# Spring Boot 3.3.2 and Java 21 Performance Benchmark

Web, Reactive,
CDS, AOT, Virtual Threads,
JVM, and Native

**@ivanfranchin**

# Goal

We implemented two **Spring Boot** applications:

- one using **Spring Web** (**Spring MVC** with **Apache Tomcat**);
- the other using **Spring Reactive Web** (**Spring WebFlux** with **Netty**).

We built both **JVM** and **Native Docker images** for different configurations, including options for **virtual threads**, **CDS**, and **AOT** optimizations, and compared them.

Let's understand the strengths and trade-offs of each approach.

# Java

**Java** has added many features over the years to improve performance:

- **Class Data Sharing (CDS)**: Introduced in Java 5, it allows classes to be pre-processed into a shared archive file that can be memory-mapped at runtime, reducing startup time and dynamic memory usage when multiple JVMs share the same archive.
- **Application CDS (AppCDS)**: Introduced in Java 10 (JEP-310), it extends CDS to include application classes in the shared archive.
- **Virtual Threads**: First previewed in Java 19 and fully supported in Java 21 (JEP-444), these are lightweight threads provided by the JDK instead of the OS.
- **Ahead of Time (AOT) Compilation**: Introduced in Java 9 (JEP-295), it improves startup time by compiling Java classes to native code before launching the JVM.

# Spring Boot

**Spring Boot** has adapted to these Java improvements, making it easier to use new features like **CDS**, **virtual threads**, and **AOT** in Spring apps.

Spring Boot provides two main ways to build web apps:

- **Spring Web**: Uses traditional Spring MVC, a synchronous model with Apache Tomcat as the default web server.
- **Spring WebFlux**: For reactive, non-blocking apps, using Netty.

# Docker

When your Java Spring Boot app is ready for production, building and deploying it as a **Docker** image is recommended.

There are two main types of Docker images for Java apps:

- **JVM Docker images**: Use the traditional Java setup with the JVM, offering fast compilation but slower startup and higher memory usage.
- **Native Docker images**: Compiled with GraalVM Native Image, they start faster and use less memory but are more complex to create and have a slower compilation process.

# Which one should we choose?

- Should we go for a fully reactive app using Spring Reactive Web?
- If we're not comfortable with reactive programming, what about using Spring Web with Virtual Threads? Will it perform as well as a Spring Reactive Web app?
- No matter which approach we pick, can we make it better with CDS and AOT?
- Will these make it start faster and use less memory?
- Does a native app really use less memory than a JVM one?

# Spring Boot Web and Reactive apps

We created two Spring Boot Greetings API applications:

- **spring-boot-greetings-api-web**
- **spring-boot-greetings-api-reactive**

They have a simple business logic that exposes an endpoint **/greetings?name=?**

The endpoint returns a greeting. If no name is provided, it will return "[greeting-word] World!". Otherwise, it will return "[greeting-word] [name]!".

The greeting word is chosen randomly by a service class, which can easily be replaced by a database call or an external API. **To simulate processing time**, we have added a **delay of 1 seconds** to the word selection.

# Benchmark and Metrics

To collect important data from our Docker containers, we will use **ivangfr/api-oha-benchmarker**. It uses:

- **Testcontainers**: to manage Docker containers.
- **OHA**: to load testing and to obtain metrics such as the slowest, fastest, and average request times in seconds;
- **docker stats**: to collect information like CPU and memory usage.

The benchmark will involve load testing both the JVM and Native Docker images for each configuration.

For each iteration:

1. Start the container
2. Conduct rounds of OHA testing with **100**, **300**, **900**, and finally **2700** concurrent requests.
3. Shut down the container.

***@ivanfranchin***

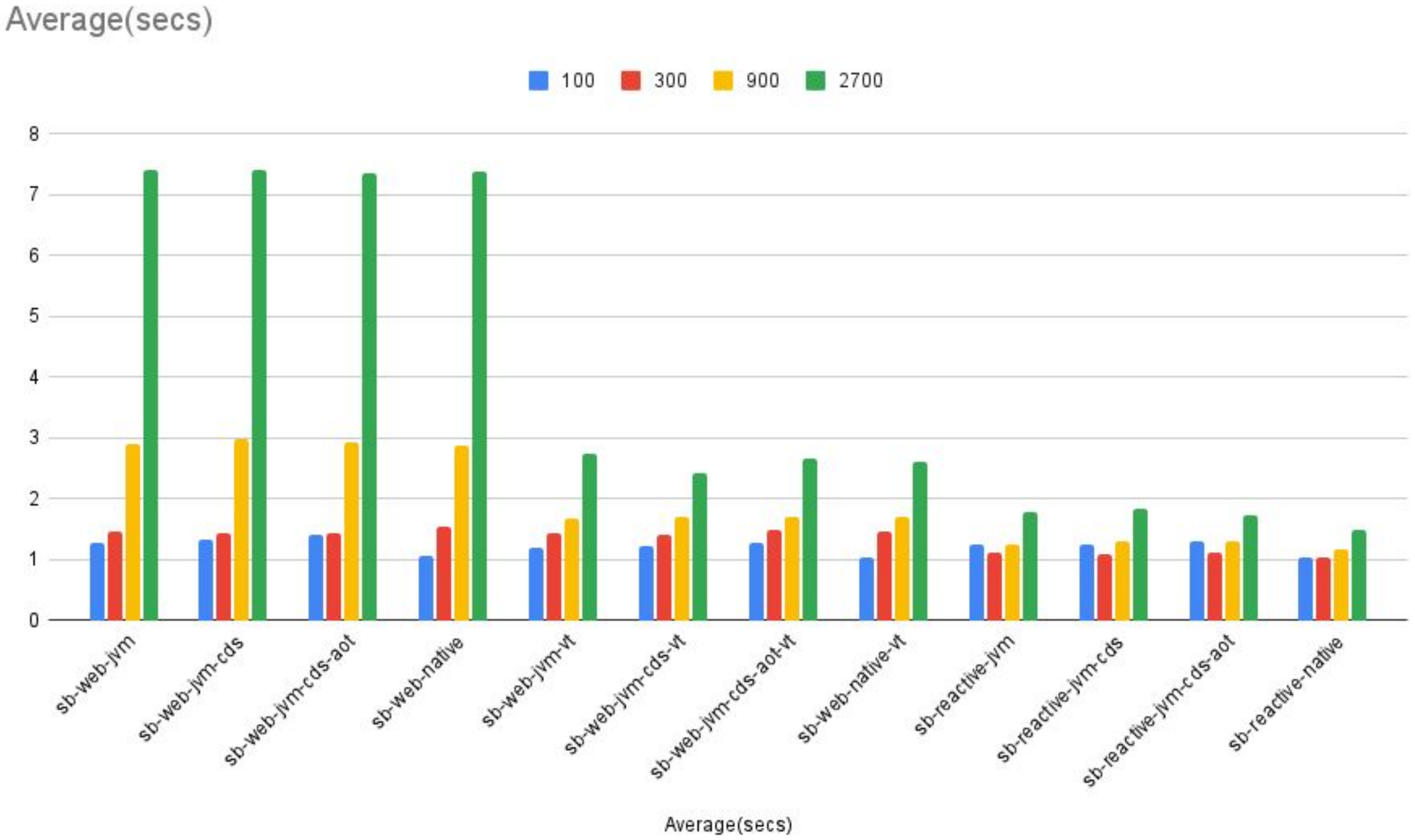
# Configuration and Container Name

| **Configuration** | **Apps name** |
|---|---|
| **JVM** Greetings API **Web** | **sb-web-jvm** |
| **JVM** Greetings API **Web** with **CDS** enabled | **sb-web-jvm-cds** |
| **JVM** Greetings API **Web** with **CDS** and **AOT** enabled | **sb-web-jvm-cds-aot** |
| **Native** Greetings API **Web** | **sb-web-native** |
| **JVM** Greetings API **Web** with **Virtual Threads** enabled | **sb-web-jvm-vt** |
| **JVM** Greetings API Web with **CDS** and **Virtual Threads** enabled | **sb-web-jvm-cds-vt** |
| **JVM** Greetings API Web with **CDS**, **AOT** and **Virtual Threads** enabled | **sb-web-jvm-cds-aot-vt** |
| **Native** Greetings API Web with **Virtual Threads** enabled | **sb-web-native-vt** |
| **JVM** Greetings API **Reactive** Web | **sb-reactive-jvm** |
| **JVM** Greetings API **Reactive** Web with **CDS** enabled | **sb-reactive-jvm-cds** |
| **JVM** Greetings API **Reactive** Web with **CDS** and **AOT** enabled | **sb-reactive-jvm-cds-aot** |
| **Native** Greetings API **Reactive** Web | **sb-reactive-native** |

***@ivanfranchin***

# Benchmark Results

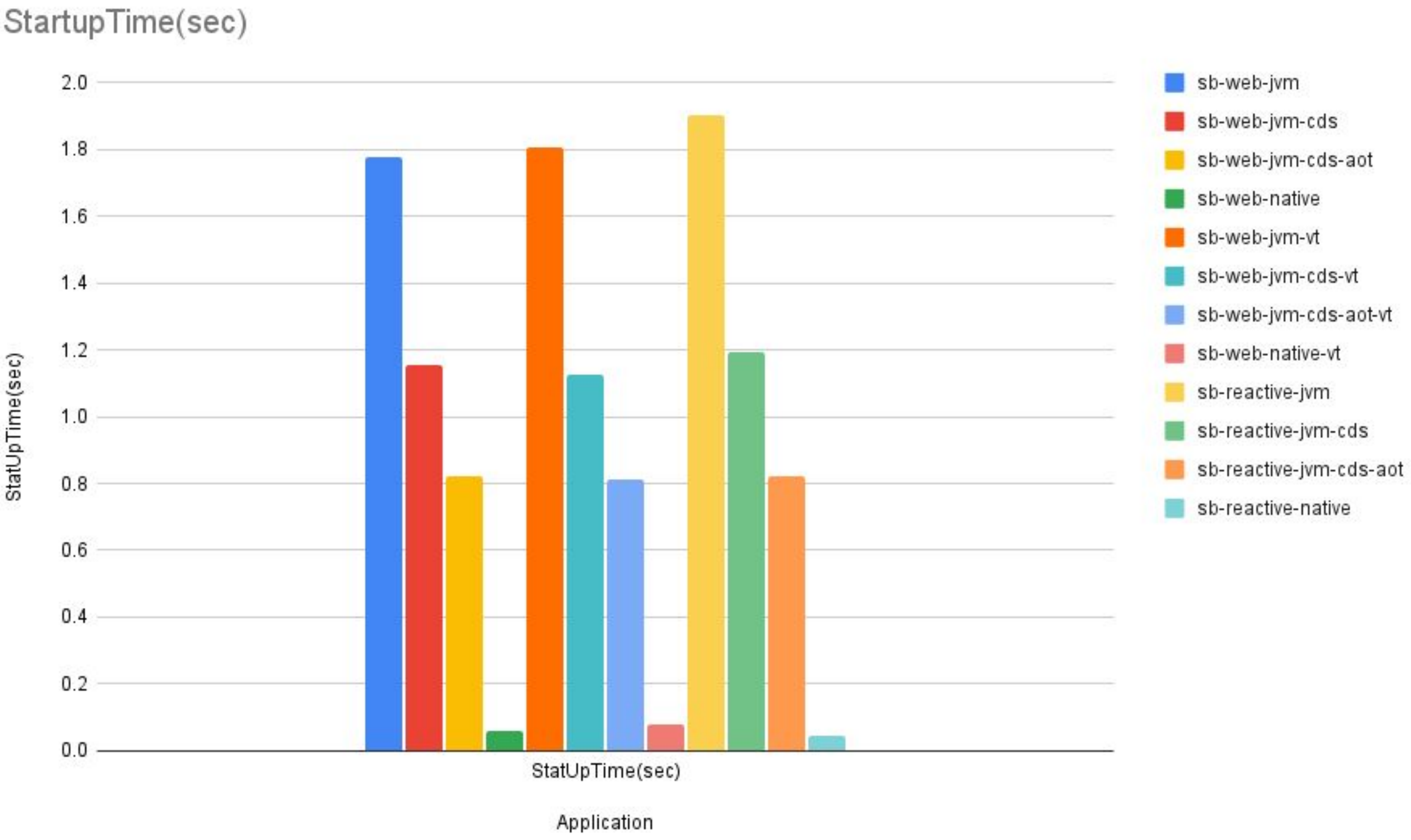
Average Time per Request



**@ivanfranchin**

# Benchmark Results

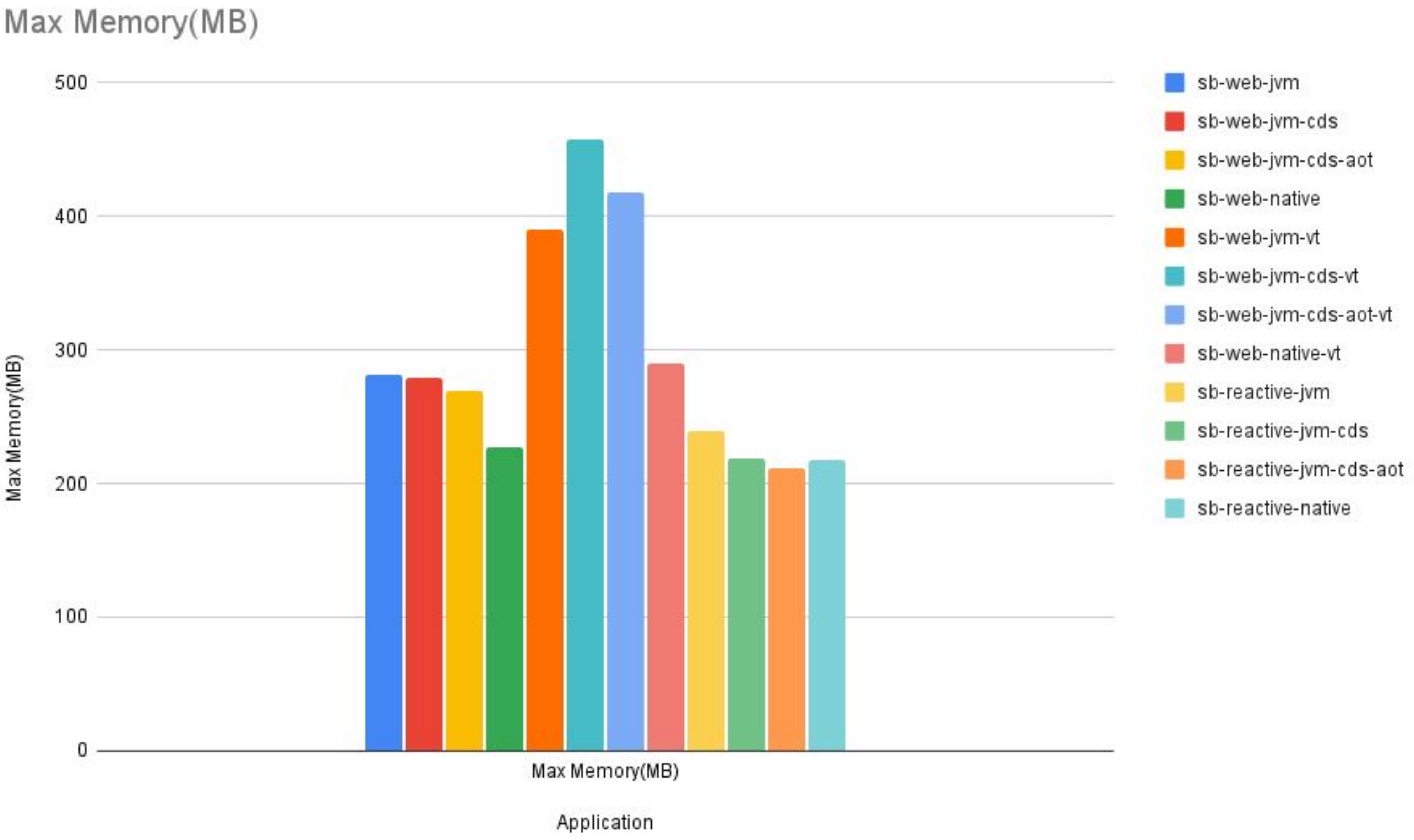
Startup Time



***@ivanfranchin***

# Benchmark Results

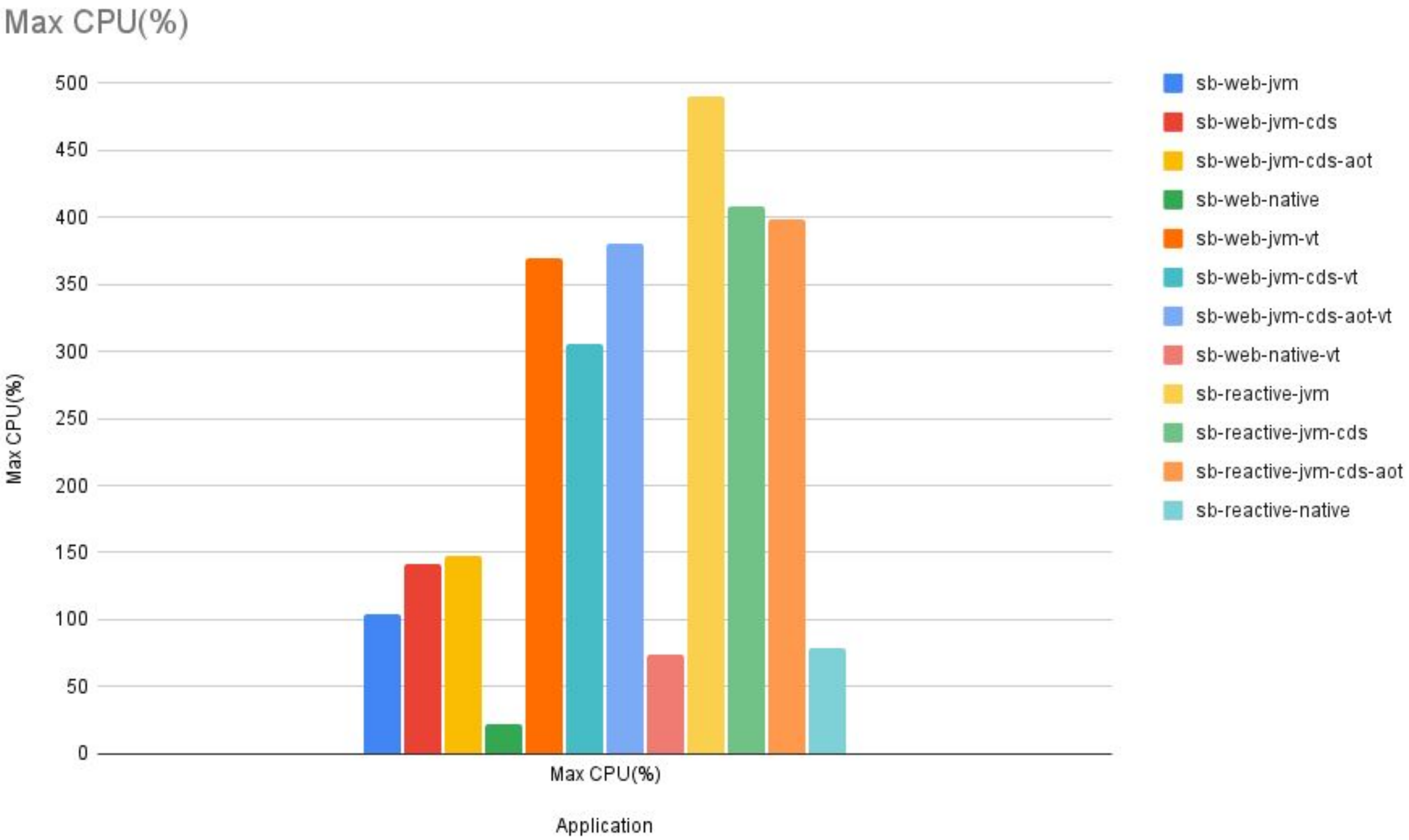
Maximum Memory Usage



**@ivanfranchin**

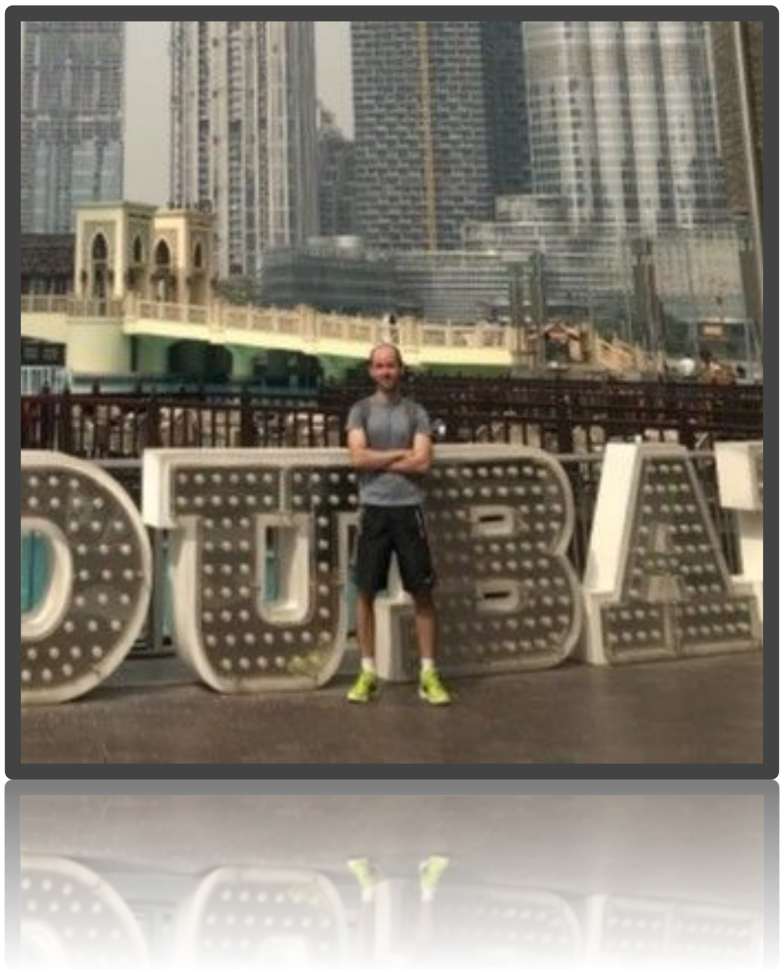
# Benchmark Results

Maximum CPU Usage



@ivanfranchin

# That’s all



You can read more about Java Concurrency in the **Medium** article:

**"Spring Boot 3.3.2 Benchmark: Web, Reactive, CDS, AOT, Virtual Threads, JVM, and Native"**

***Let’s connect:*** in ***@ivanfranchin***

***Follow me:*** ***@ivangfr***