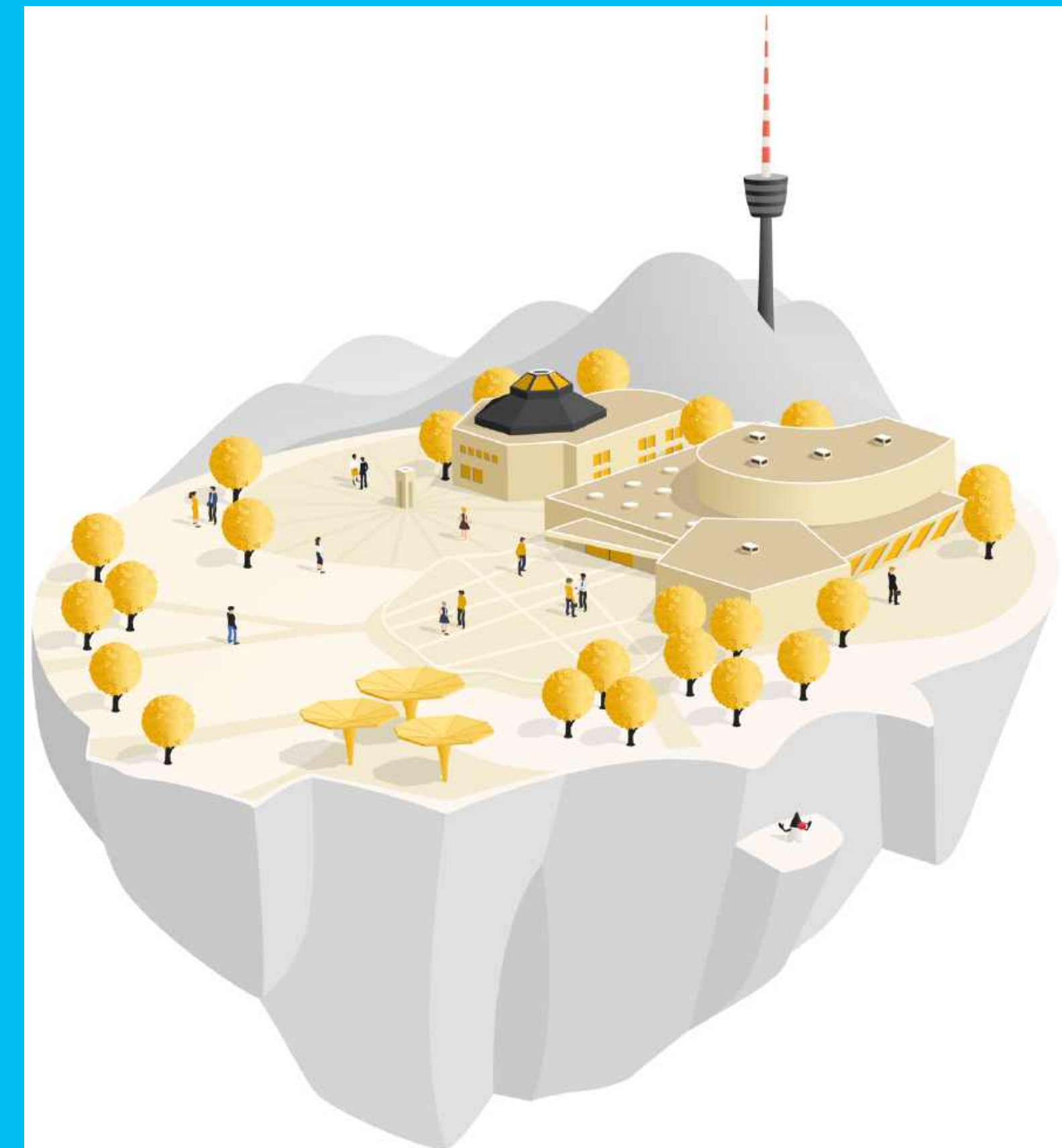


KARSTEN SILZ, 10. JULI 2025

WIE KANN ICH JAVA SCHNELLER STARTEN – UND WANN LOHNT SICH DAS?

JAVAFORUM *stuttgart* 2025



WANN KANN ES
SICH LOHNEN, JAVA
SCHNELLER ZU STARTEN?

SERVERLESS,
VIELE SERVER,
SEHR HOHE VERFÜGBARKEIT

JAVA BY DESIGN:

MEHR ZEIT,

MEHR CPU,

MEHR SPEICHER

1. SPRING BOOT TUNING

**2. CLASS DATA SHARING (CDS) &
PROJECT LEYDEN**

3. PROJECT CRAC

4. GRAALVM NATIVE IMAGE

FOLIEN, CODE & MEHR



[HTTPS://BPF.LI/JF5](https://bpf.li/jf5)

 Architektur & Sicherheit REST APIs from Hell Wahre Geschichten über Misserfolge Miriam Greis A2	 Trends & neue Technologien (KI o.ä.) KI-gestützte Java Migrationen Michael Pisula, Marijn van Geest G2	 Methodik & Praxis Qualität trotz immer kürzerer Releasezyklen – mit Continuous Quality Control Tobias Röhm E2	 Architektur & Sicherheit Domain-Driven Transformation Modularisieren und Modernisieren von Legacy-Software Henning Schwentner B2	 Microservices, Container & Cloud OpenAPI Codegenerierung: The Good, The Bad & The Ugly Ingo Düppe F2	 Core Java & JVM-Sprachen Multiplattform mit Kotlin Wie Kotlin mit Multiplattform das Korsett der JVM sprengt Christian Babsek D2
---	---	--	--	---	--

**WAS MACHT MICH ZUM
EXPERTEN?**

**2022: BETREUTE ARTIKEL-SERIE
ÜBER GRAALVM**

**SEITDEM: ARTIKEL & VORTRÄGE ZU
GRAALVM, LEYDEN & CRAC VERFASST**

LOHNT SICH DAS?

WARUM JETZT?

WIESO LANGSAM?

WIE SCHNELLER?

LOHNT SICH DAS?

ENTWICKLER SOLLLEN

BUSINESS VALUE

SCHAFFEN...

NEUE FEATURES

VERBESSERUNGEN

FEHLERBEHEBUNGEN

...**NICHT** KOSTENEIN-
SPARUNGEN!

FRAGEN SIE **CHEF** FÜR
JAVA-STARTUP-
OPTIMIERUNG
UM **ERLAUBNIS!**

SERVERLESS,
VIELE SERVER
SEHR HOHE VERFÜGBARKEIT

SERVERLESS

**SCHNELLERER STARTUP =
GERINGERE KOSTEN**

AWS LAMBDA **SNAPSTART:** LAMBDA MIT **CRAC**

VIELE SERVER

VIELE **STANDBY-
INSTANZEN
FÜR LASTSPITZEN:**

SCHNELLERER STARTUP =

WENIGER INSTANZEN =

WENIGER **SERVIER**

**GILT IMMER FÜR
VIELE SERVER:**

WENIGER **SPEICHER** PRO

ANWENDUNG =

MEHR INSTANZEN PRO SERVER =

WENIGER **SERVER**

NUR, WENN
ANWENDUNGEN NICHT
CPU-LIMITIERT!

**SEHR HOHE
VERFÜGBARKEIT**

DOWNTIME PRO JAHR &
VERFÜGBARKEITS-LEVEL

99.9%: KNAPP 9H

99.99%: GUT 52 MIN

99.999%: 5 MIN

WAS SIE SOWIESO MACHEN:
MEHRERE INSTANZEN,
ROLLING UPGRADES

**SCHNELLER STARTUP:
VERSICHERUNG FÜR
GEPLANTE ODER
UNGEPLANTE AUSFÄLLE**

BEWERTUNG DER VARIANTEN

KOSTEN

**EINMAL-KOSTEN, Z.B. UMSTELLUNG
BUILD & DEPLOYMENT ODER
ENTWICKLER-TRAINING**

**WIEDERKEHRENDE KOSTEN, Z.B.
WARTUNG ODER UPGRADES**

DEV EXPERIENCE

VERSCHLECHTERUNG DER
LOKALEN ENTWICKLUNGS-
ARBEIT ODER TROUBLESHOOTING
VON PRODUKTION

	TUNING	CDS & LEYDEN	CRAC	GRAALVM
ZEIT: SERVERLESS, STANDBY, VERFÜG- BARKEIT				
SPEICHER: WENIGER SERVER KOSTEN				
DEV EXPERIENCE				

**DAS LOHNT SICH
MANCHMAL!**

LOHNT SICH DAS?

WARUM JETZT?

WIESO LANGSAM?

WIE SCHNELLER?

LOHNT SICH DAS?

WARUM JETZT?

WIESO LANGSAM?

WIE SCHNELLER?

WARUM JETZT?

**JAVA TEURER ALS
NEUE KONKURRENZ**

**ANWENDUNGEN LAUFEN IMMER
MEHR IM CONTAINER...**

**... UND DEM CONTAINER IST DIE
PROGRAMMIERSPRACHE EGAL:
JAVASCRIPT, C#, PYTHON, GO, RUST, ...**

DOCKRHUB: ÖFFENTLICHES
CONTAINER-IMAGE-REPOSITORY

ERMÖGLICHT **KOSTENGÜNSTIGE**
INFRASTRUKTUR: DATENBANKEN,
WEB-SERVER, PROXIES, CACHES, ...

STANDARDISIERTE KOMMUNIKATION VON ANWENDUNGEN:

REST/GRAPHQL + JSON

**CONTAINER + DOCKERHUB
+ REST/GRAPHQL + JSON =
NEUE KONKURRENZ**

**JAVA TEURER ALS
NEUE KONKURRENZ**

**CONTAINER LAUFEN
AUF SERVERN**

**CLOUD-RECHNUNG MACHT
SERVER-KOSTEN SICHTBAR**

JAVA BRAUCHT MEHR
ZEIT, CPU & RAM ALS
ANDERE SPRACHEN

JAVA **TEURER** IN CONTAINERN

**JAVA TEURER ALS
NEUE KONKURRENZ**

DARUM JETZT!

LOHNT SICH DAS?

WARUM JETZT?

WIESO LANGSAM?

WIE SCHNELLER?

WIESO LANGSAM?

JAVA-ANWENDUNGEN LAUFEN ALS **BYTECODE:** .CLASS-DATEIEN IN JARS

BYTECODE LÄUFT AUF LINUX,
WINDOWS, MACOS...

**...ABER WIRD
INTERPRETIERT –
LANGSAM!**

JIT-COMPILER (JUST IN TIME):
BYTECODE ZU MASCHINENCODE

STANDARD JIT-COMPILER:
“HOTSPOT” SEIT 1999

HOTSPOT-TEILE

PROFILER

VERMERKT, WIE **OFT**
METHODEN & SCHLEIFEN
DURCHLAUFEN WERDEN

C1 COMPILER

SCHNELLE COMPILIERUNG => **NICHT-
OPTIMIERTER** MASCHINENCODE

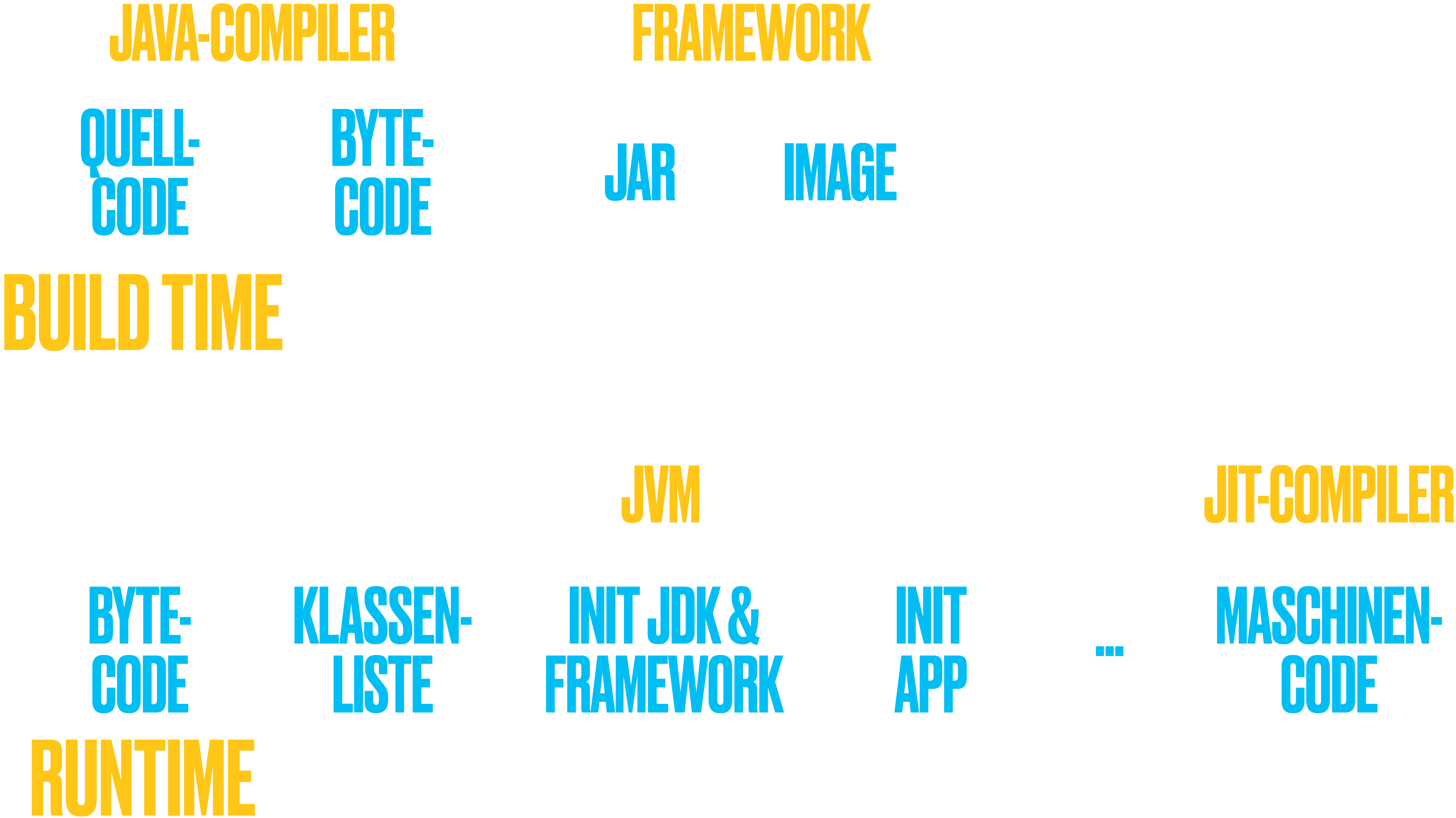
MASCHINENCODE ENTHÄLT
PROFILING-CODE

G2 COMPILER

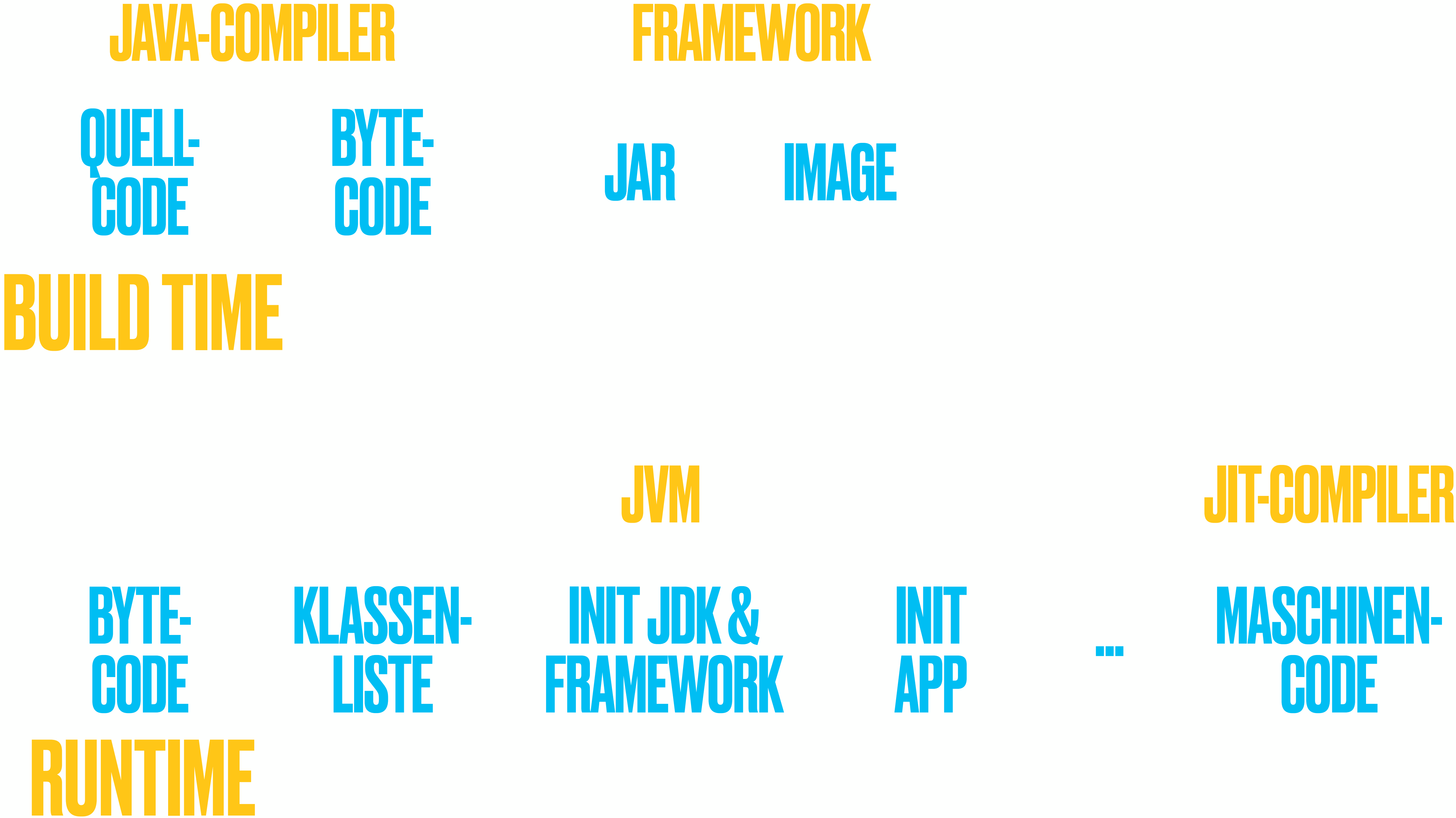
**LANGSAME COMPILIERUNG =>
OPTIMIERTER MASCHINENCODE**

**METHOD INLINING, LOOP UNROLLING,
SCALAR REPLACEMENT, VECTORIZATION
OF ARITHMETIC LOOPS, ...**

WIE STARTET JAVA?



STARTUP TIME &
TIME TO **PEAK**
PERFORMANCE



JAVA-COMPILER

FRAMEWORK

QUELL-
CODE

BYTE-
CODE

JAR

IMAGE

TIME TO **PEAK**
PERFORMANCE

BUILD TIME

JVM

JIT-COMPILER

BYTE-
CODE

KLASSEN-
LISTE

INIT JDK &
FRAMEWORK

INIT
APP

...

MASCHINEN-
CODE

RUNTIME

JAVA BY DESIGN:

MEHR ZEIT,

MEHR CPU,

MEHR SPEICHER

JAVA-COMPILER

QUELL-
CODE

BYTE-
CODE

FRAMEWORK

JAR

IMAGE

LÄUFT JEDES MAL,
GLEICHES RESULTAT:
VIELE JAVA-
OBJEKTE

BUILD TIME

JVM

JIT-COMPILER

BYTE-
CODE

KLASSEN-
LISTE

INIT JDK &
FRAMEWORK

INIT
APP

...

MASCHINEN-
CODE

RUNTIME

MEHR ZEIT FÜR

STARTUP

MEHR CPU

JAVA-COMPILER

**QUELL-
CODE**

**BYTE-
CODE**

FRAMEWORK

JAR

IMAGE

BUILD TIME

**PROFILING
C1 COMPILER
PROFILING
C2 COMPILER**

JVM

JIT-COMPILER

**BYTE-
CODE**

**KLASSEN-
LISTE**

**INIT JDK &
FRAMEWORK**

**INIT
APP**

...

**MASCHINEN-
CODE**

RUNTIME

MEHR ZEIT FÜR

TIME TO PEAK

MEHR CPU

JAVA-COMPILER

**QUELL-
CODE**

**BYTE-
CODE**

FRAMEWORK

JAR

IMAGE

**PROFILER +
2 COMPILER
IN APP!**

BUILD TIME

JVM

JIT-COMPILER

**BYTE-
CODE**

**KLASSEN-
LISTE**

**INIT JDK &
FRAMEWORK**

**INIT
APP**

...

**MASCHINEN-
CODE**

RUNTIME

JAVA BY DESIGN:

MEHR ZEIT,

MEHR CPU,

MEHR SPEICHER

BY DESIGN?

"SIMPLE, OBJECT-ORIENTED, DISTRIBUTED,
INTERPRETED, ROBUST, SECURE, ARCHITECTURE-
NEUTRAL, **PORTABLE**, HIGH-PERFORMANCE,
MULTITHREADED DYNAMIC LANGUAGE"

ORIGINALE JAVA-BESCHREIBUNG

BY DESIGN!

JAVA BY DESIGN:

MEHR ZEIT,

MEHR CPU,

MEHR SPEICHER

DARUM LANGSAM!

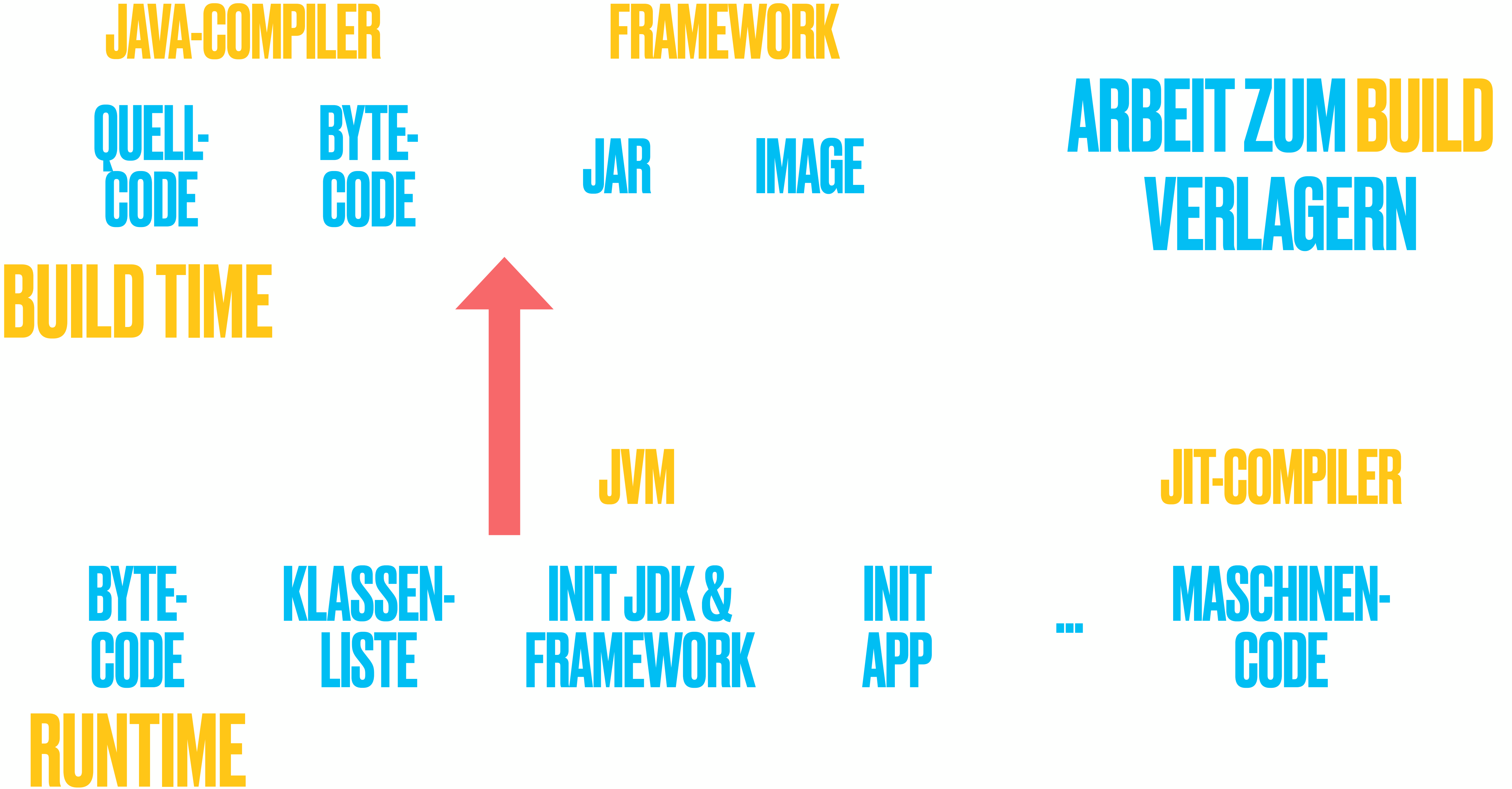
LOHNT SICH DAS?

WARUM JETZT?

WIESO LANGSAM?

WIE SCHNELLER?

WIE SCHNELLER?



JAVA-COMPILER

FRAMEWORK

**QUELL-
CODE**

**BYTE-
CODE**

JAR

IMAGE

**ERGEBNISSE
CACHEN**

BUILD TIME

JVM

JIT-COMPILER

**BYTE-
CODE**

**KLASSEN-
LISTE**

**INIT JDK &
FRAMEWORK**

**INIT
APP**

...

**MASCHINEN-
CODE**

RUNTIME

ARBEIT **ZUM BUILD**

VERLAGERN

ERGEBNISSE **CACHEN**

BEISPIEL: SPRING PETCLINIC

**WEB-ANWENDUNG MIT
THYMELAF, SPRING DATA &
POSTGRESQL**

BENCHMARK:
AWS EC2 T3.SMALL

2 GB RAM, 2 CPUS

**BENCHMARK MIT DIESEN
PARAMETERN:**

-XMS512M -XMX1G
-XX:+USEG1GC
--SPRING.PROFILES.
ACTIVE=POSTGRES

3 WARMUP-LÄUFE,
7 BENCHMARK-LÄUFE,
MITTELWERT VON 5 LÄUFEN

**ERGEBNISSE GELTEN NUR
FÜR PETCLINIC IN MEINER
UMGEBUNG!**

SCRIPTS SIND IN MEINEM
PETGLINIC-**FORK** – MESSEN
SIE FÜR IHRE **ANWENDUNG!**

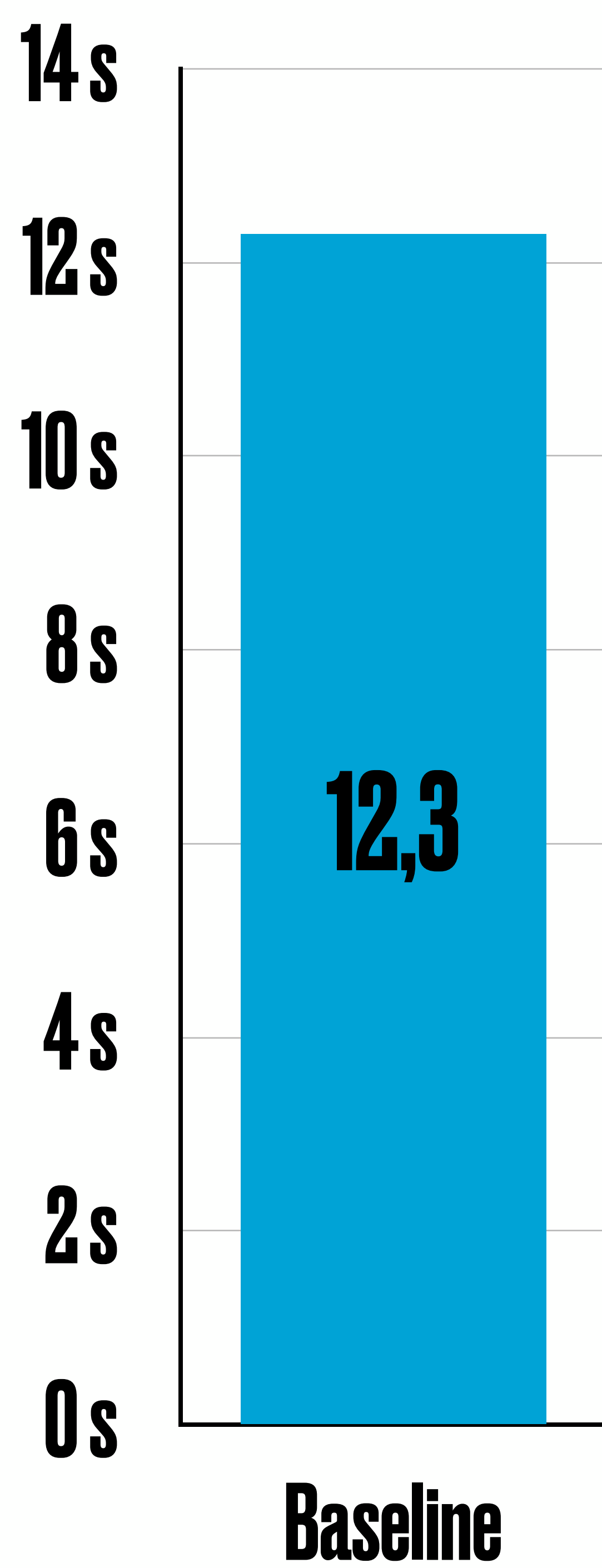
0:

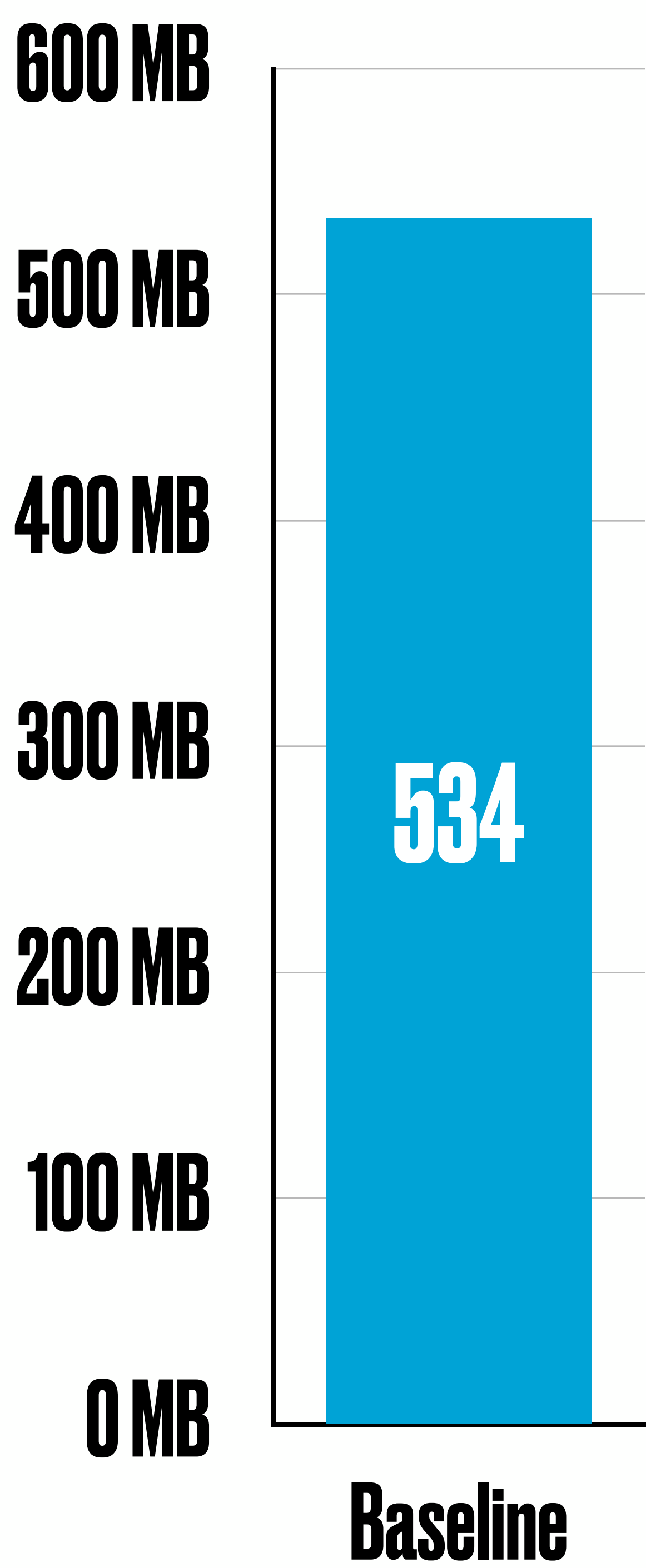
PERFORMANCE **BASLINE**

**SPRING BOOT 3,
JDK 17 (TEST: 21)**

```
./GRADLEW CLEAN BOOTJAR
```

```
JAVA -JAR BUILD/LIBS/MY-APP-1.4.JAR
```





APP JAR: 66 MB

1:

SPRING BOOT TUNING

VOM SPRING-TEAM

VERLAGERT SCHRITTE ZUM BUILD

REDUZIERT STARTUP TIME

SPRING BOOT 3.0+, JDK 17+ (TEST: 21)

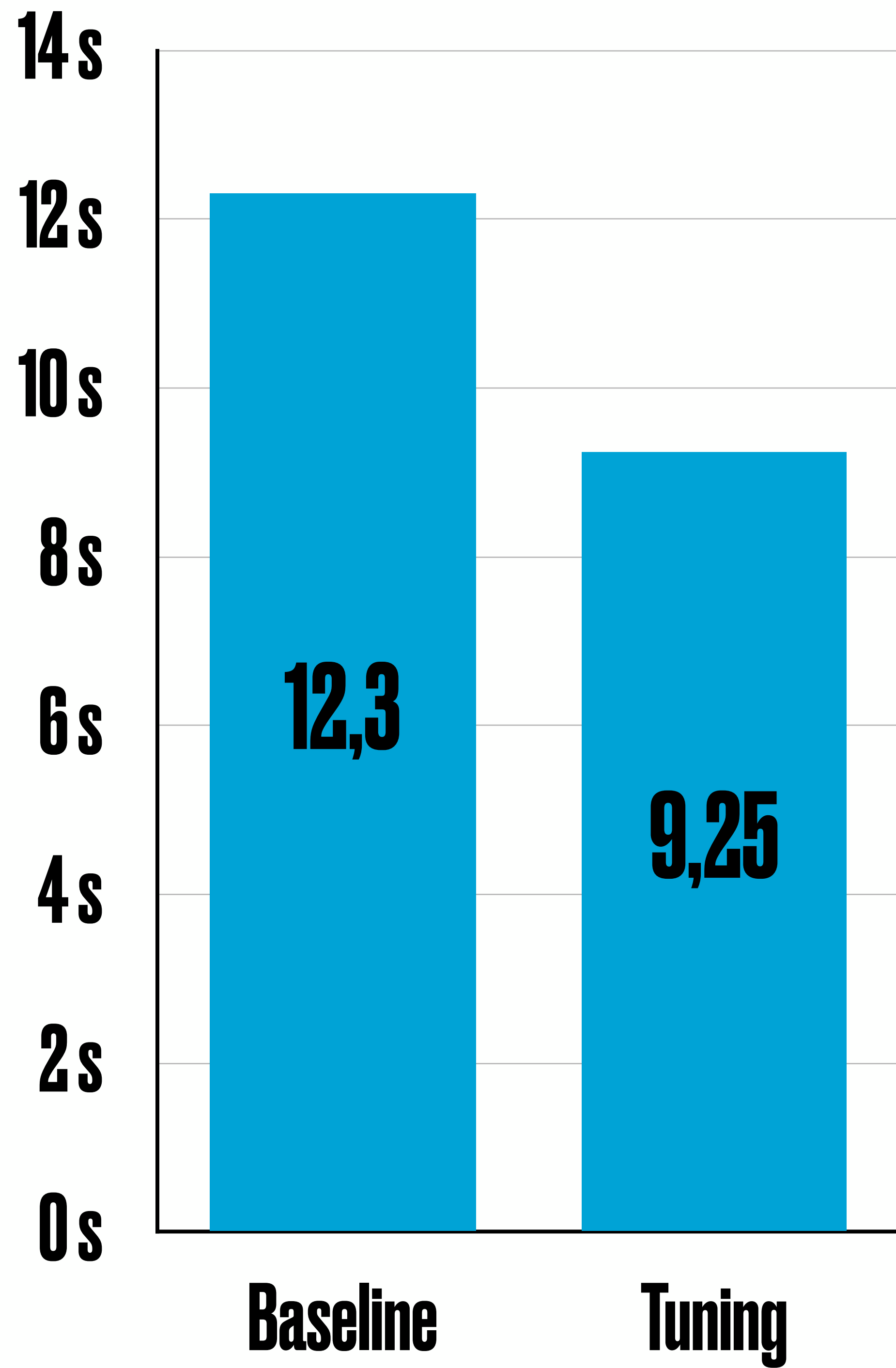
SPRING-BOOT-INITIALISIERUNG
TEILWEISE IM BUILD

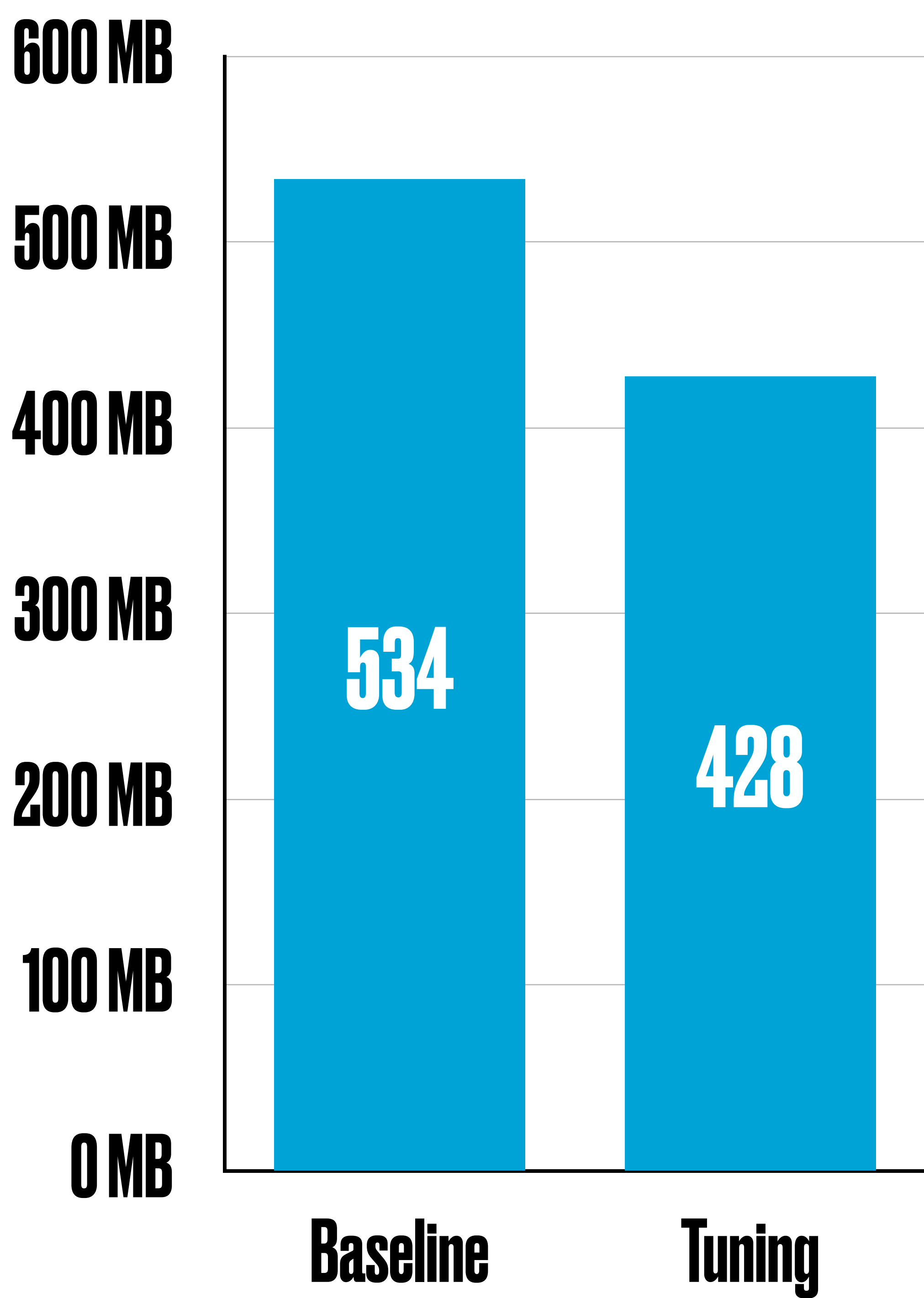
SPRING AOT ENTFERNT NICHT
BENÖTIGTEN CODE & KONFIGURATION

BOOT-JAR ENTPACKT

```
./GRADLEW CLEAN BOOTJAR && \  
  JAVA -DJARMODE=TOOLS \  
  -JAR BUILD/LIBS/MY-APP-1.4.JAR EXTRACT
```

```
JAVA -DSPRING.AOT.ENABLED=TRUE \  
  -JAR MY-APP-1.4/MY-APP-1.4.JAR
```





APP JAR: 1 MB

LIB JARS: 111 MB

PROBLEME?

NISCHEN-BUGS FÜR
SPRING AOT IN
SPRING BOOT 3.5

	TUNING	CDS & LEYDEN	CRAC	GRAALVM
ZEIT: SERVERLESS, STANDBY, VERFÜG- BARKEIT	★☆☆☆☆			
SPEICHER: WENIGER SERVER	★☆☆☆☆			
KOSTEN	★★★★★			
DEV EXPERIENCE	★★★★★			

2:

CLASS DATA SHARING
(CDS) & PROJECT LEYDEN

CDS IN JAVA **EINGEBAUT**, LEYDEN OPEN-JDK-
PROJEKT VON ORACLES **JAVA-TEAM**

CACHET ERGEBNISSE

REDUZIERT **STARTUP** TIME & TIME TO **PEAK**

CDS: JDK **17+** (TEST: **21**),
LEYDEN: JDK **24+** (TEST: **25**)

BRAUCHT ANWENDUNGS-TRAININGSLAUF

**SCHREIBT BEI INITIALISIERUNG
ANFALLENDEN DATEN IN CACHE-DATEI**

**ANWENDUNG LÄDT
CACHE-DATEI IN PRODUKTION**

CDS:

CACHE-DATEI

MY-APP.JSA

PROJECT **LEYDEN:**

AOT CACHE

MY-APP.AOT

```
./GRADLEW CLEAN BOOTJAR && \  
  JAVA -DJARMODE=TOOLS \  
  -JAR BUILD/LIBS/MY-APP-1.4.JAR EXTRACT
```

```
JAVA -DSPRING.AOT.ENABLED=TRUE \  
  -XX:ARCHIVECLASSESATEXIT=MY-APP.JSA  
  -JAR MY-APP-1.4/MY-APP-1.4.JAR
```

```
JAVA -DSPRING.AOT.ENABLED=TRUE \  
  -XX:SHAREDARCHIVEFILE=MY-APP.JSA  
  -JAR MY-APP-1.4/MY-APP-1.4.JAR
```

```
./GRADLEW CLEAN BOOTJAR && \  
  JAVA -DJARMODE=TOOLS \  
  -JAR BUILD/LIBS/MY-APP-1.4.JAR EXTRACT
```

```
JAVA -DSPRING.AOT.ENABLED=TRUE \  
  -XX:AOTCACHEOUTPUT=MY-APP.AOT \  
  -JAR MY-APP-1.4/MY-APP-1.4.JAR
```

```
JAVA -DSPRING.AOT.ENABLED=TRUE \  
  -XX:AOTCACHE=MY-APP.AOT \  
  -JAR MY-APP-1.4/MY-APP-1.4.JAR
```

**WELCHE DATEN
IM CACHE?**

JAVA-COMPILER

FRAMEWORK

**QUELL-
CODE**

**BYTE-
CODE**

JAR

IMAGE

BUILD TIME

JVM

JIT-COMPILER

**BYTE-
CODE**

**KLASSEN-
LISTE**

**INIT JDK &
FRAMEWORK**

**INIT
APP**

...

**MASCHINEN-
CODE**

RUNTIME

LEYDEN (JDK 24):
VERLINKT

LEYDEN (JDK ??):
MASCHINEN-CODE

CDS (JDK 17):
NICHT VERLINKT

LEYDEN (JDK 25):
PROFILING-DATEN

JVM

JIT-COMPILER

BYTE-
CODE

KLASSEN-
LISTE

INIT JDK &
FRAMEWORK

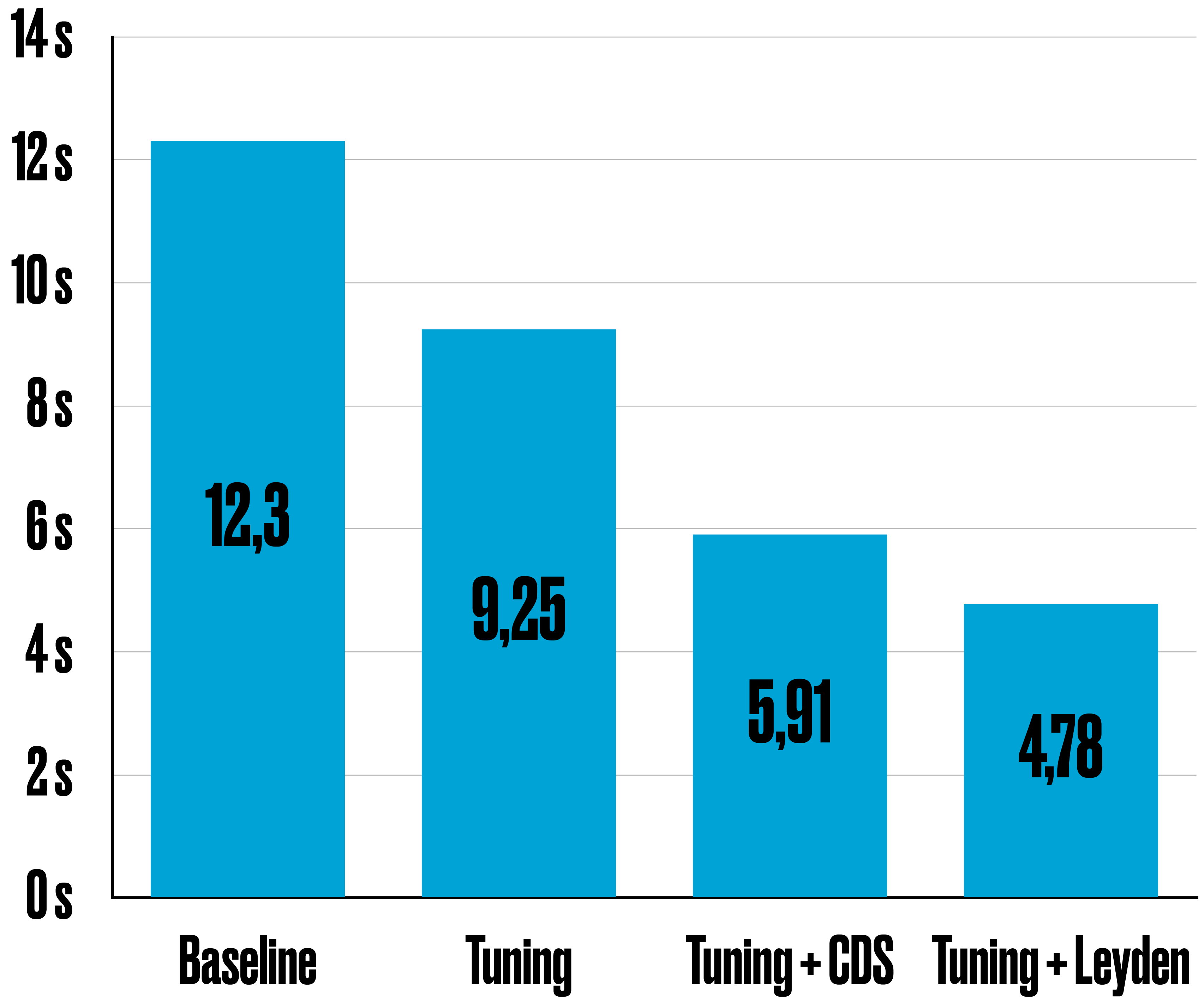
INIT
APP

...

MASCHINEN-
CODE

RUNTIME





600 MB

500 MB

400 MB

300 MB

200 MB

100 MB

0 MB

Baseline

Tuning

Tuning + CDS

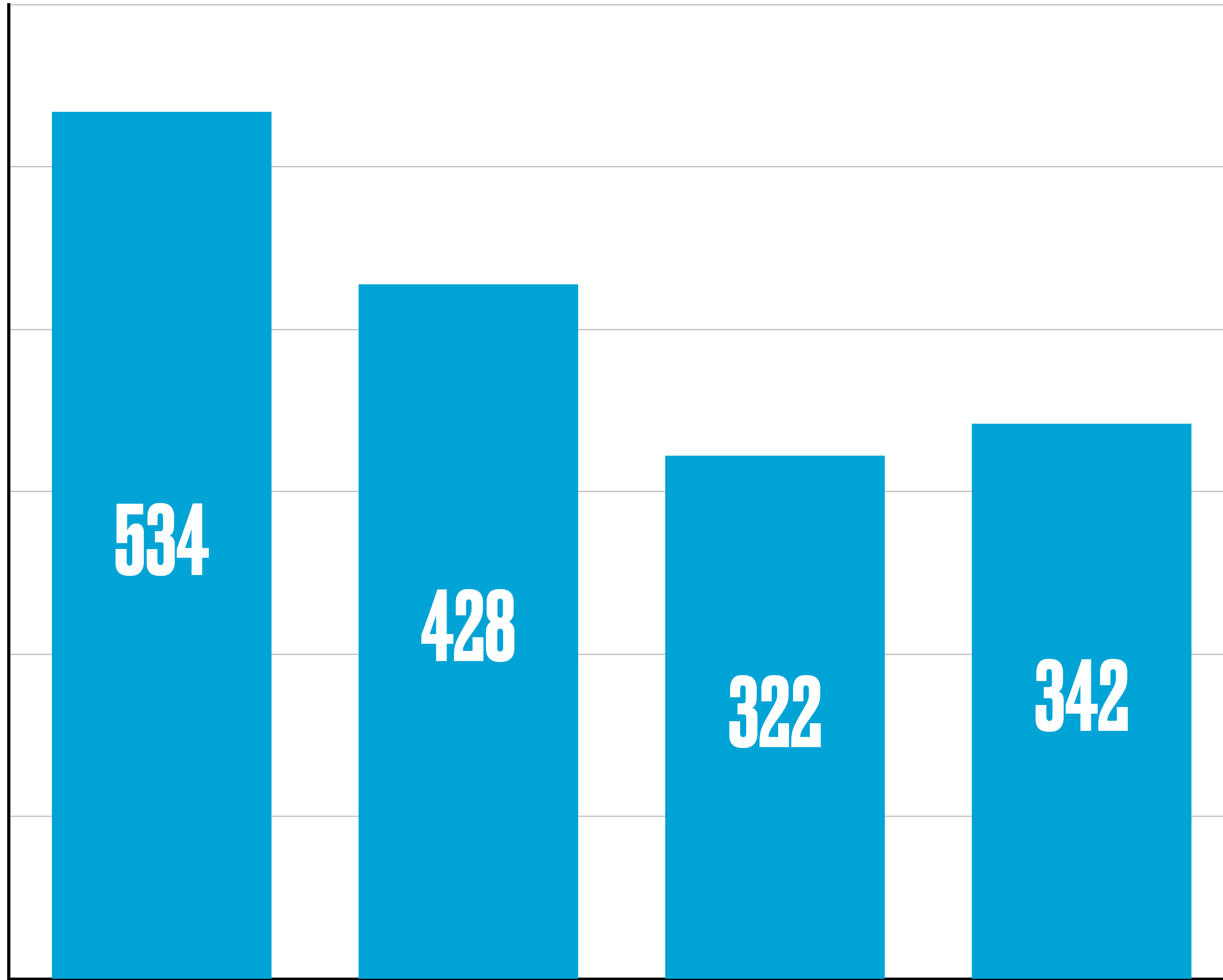
Tuning + Leyden

534

428

322

342



CDS CACHE-DATEI: 101 MB

LEYDEN AOT CACHE:

127 MB

PROBLEME?

PRODUKTIONSNAHER TRAININGSLAUF

CACHE-DATEI **WÄHREND** DES BUILDS & IN
CONTAINER-**IMAGE** (**CLOUD NATIVE
BUILDPACKS** IN SPRING BOOT)...

...ODER **NACH** BUILD & IN PRODUKTION
IN CONTAINER **GEMAPPT**

CACHE-DATEI IN JEDEM
BUILD & FÜR JEDES
BETRIEBSSYSTEM BAUEN,
WEIL SONST UNGÜLTIG

LEYDEN CACHE-BERECHNUNG
BRAUCHT **SPEICHER**

LEYDEN FUNKTIONIERT NICHT MIT
USER-DEFINED CLASS LOADER
(Z.B. QUARKUS)

LEYDEN WIRD
BESSER WERDERN

	TUNING	CDS & LEYDEN	CRAC	GRAALVM
ZEIT: SERVERLESS, STANDBY, VERFÜG- BARKEIT				
SPEICHER: WENIGER SERVER				
KOSTEN				
DEV EXPERIENCE				

3:

PROJECT CRAC

“COORDINATED RESTORE AT CHECKPOINT”

OPEN-JDK-PROJEKT, INITIAL VON AZUL

CACHET ERGEBNISSE

REDUZIERT STARTUP TIME & TIME TO PEAK

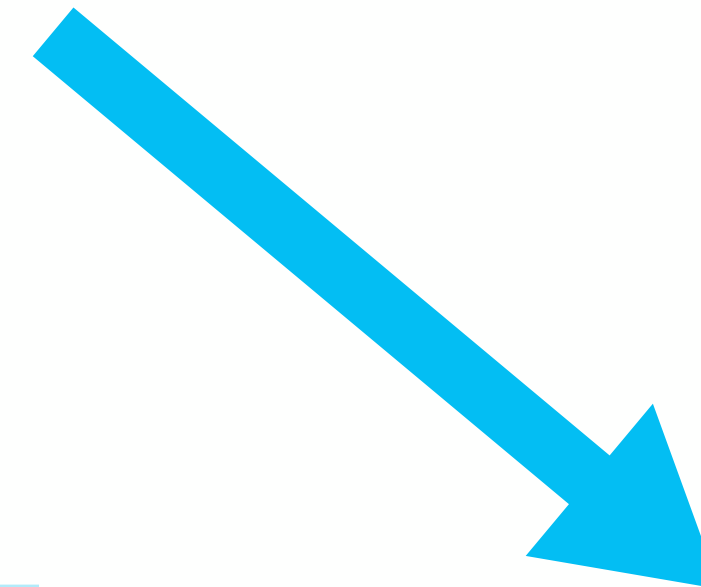
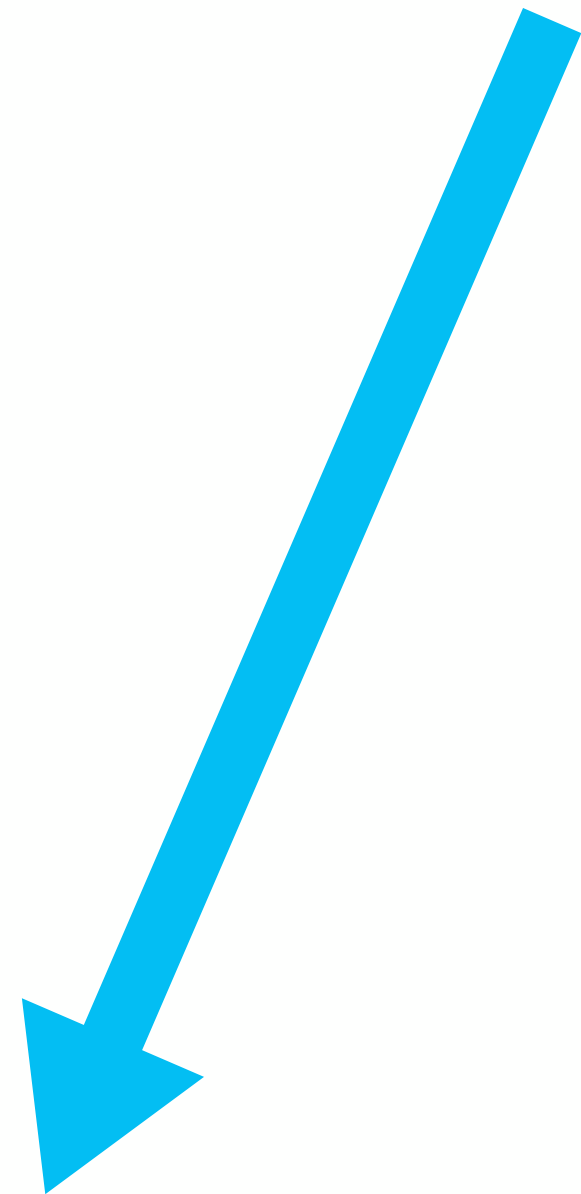
JDK 17 (TEST: 24)

LEYDEN VS. CRAC

LEYDEN (JDK 24):
VERLINKT

LEYDEN (JDK ??):
MASCHINEN-CODE

LEYDEN (JDK 25):
PROFILING-DATEN



JVM

JIT-COMPILER

BYTE-
CODE

KLASSEN-
LISTE

INIT JDK &
FRAMEWORK

INIT
APP

...

MASCHINEN-
CODE

RUNTIME

KOMPLETTEN APPLIKATION-SPEICHER ALS MEMORY-SNAPSHOT SPEICHERN & SPÄTER LADEN

JVM

JIT-COMPILER

BYTE-
CODE

KLASSEN-
LISTE

INIT JDK &
FRAMEWORK

INIT
APP

...

MASCHINEN-
CODE

RUNTIME

TRAININGSLAUF:



PRODUKTION:



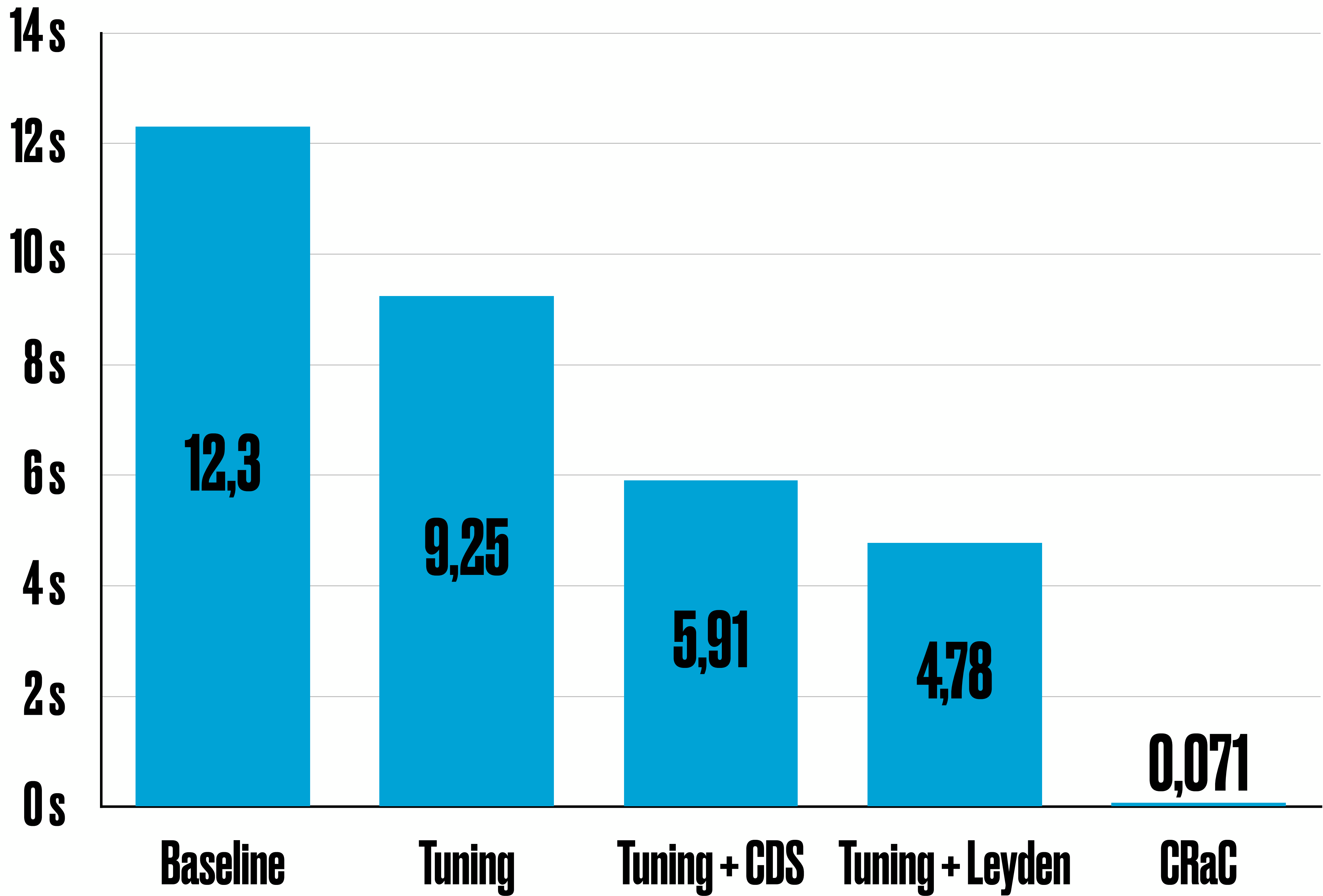
NUTZER/APPLIKATION LÖSEN
CHECKPOINT IM TRAININGSLAUF AUS:
ERZEUGT **SNAPSHOT** ALS **MEMORY DUMP**
VON **ALLEN** DATEN DER ANWENDUNG,
BIBLIOTHEKEN, JVM & HOTSPOT

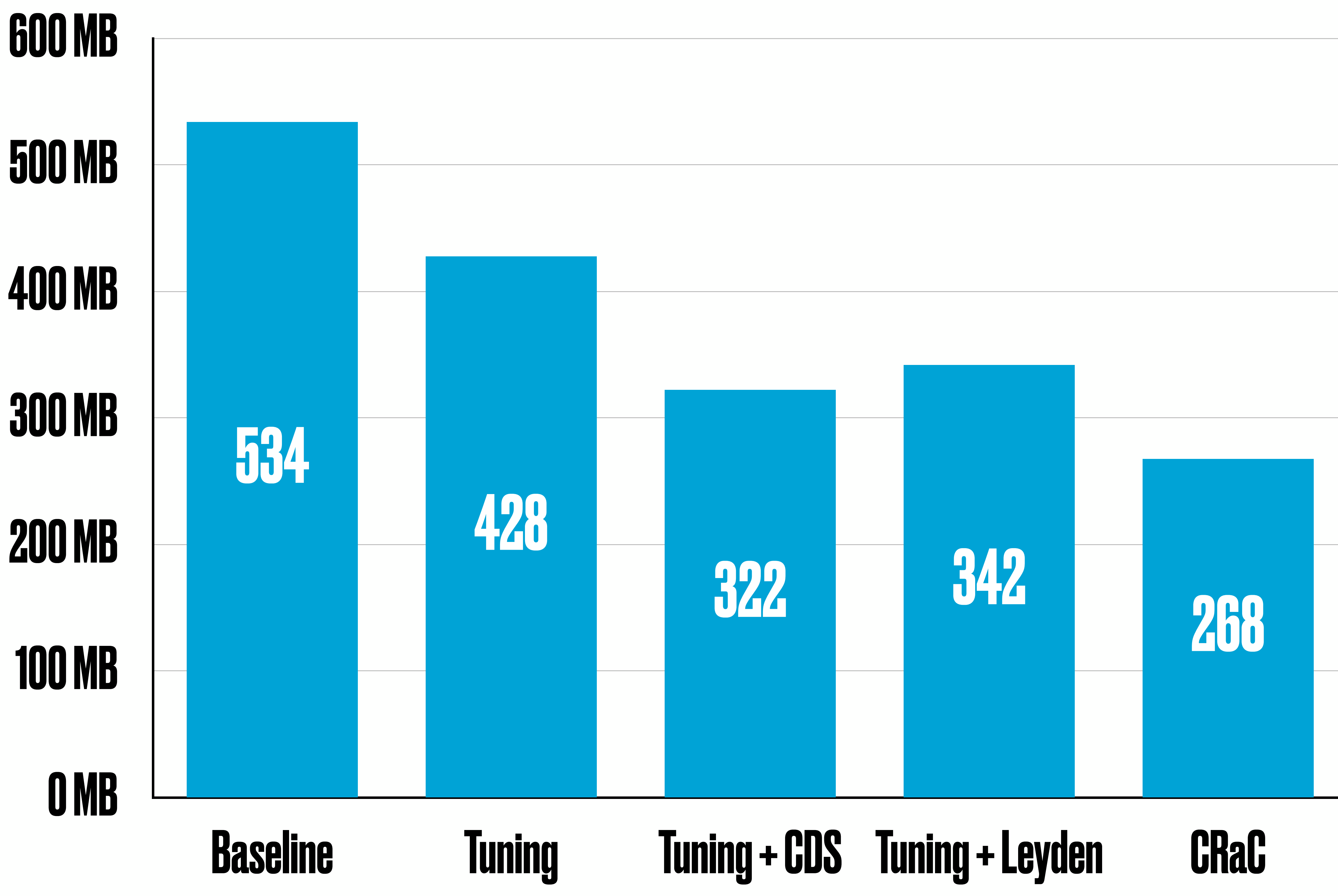
```
./GRADLEW CLEAN BOOTJAR -PCRAC=TRUE
```

```
JAVA -XX:CRACENGINE=WARP \  
-XX:CRACCHECKPOINTTO=MY-APP-CRAC \  
-JAR BUILD/LIBS/MY-APP-1.4.JAR
```

```
JCMD [PID] JDK.CHECKPOINT
```

```
JAVA -XX:CRACRESTOREFROM=MY-APP-CRAC
```





SNAPSHOT: 202 MB

PROBLEME?

**"CRAC TRIES TO MAKE THINGS RUN FASTER
POTENTIALLY AT THE EXPENSE OF CORRECT-
NESS, AND I DON'T LIKE THAT TRADE-OFF."**

**BRIAN GOETZ, JAVA LANGUAGE
ARCHITECT BEI ORACLE**

VOR SNAPSHOT-SPEICHERN ALLE
DATEIEN & PORTS SCHLIEßEN...

...UND **NACH** SNAPSHOT-LADEN
WIEDER **ÖFFNEN**

CRAC BRAUCHT SUPPORT VON **JDK,
FRAMEWORK, BIBLIOTHEKEN & ANWENDUNG**

JDK: AZUL, BELLSOFT LIBERICA,
CANONICAL/UBUNTU

FRAMEWORKS: SPRING BOOT 3.2+,
QUARKUS 2.10.0, MICRONAUT 3.7, HELIDON 4.2

CRAC FUNKTIONIERT **NUR IN LINUX**

– NOOP-IMPLEMENTIERUNG IN
WINDOWS & MACOS

BRAUCHT CRAC JAVA-**BIBLIOTHEK**

SICHERHEITSRISIKO SNAPSHOT:

PASSWÖRTER, SECRETS,

BYTECODE & MASCHINENCODE

IM **KLARTEXT**

**VERSCHLÜSSELUNG FÜR
SNAPSHOTS GEPLANT**

	TUNING	CDS & LEYDEN	CRAC	GRAALVM
ZEIT: SERVERLESS, STANDBY, VERFÜG- BARKEIT	★ ★ ★ ★ ★	★ ★ ★ ★ ★ ★ ★ ★ ★ ★	★ ★ ★ ★ ★	
SPEICHER: WENIGER SERVER	★ ★ ★ ★ ★	★ ★ ★ ★ ★	★ ★ ★ ★ ★	
KOSTEN	★ ★ ★ ★ ★	★ ★ ★ ★ ★	★ ★ ★ ★ ★	
DEV EXPERIENCE	★ ★ ★ ★ ★	★ ★ ★ ★ ★	★ ★ ★ ★ ★	

4:

GRAALVM NATIVE IMAGE

PROJEKT VON **ORACLE LABS**

VERLAGERT SCHRITTE ZUM **BUILD**

REDUZIERT **STARTUP** TIME & TIME TO **PEAK**

SPRING BOOT **3**,
GRAALVM JDK **17** (HIER: **24**)

**"JAVA MINUS
JIT-COMPILER"**

ORACLE LABS !=
ORACLE JAVA-TEAM

AHEAD-OF-TIME-COMPILER (AOT)
“NATIVE IMAGE” BAUT AUSFÜHRBARE
PROGRAMME FÜR LINUX,
WINDOWS & MACOS
BENÖTIGT SPRING AOT

**NATIVE IMAGE ENTFERNT
BEIM BUILD ALLE NICHT
BENÖTIGTEN KLASSEN &
RESSOURCEN**

PROFILE-GUIDED OPTIMIZATION
(**PGO**): **TRAININGSLAUF** MIT
INSTRUMENTIERTER ANWENDUNG,
DIE **PROFILING**-DATEN FÜR ZWEITEN
COMPILIERUNGS-LAUF SAMMELT

MEHR **SICHERHEIT**: KEIN BÖSARTIGER
BYTECODE LADBAR ZUR LAUFZEIT,
SICHERHEITSLÜCKEN IN UNBENÜTZTEM
CODE ENTFERNT

ANWENDUNGEN **KLEINER** ALS JAR + JRE

ZUR LAUFZEIT IMMER NOCH

JVM ("SUBSTRATE VM")
FÜR THREADS & GARBAGE
COLLECTION

GRAALVM BRAUCHT SUPPORT VON FRAMEWORK, BIBLIOTHEKEN & ANWENDUNG

FRAMEWORKS: SPRING BOOT 3.0+,
QUARKUS, MICRONAUT, HELIDON

BIBLIOTHEKEN: ÖFFENTLICHES
KONFIGURATIONS-REPOSITORY

GRAALVM JDK = OPENJDK + NATIVE IMAGE
+ **GRAAL JIT-COMPILER**
(HOTSPOT-ERSATZ)

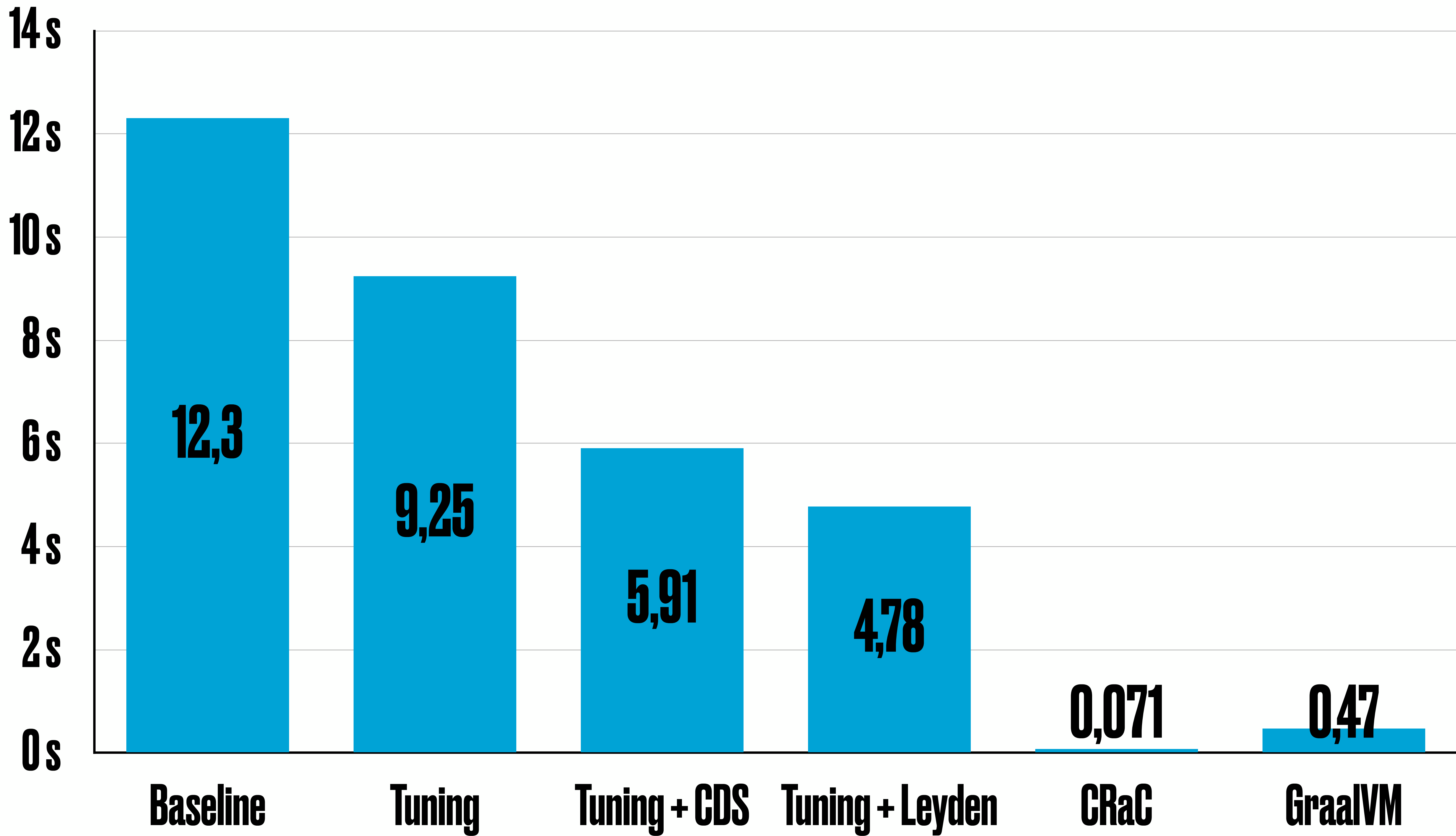
SEPARATER DOWNLOAD, ABER RELEASES
SYNCHRON MIT OPENJDK

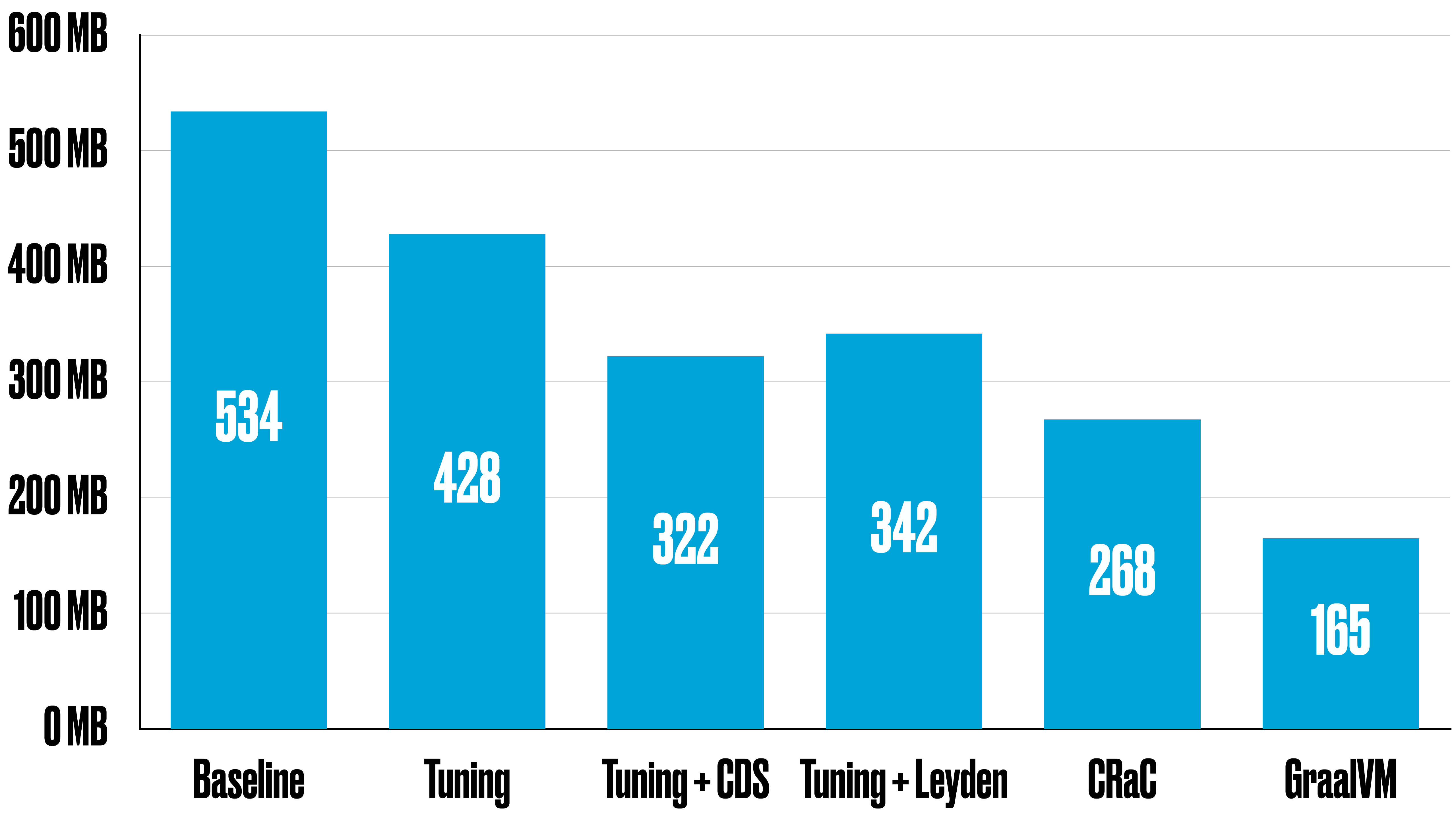
./GRADLEW NATIVECOMPILE --PGO-INSTRUMENT

./BUILD/NATIVE/NATIVECOMPILE/MY-APP-
INSTRUMENTED

./GRADLEW NATIVECOMPILE

./BUILD/NATIVE/NATIVECOMPILE/MY-APP





LINUX APP: 178 MB

NATIVE IMAGE: 10 MIN UND 5 MIN

MIT 32 GB RAM & 8 CPUS

(T3.2XLARGE)

PROBLEME?

EINIGE JAVA-FEATURES
UNMÖGLICH, ANDERE
FUNKTIONIEREN NUR **TEILWEISE**,
ANDERE ERFORDERN
KONFIGURATION

**REFLECTION & EXPLIZITES KLASSEN-
LADEN** ERFORDERN KONFIGURATION

GRAALVM KANN KONFIGURATION
AUTOMATISCH DURCH "MITHÖREN
BEI JIT-JAVA" ERZEUGEN

NERVIG

LANGE BUILD-ZEITEN & VIEL SPEICHER

EINE ANWENDUNG PRO BETRIEBSSYSTEM

DEBUGGEN AUF MAC/WINDOWS EXTREM

ZÄH, WEIL ANWENDUNG IN LINUX-

CONTAINER LÄUFT

	TUNING	CDS & LEYDEN	CRAC	GRAALVM
ZEIT: SERVERLESS, STANDBY, VERFÜG- BARKEIT				
SPEICHER: WENIGER SERVER				
KOSTEN				
DEV EXPERIENCE				

SO SCHNELLER!

LOHNT SICH DAS?

WARUM JETZT?

WIESO LANGSAM?

WIE SCHNELLER?

LOHNT SICH DAS?

WARUM JETZT?

WIESO LANGSAM?

WIE SCHNELLER?

ZUSAMMENFASSUNG

WANN KANN ES
SICH LOHNEN, JAVA
SCHNELLER ZU STARTEN?

SERVERLESS,
VIELE SERVER,
SEHR HOHE VERFÜGBARKEIT

JAVA BY DESIGN:

MEHR ZEIT,

MEHR CPU,

MEHR SPEICHER

1. SPRING BOOT TUNING

**2. CLASS DATA SHARING (CDS) &
PROJECT LEYDEN**

3. PROJECT CRAC

4. GRAALVM NATIVE IMAGE

The image is a stylized title card. It features a series of concentric circles in shades of red, creating a tunnel-like effect that draws the eye toward the center. The circles are slightly offset, giving a 3D or layered appearance. In the very center is a solid black circle. Overlaid on this central black circle is the text "The End" in a white, elegant, cursive script. The text has a subtle drop shadow, making it stand out against the black background.

The End

MEIN VORTRAG HÖRT
NICHT AUF, WENN ICH
AUFHÖRE VORZUTRAGEN!

FOLIEN, CODE & MEHR



[HTTPS://BPF.LI/JF5](https://bpf.li/jf5)