GraalVM Native Image

Shortcut or longcut towards JVM Based micro-services

CADEC 2021 Peter Larsson

JVM based Micro Services

1. Large memory footprint
2. Long startup time
3. Initial warmup required (JIT)

Makes it expensive for large systems and impossible to scale to zero.

GraalVM to the Rescue?

"Run Programs Faster Anywhere"

- Increase application throughput and reduced latency
- Compile applications into small self-contained native binaries
- Seamlessly use multiple languages and libraries

GraalVM Provides 2 Editions

- Community Edition (GPL with classpath exception)
- Enterprise Edition (Commercial, Oracle Supported)

Native Considerations...

Supported

- Unsafe Memory Access
- References
- Threads
- Signal Handlers

Requires Configuration

- Reflections, Dynamic Class Loading
- Dynamic Proxies (JDK)
- Resource Access
- Java Native Interface (JNI)

Unsupported

- CGLIB, Invoke Dynamic and Method Handles, Finalizers, Security Manager, JVMTI

...Considerations cont'd

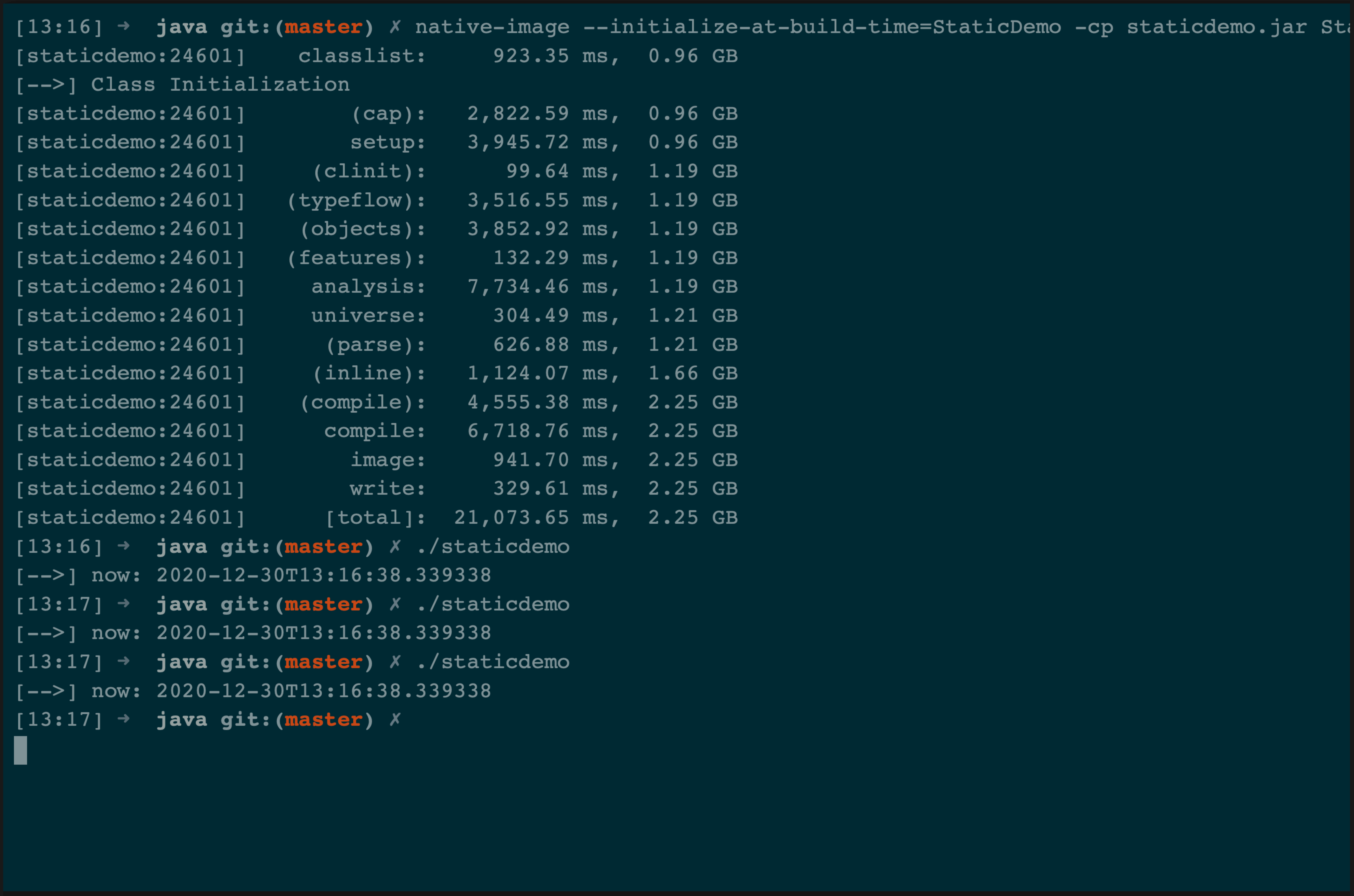
- Understand build-time vs. run-time (default) class initialization
- Frameworks/libs without native support
- Use and maintain configurations for Reflection, Proxies, Resources and JNI
  - Static config or dynamic as code

Build Time Class Initialization

```java
public class StaticDemo {
        static final LocalDateTime NOW = LocalDateTime.now();
        static {
                log("Class Initialization");
        }
        public static void main(String[] args) {
                log("now: " + NOW);
        }
        static void log(String msg) {
                System.out.println("[-->] " + msg);
        }
}
```

```
$ java -cp staticdemo.jar StaticDemo
[-->] Class Initialization
[-->] now: 2020-12-29T14:41:17.569383
```

Build Native Image (5x speed)

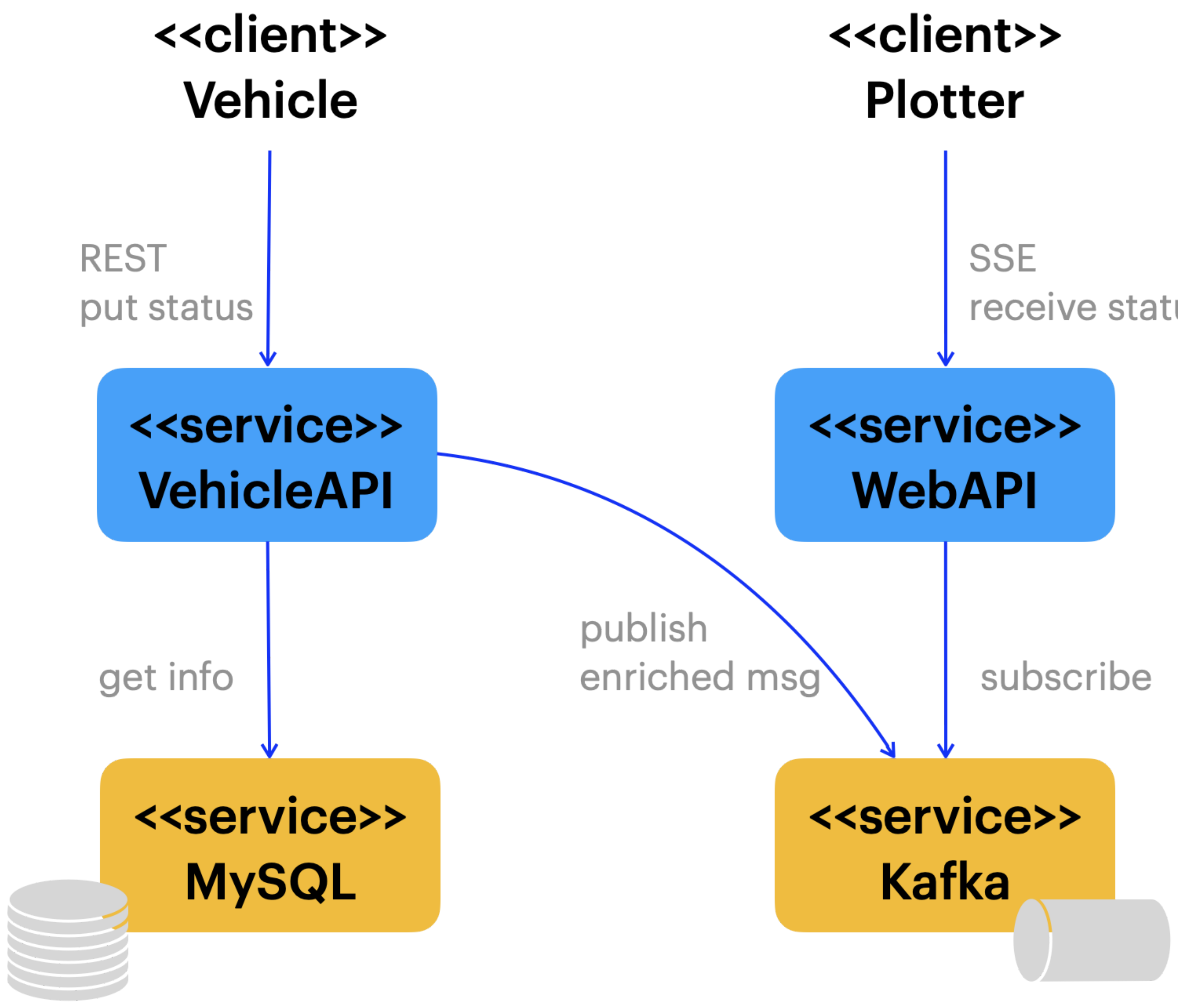


```
[13:16] ➜  java git:(master) ✗ native-image --initialize-at-build-time=StaticDemo -cp staticdemo.jar Sta
[staticdemo:24601]    classlist:     923.35 ms,  0.96 GB
[-->] Class Initialization
[staticdemo:24601]        (cap):   2,822.59 ms,  0.96 GB
[staticdemo:24601]        setup:   3,945.72 ms,  0.96 GB
[staticdemo:24601]     (clinit):      99.64 ms,  1.19 GB
[staticdemo:24601]   (typeflow):   3,516.55 ms,  1.19 GB
[staticdemo:24601]    (objects):   3,852.92 ms,  1.19 GB
[staticdemo:24601]   (features):     132.29 ms,  1.19 GB
[staticdemo:24601]     analysis:   7,734.46 ms,  1.19 GB
[staticdemo:24601]     universe:     304.49 ms,  1.21 GB
[staticdemo:24601]      (parse):     626.88 ms,  1.21 GB
[staticdemo:24601]     (inline):   1,124.07 ms,  1.66 GB
[staticdemo:24601]    (compile):   4,555.38 ms,  2.25 GB
[staticdemo:24601]      compile:   6,718.76 ms,  2.25 GB
[staticdemo:24601]        image:     941.70 ms,  2.25 GB
[staticdemo:24601]        write:     329.61 ms,  2.25 GB
[staticdemo:24601]      [total]:  21,073.65 ms,  2.25 GB
[13:16] ➜  java git:(master) ✗ ./staticdemo
[-->] now: 2020-12-30T13:16:38.339338
[13:17] ➜  java git:(master) ✗ ./staticdemo
[-->] now: 2020-12-30T13:16:38.339338
[13:17] ➜  java git:(master) ✗ ./staticdemo
[-->] now: 2020-12-30T13:16:38.339338
[13:17] ➜  java git:(master) ✗
```

Spring Boot and GraalVM Native

- Spring team collaborates with GraalVM and also 3rd party library projects (Tomcat, Netty, ...)
- No need for CGLIB proxies
  - `@SpringBootApplication(proxyBeanMethods = false)`
  - `@Configuration(proxyBeanMethods = false)`
- spring-graal-native project
  - Provides a GraalVM `@AutomaticFeature`
  - Configures GraalVM Native (dynamic inspection of app and deps)

FleetDemo App (Spring Boot and Go)



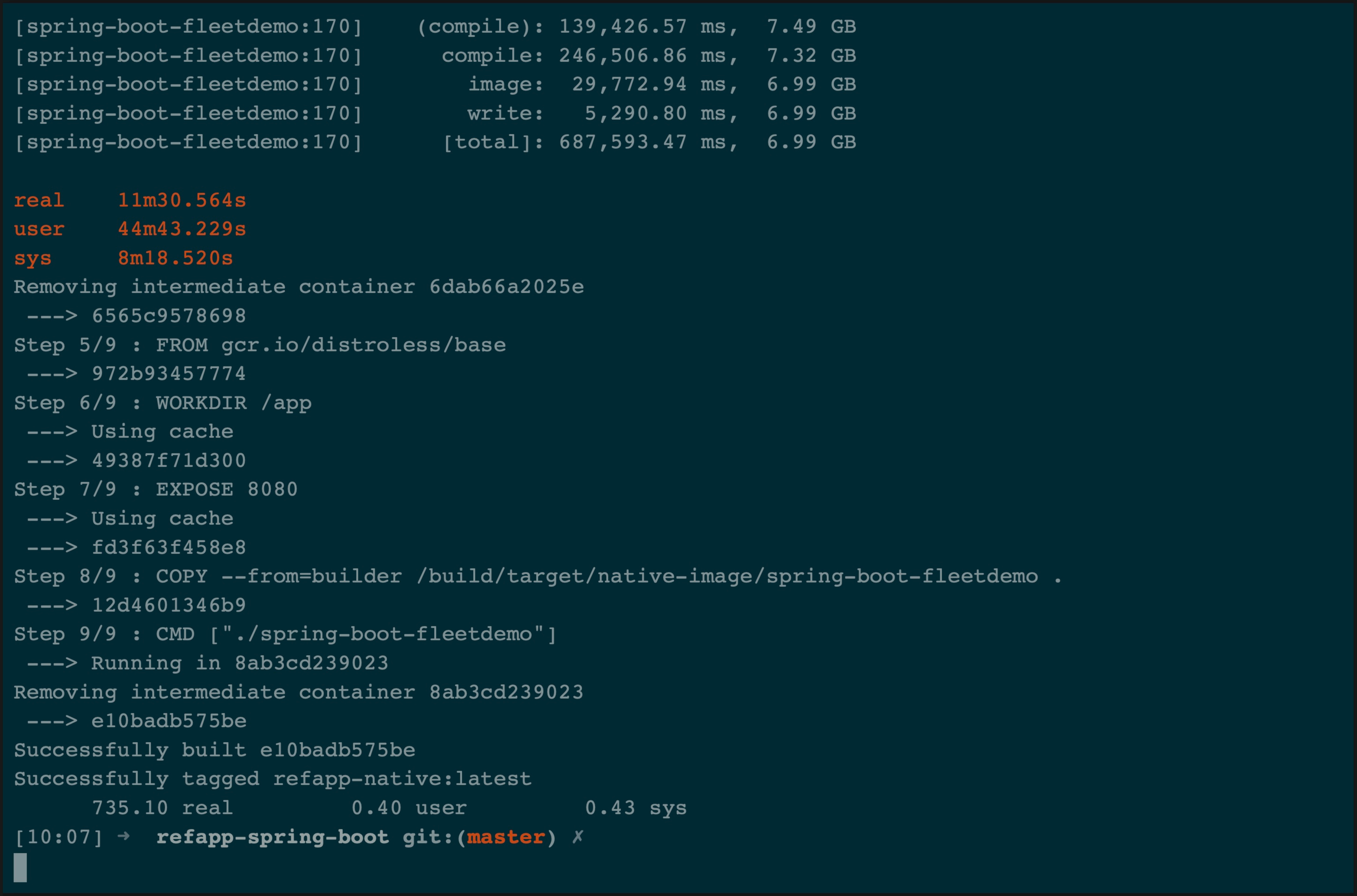
Spring Boot: WebFlux, R2DBC, Reactor Kafka
Go: Fiber, sqlx, Sarama

Road to Enable Native

GraalVM 20.3

1. Upgrade to Spring Boot 2.4
2. Add GraalVM native support. Substrate VM (`svm`)
3. Add Spring native support (`spring-graalvm-native`)
4. Create build script or use maven plugin (build.sh)
5. Declare all Reflections (for DTO beans) and resources
    - Manually or use `native-image-agent` to generate
6. Compile, run and fix remaining stuff (trial and error)
    - Reflection config for Kafka and JSON serializers
    - Resource config for Kafka
    - Substitute Kafka class using Method Handles

Build Native Image (20x speed)



```
[spring-boot-fleetdemo:170]    (compile): 139,426.57 ms,  7.49 GB
[spring-boot-fleetdemo:170]      compile: 246,506.86 ms,  7.32 GB
[spring-boot-fleetdemo:170]        image:  29,772.94 ms,  6.99 GB
[spring-boot-fleetdemo:170]        write:   5,290.80 ms,  6.99 GB
[spring-boot-fleetdemo:170]      [total]: 687,593.47 ms,  6.99 GB

real    11m30.564s
user    44m43.229s
sys     8m18.520s
Removing intermediate container 6dab66a2025e
 ---> 6565c9578698
Step 5/9 : FROM gcr.io/distroless/base
 ---> 972b93457774
Step 6/9 : WORKDIR /app
 ---> Using cache
 ---> 49387f71d300
Step 7/9 : EXPOSE 8080
 ---> Using cache
 ---> fd3f63f458e8
Step 8/9 : COPY --from=builder /build/target/native-image/spring-boot-fleetdemo .
 ---> 12d4601346b9
Step 9/9 : CMD ["./spring-boot-fleetdemo"]
 ---> Running in 8ab3cd239023
Removing intermediate container 8ab3cd239023
 ---> e10badb575be
Successfully built e10badb575be
Successfully tagged refapp-native:latest
      735.10 real         0.40 user         0.43 sys
[10:07] ➜  refapp-spring-boot git:(master) ✗
```

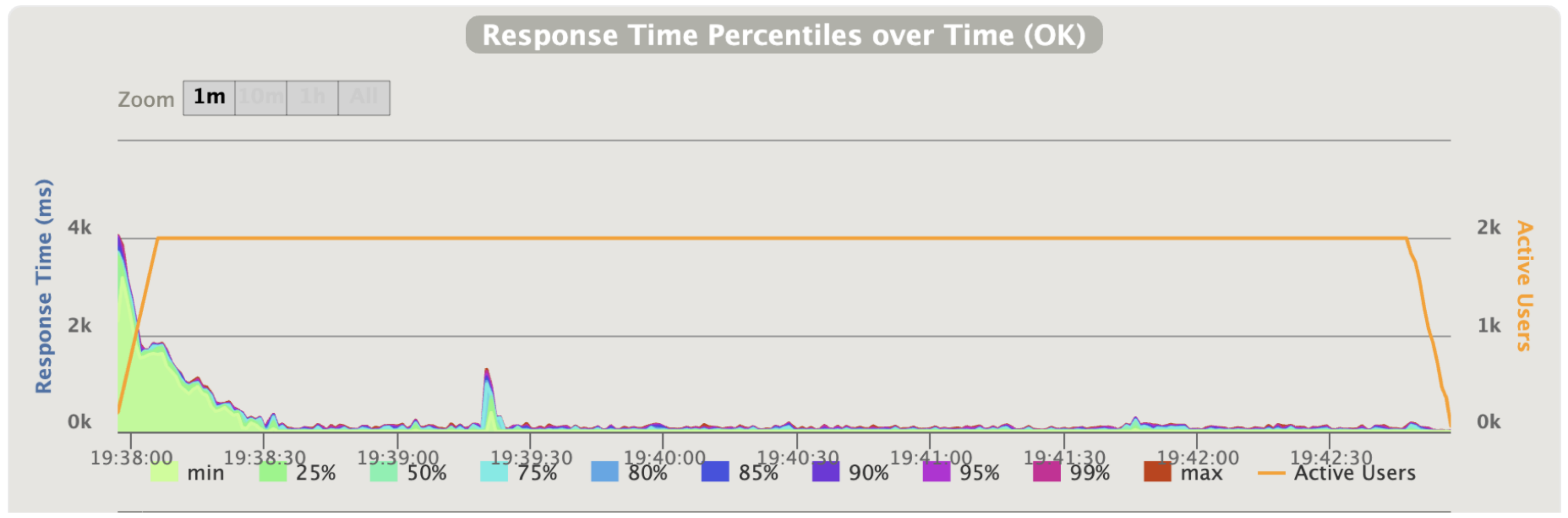
Develop and Build Findings

- Unable to compile static executable and build from a scratch docker-image
  - Both Go and GraalVM native executables depends on shared C/C++ libraries
  - Googles `gcr.io/distroless/base` is used instead
- Even minor changes breaks the build
  - Spring Boot 2.4.0-RC1 to 2.4.0 release update
  - Graal 20.2 to 20.3 minor update
  - Use of new features from existing 3rd party libraries
  - Adding 3rd party libraries
- Discrepancy between dev and runtime environments
- What's the credibility of unit tests

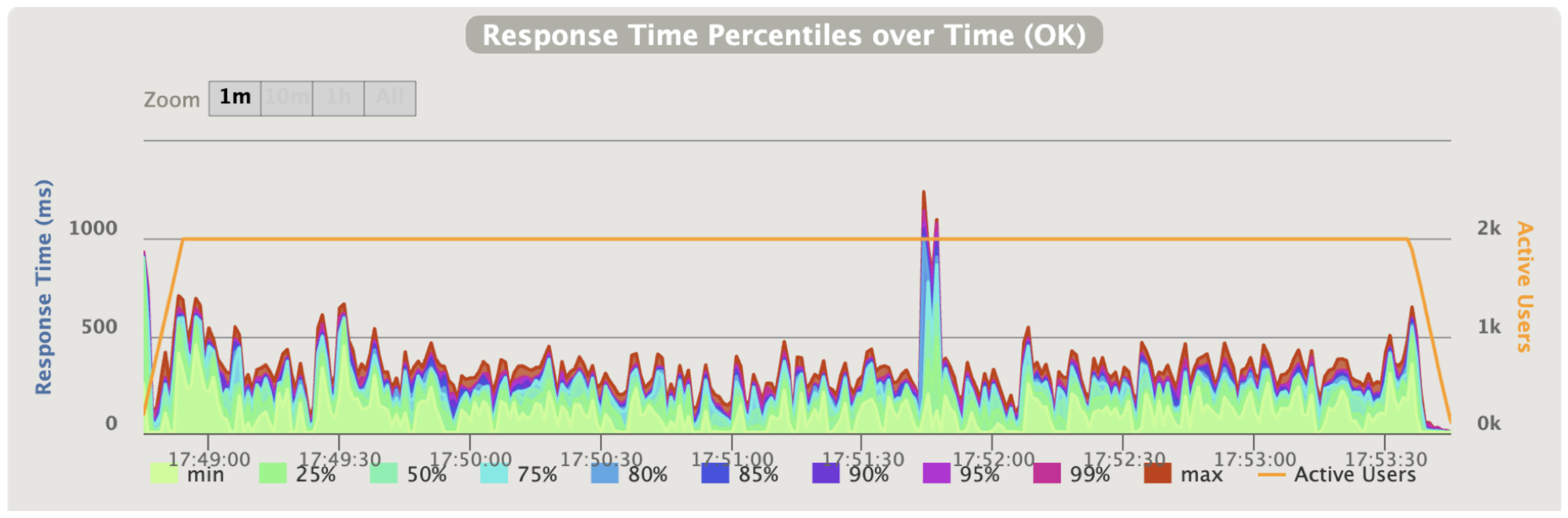
Demo

Load Test 2K req/s during 5 minutes

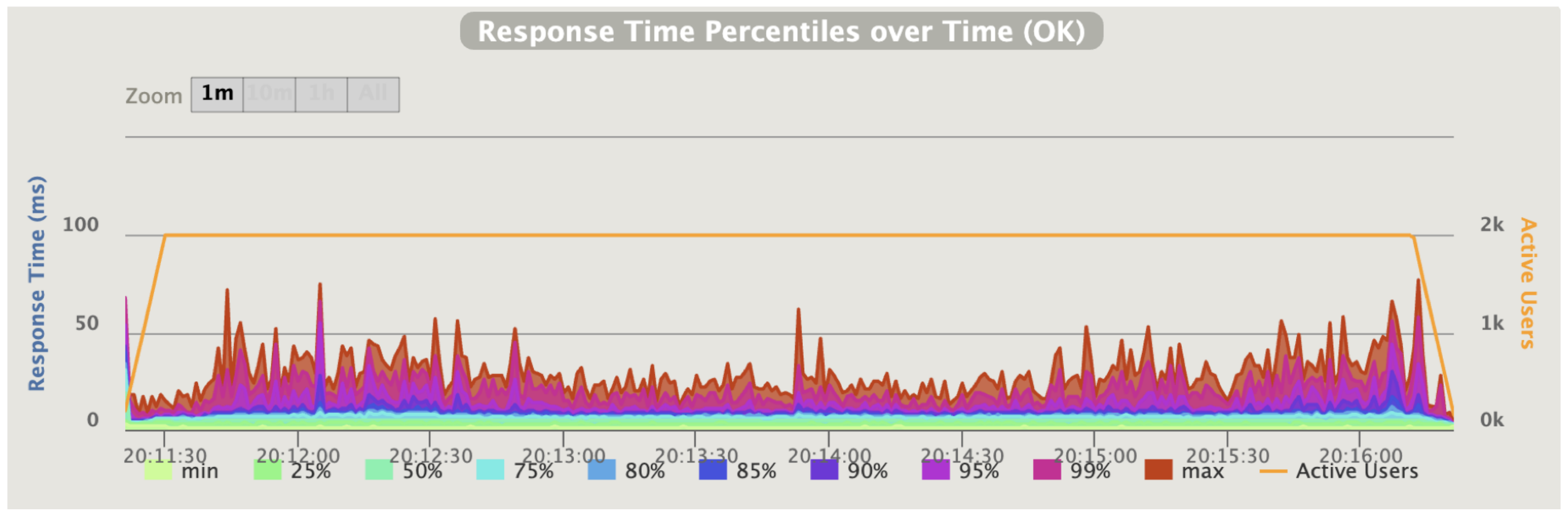
OpenJDK 11



GraalVM Native 20.3



GO 1.14



Load Test Metrics

App	Memory S/E [MB]	TPS	Mean Resp [ms]	Max Resp [ms]	Con. Size [MB]	Startup [ms]
JVM	322/551	1797	74	4048	444	5527
Native	49/643	1628	185	1242	175	150
Go	3/107	1920	6	77	31	10

2000 clients reports approx. one msg per second for 5 minutes
(cold start, ephemeral micro-service)

Runtime Findings

- JVM performs better than Native (throughput and latency)
  - Significantly better performance for warm JVM compared to Native
- Higher memory consumption for Native compared to JVM (mx256m)
- Go is a magnitude better on almost everything

GraalVM to the Rescue!

- Increase application throughput and reduced latency
- Compile applications into small self-contained native binaries

Recommendation

- A lot of effort is put into GraalVM Native, and it should be on your tech radar
- If startup-time is crucial and for greenfield JVM micro-services GraalVM Native might be of interest
  - Long startup times can also be mitigated in the execution platform
  - Though, a more appropriate language such as Go definitely is an option
- GraalVM Native is of no interest for legacy JVM services without framework support

Dear fellow JVM'ers!

"There's no Holy Graal, just loads of hard work and Java."

- Me