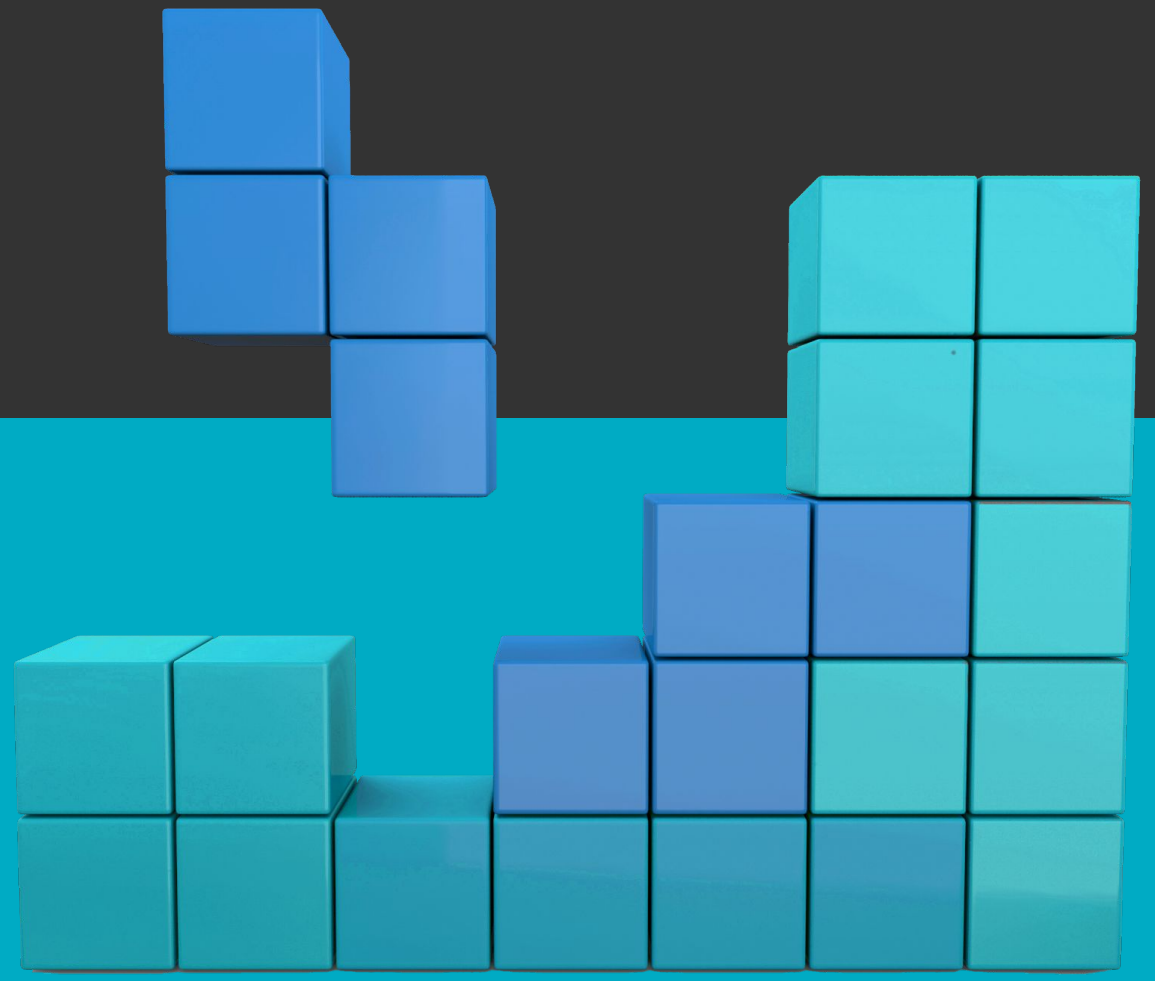
BELLSOFT



# Don't put all your eggs in one buildpack

Dmitry Chuyko

2022

# Who we are


## Dmitry Chuyko

BELLSOFT

Liberica www.bell-sw.com
supported OpenJDK binaries

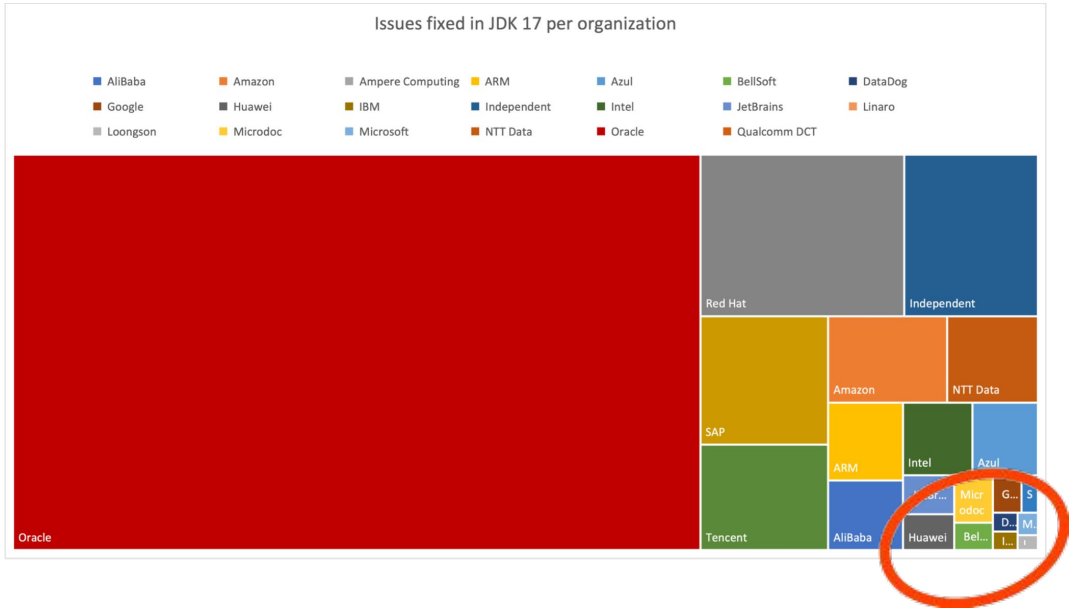
ex-employers:

ORACLE

@dchuyko

# OpenJDK Contributions

LTS


BELLSOFT

# Start a project

Business

# Go to prod

Dev DevOps CloudOps

# Go to market




Business

# Cloud cost optimization Zen

- Business
  - Profits
  - Up and running services
- CloudOps
  - Run services
  - No difference if it is Java or JavaScript in a container
  - In Cloud algorithms they trust

- Developers
  - Craft services

BELLSOFT

# Java becomes more container-aware

There are great build tools helping to create containers that don't break instantly. Monitoring helps to catch obvious problems. But technically, it's not enough. You also need to arrange the right development process. Otherwise, costs will continue to rise and everyone will be unhappy.

# Important information is not used

Hardware resources are used inefficiently. (See how)

It is not clear what the goals and efficiency metrics are.

# Efficiency

Not just reliability / SRE

**SLA**

SERVICE LEVEL AGREEMENT

The agreement you make with your users or clients

**SLOs**

SERVICE LEVEL OBJECTIVES

The objectives your team must reach
to meet the agreement

**SLIs**

SERVICE LEVEL INDICATORS

Your real performance numbers

# Solution for a service

- Define SLA
- Define SLOs
- **Involve developers**
- Determine best performance levels
- Construct pod configuration to reach the best price-performance match
  - JVM tuning
  - Full utilization of at least one resource
- Observe and capture Prod behavior
  - SLIs
  - Load profiles
- Tune again in Dev
  - Or redesign

# Expectations & tracking

# Service Level Objectives

- Common SLOs aligned with business requirements
- Per service SLO can redefine common SLOs
- Metrics – not all metrics are SLIs
  - Uptime
  - QoS, priorities
  - Time to response
  - Time to performance
  - Peak-performance
  - Response time / Latency

# JVM & SLOs

- **Peak performance (ops/s, data/s etc.)**
  - Effective code compilation
  - Garbage collection
- **Response time (latency)**
  - From the start / after warm up
  - Microservice architecture patterns
  - Garbage collection
  - Effective code compilation
  - Safepoints

# 2 goals at once

- **Meet SLOs**
  - It can require more resources
- **Reduce cloud bills**
  - It can block the achievement of SLO goals and break SLAs

# Can do both! :-)

- **Spend a lot of resources**
- **Break SLAs**

# Underutilization

- “A majority of Kubernetes workloads are underutilizing CPU and memory”
- **“**49 percent of containers are using less than 30 percent of their requested CPU, and 74 percent of containers are using under 80 percent.”
- “1 in 3 AWS container environments runs Fargate.”
- “On average, Kubernetes organizations run 16 pods per host, while organizations using Amazon Elastic Container Service (ECS) run 5 tasks per host.”

— Datadog Container reports [1, 2]

# Underutilization

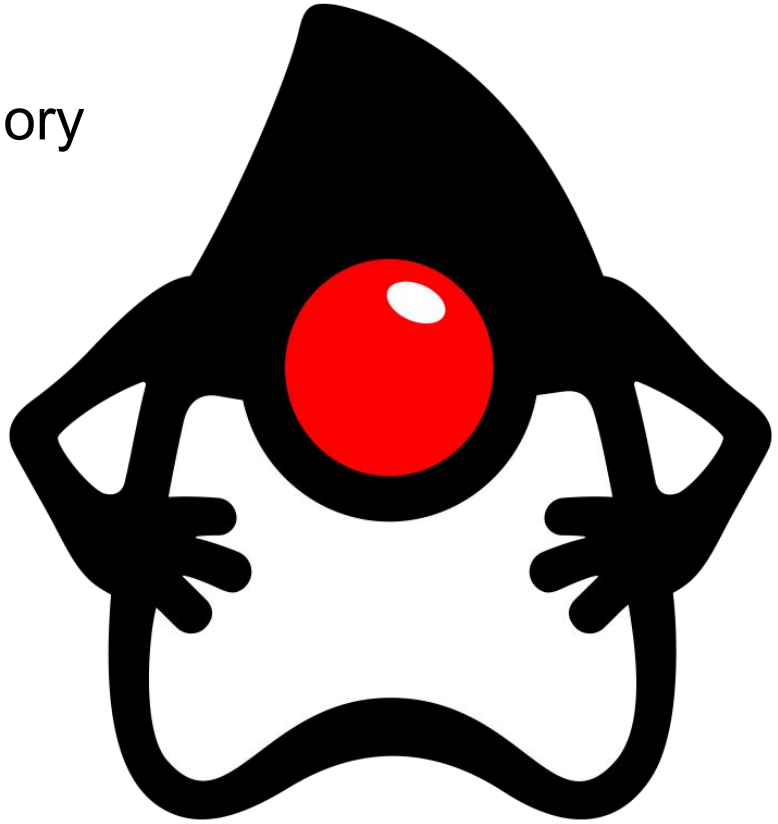
- Metric sources
  - Instances monitoring
  - Clusters monitoring
  - APMs
- Memory
- CPU
- I/O
- Limits that are lower than node capacity (for Guaranteed QoS)
- Restarting an underused instance is strange

# Container

> ***...is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.***

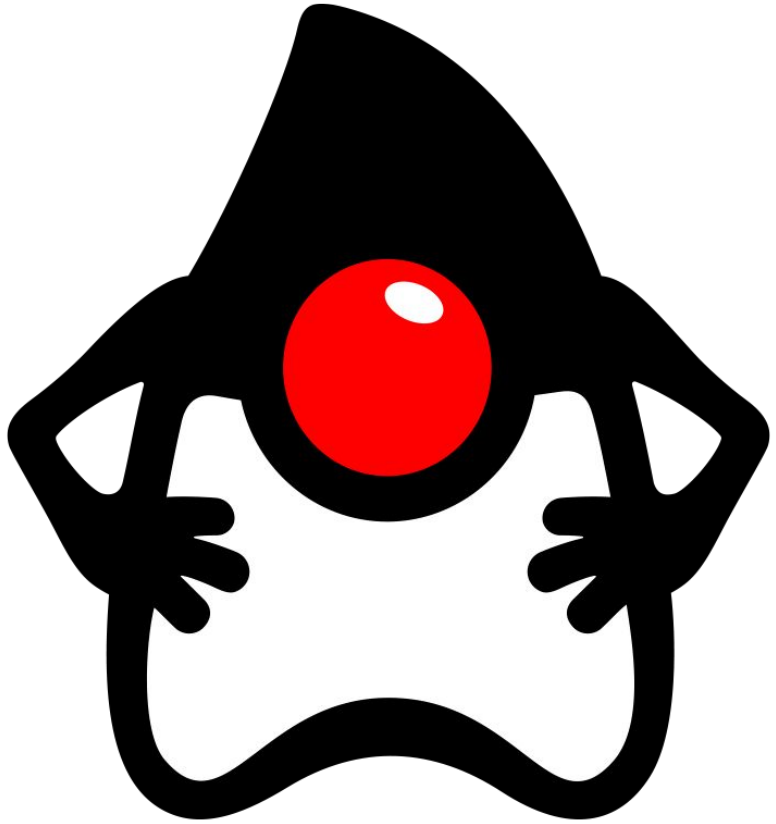
# HotSpot JVM in containers

- Sees host resources (CPU, memory)
- Detects container resources: CPU quotas and cores, memory
  - Interprets CPU information at start time
  - Treats [container] environment as a [smaller] machine
    - Xmx=RAM/4, constant CPUs
- JDK 11+ collects and uses cgroups v1 information
- JDK 17+ and 11u collect and use cgroups v2 information
- Has `-XX` flags to be adaptive/consistent in sizing
  - `UseContainerSupport`
  - `[Min|Max|Initial]RamPercentage`
  - `ActiveProcessorCount`
  - Flags and their ergonomics evolve
    - JDK 19, 18u Do not use CPU Shares to compute active processor count



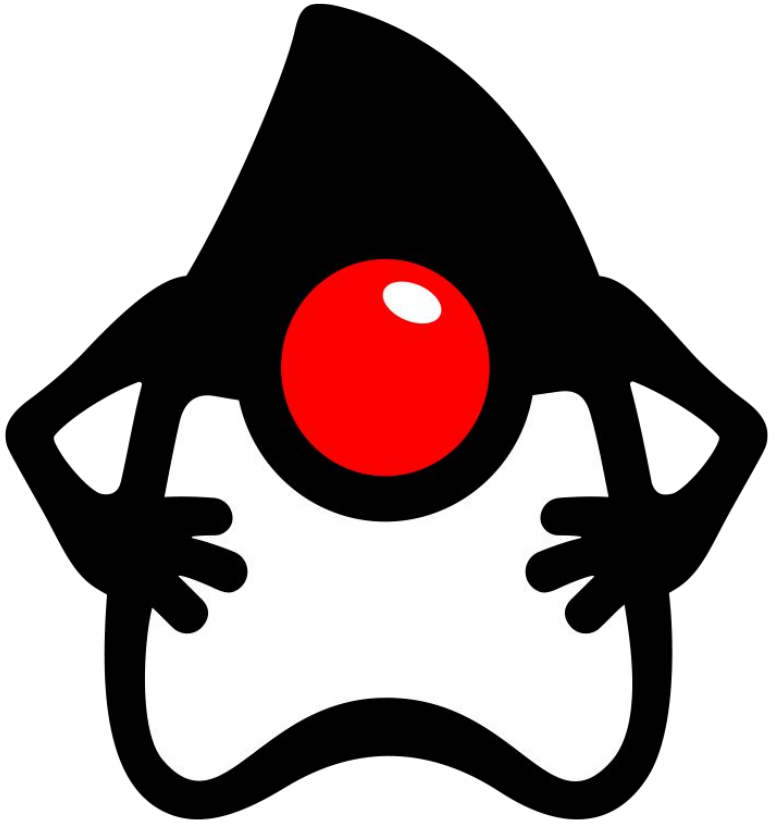
# HotSpot JVM & low resources

- Falls back to SerialGC
  - If `NeverActAsServerClassMachine`
  - Unless `AlwaysActAsServerClassMachine`
  - If less than 2 processors are available (e.g. 1 CPU)
  - If available memory is less than 1792 MB
- Stays multi-threaded
  - E.g. there are always at least 2 JIT compiler threads



# Java applications in containers

- Determine heap and native memory requirements
  - GC logs
  - Native memory tracking
- Some resources may work differently
  - tmp, random, shared memory
- The whole container is yours



# Kubernetes

> *...provides you with a framework to run distributed systems resiliently. It takes care of scaling and failover for your application, provides deployment patterns, and more.*

# JVM container in K8s

- Container image is a part of the application for configurable deployment
  - Requests
  - Limits
  - Environment variables can be used to pass extra information
  - Management system args
- Image is deployed on a pod
  - Pods and service code run on a node
  - Nodes have some actual resources

# JVM container in K8s

- CPU & memory requests are used to place a pod
- CPU limit is used to balance pods at their run time
  - JVM tunes thread pool once
- Memory limit
  - Used by the JVM to calculate MaxHeapSize
  - Controlled by host to kill a pod
  - ‘Xmx >= memory limit’ is a “way to success”
    - It is not checked by the JVM, so the app may work until the heap it too big
    - ‘Xmx < memory limit’ is not a guarantee
- Pods are monitored
- JVM processes are node processes
- JVM flags are to be tuned


SCALE THE PODS !!!

# Not all services are the same

- **Less critical**
  - Soft SLOs
  - Not elastic
  - Mutually elastic
  - Overcommit
    - Return GC memory to OS
    - Liberica Lite
  - Vertical scaling
  - “Burstable” QoS
    - Restart
- **Highly critical**
  - Strict SLOs
  - Highly elastic
    - Design for handling exponential growth
    - Horizontal scaling
    - "Guaranteed" QoS
  - Step scaling
  - Pre-scaling (time-based auto scaling)

# Scaling is simple but may be costly

- Do the right sizing
- Scale down
- Amazon: moving from Fargate to EKS with
  - Spot instances up to 70%
  - Standard instances up to 20%
- Spot is not for Black Fridays
- Long-term purchase

- Pod metrics
  - Most typical are requests
  - Baseline is necessary
  - Alert to both high and low CPU
- Object metrics (~total)

# Dependencies first

- Know, measure, and scale dependencies
- Database, service, or other external data source
- Prevent cascading failures
- Disk I/O bandwidth and latency
- Network I/O bandwidth and latency
- Calculation complexity
- Concurrency
- E.g. either SQL or NoSQL can be as slow as 100 RPS
- Multiplexing

# Pods on nodes in 2022

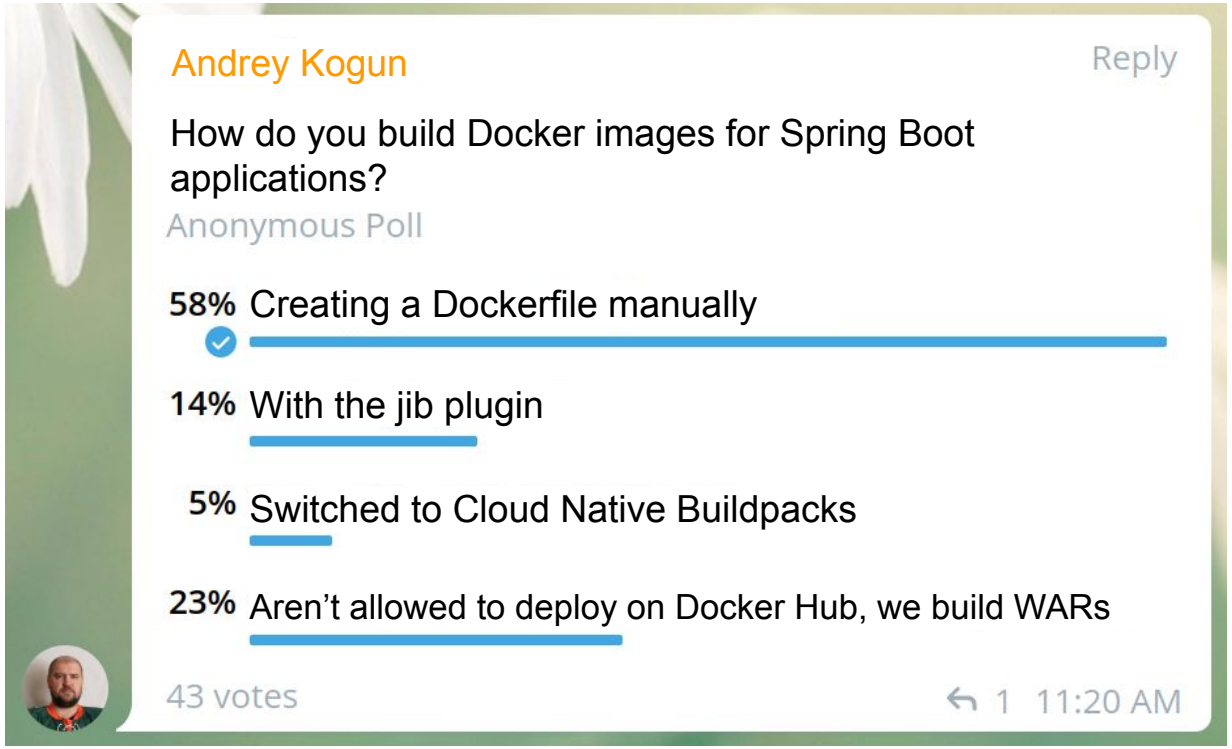

# Spring Boot Petclinic JPA

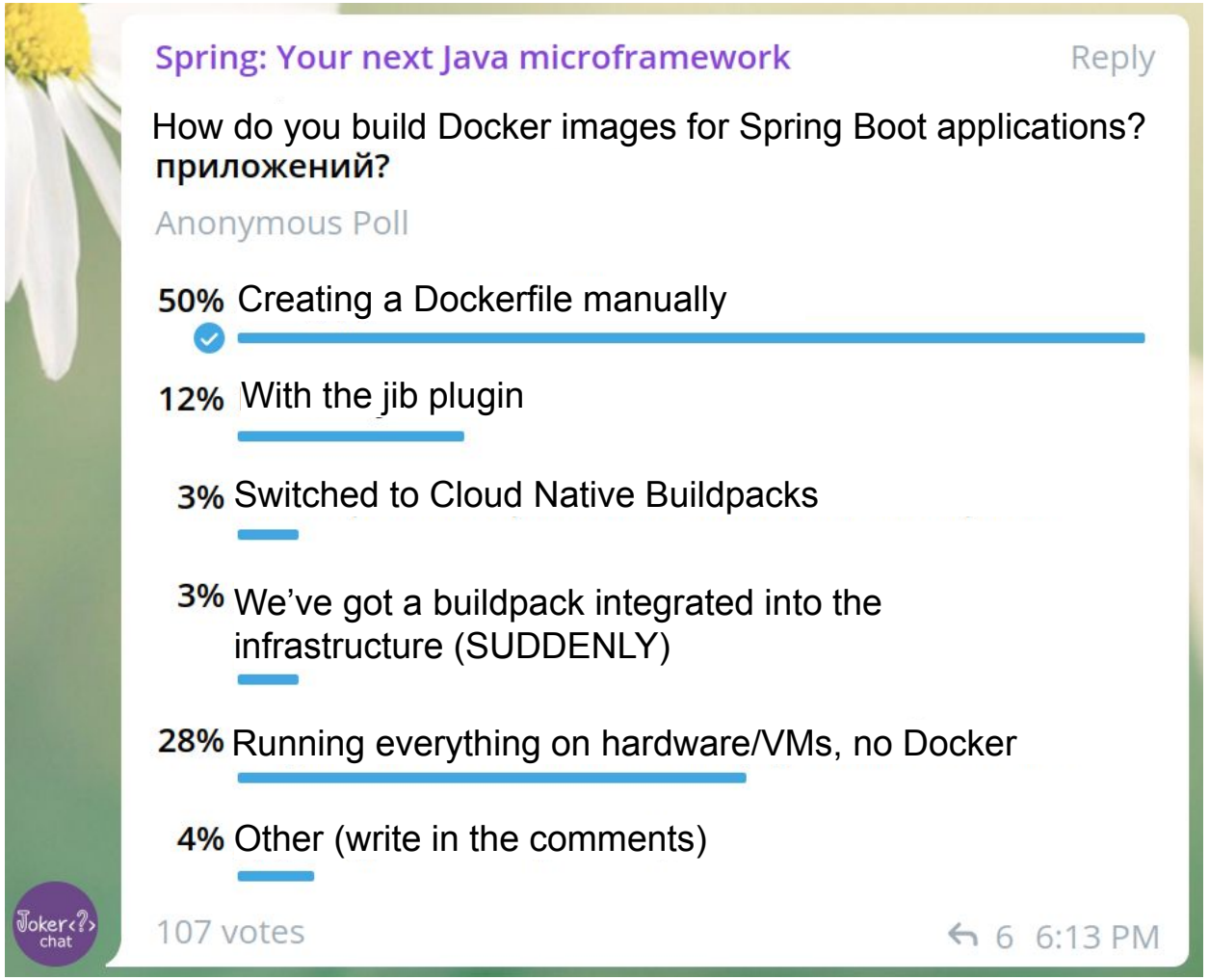
- spring-native/tree/main/samples/petclinic-jpa
- HSQLDB
- Spring Native 0.12.1-**SNAPSHOT**

- Fat jar **40 MB**
- Native image **124 MB**

# Developer Voice

- Aleksey Nesterov. Spring: Your next Java microframework
- Vladimir Plizga. Spring Boot "fat" JAR: Thin parts of a thick artifact



# Can I use some magic instead?

- Cloud specific magic
- Azure az containerapp
  - `az containerapp up --name my-app --source ./my-app`
    - Takes `Dockerfile` – not so magical!
  - `az containerapp up --repo https://github.com/my-name/my-app`
  - Creates a resource group, Container Registry, Log Analytics workspace, Container Apps environment, Container App, ingress
  - Builds, pushes, and runs the container image in the cloud
- AWS App2Container
  - Analyzes any running Java application
  - Matches the environment
    - Base image most similar to OS
  - Wraps as a container

# App2Container

```
$ sudo app2container init
...
All application artifacts will be created under /root/app2container. Please ensure that th
folder permissions are secure.
...

$ sudo app2container inventory
{
        "java-generic-ecda35dd": {
                "processId": 40974,
                "cmdline": "/home/bellsoft/jdk-17.0.3.1/bin/java -jar
petclinic-jpa-0.0.1-SNAPSHOT.jar ",
                "applicationType": "java-generic",
                "webApp": ""
        }
}
```

# App2Container

```
$ sudo app2container analyze --application-id java-generic-0ef0c798
analysis.json
"env": {
...
        "JDK_JAVA_OPTIONS": "",
...
       },

$ sudo app2container containerize --application-id java-generic-ecda35dd
```

ECR

# How about good old Dockerfile?

```
FROM bellsoft/liberica-openjdk-alpine-musl:17.0.3.1-2

RUN addgroup -S spring && adduser -S spring -G spring

USER spring:spring

VOLUME /tmp

ADD petclinic-jpa-0.0.1-SNAPSHOT.jar /app/app.jar

EXPOSE 8080

ENV JAVA_TOOL_OPTIONS="-Djava.security.egd=file:/dev/./urandom
-XX:MaxRAMPercentage=80.0"

ENTRYPOINT exec java $JAVA_TOOL_OPTIONS -jar /app/app.jar
```

# Automation instead of magic

- Paketo Buildpacks
  - Transform your application source code into container images
  - Open source production-ready buildpacks for the most popular languages and frameworks
  - Cloud Foundry Project
  - `pack build samples/java --path java/maven`
    - Get everything that is necessary
    - By default, the Paketo Java buildpack will use the Liberica JVM
  - Special Spring Boot buildpack

# Maven way. Settings (parent POM)

```
<!-- `tiny` builder allows small footprint and reduced surface attack, you can also use
`base` (the default) or `full` builders to have more tooling available in the image for an
improved developer experience -->

<builder>paketobuildpacks/builder:tiny</builder>

<java.version>17</java.version>

...

                                <BP_NATIVE_IMAGE>true</BP_NATIVE_IMAGE>
```

# Maven way. Build

```
$ mvn spring-boot:build-image

Version info: 'GraalVM 22.1.0 Java 11 CE'

[INFO]     [creator]     Paketo BellSoft Liberica Buildpack 9.3.6
[INFO]     [creator]       https://github.com/paketo-buildpacks/bellsoft-liberica
[INFO]     [creator]       Build Configuration:
[INFO]     [creator]         $BP_JVM_TYPE                 JRE             the JVM type - JDK or JRE
[INFO]     [creator]         $BP_JVM_VERSION              11.*            the Java version
[INFO]     [creator]       Launch Configuration:
[INFO]     [creator]         $BPL_DEBUG_ENABLED           false           enables Java remote debugging support
[INFO]     [creator]         $BPL_DEBUG_PORT              8000            configure the remote debugging port
[INFO]     [creator]         $BPL_DEBUG_SUSPEND           false           configure whether to suspend execution until a
debugger has attached
[INFO]     [creator]         $BPL_HEAP_DUMP_PATH                          write heap dumps on error to this path
[INFO]     [creator]         $BPL_JAVA_NMT_ENABLED        true            enables Java Native Memory Tracking (NMT)
[INFO]     [creator]         $BPL_JAVA_NMT_LEVEL          summary         configure level of NMT, summary or detail
[INFO]     [creator]         $BPL_JFR_ARGS                                configure custom Java Flight Recording (JFR)
arguments
[INFO]     [creator]         $BPL_JFR_ENABLED             false           enables Java Flight Recording (JFR)
[INFO]     [creator]         $BPL_JMX_ENABLED             false           enables Java Management Extensions (JMX)
[INFO]     [creator]         $BPL_JMX_PORT                5000            configure the JMX port
[INFO]     [creator]         $BPL_JVM_HEAD_ROOM           0               the headroom in memory calculation
[INFO]     [creator]         $BPL_JVM_LOADED_CLASS_COUNT  35% of classes  the number of loaded classes in memory calculation
[INFO]     [creator]         $BPL_JVM_THREAD_COUNT        250             the number of threads in memory calculation
[INFO]     [creator]         $JAVA_TOOL_OPTIONS                           the JVM launch flags
```

# Maven way. Build

```
[INFO]     [creator]         Using Java version 11.* from BP_JVM_VERSION
[INFO]     [creator]       BellSoft Liberica NIK 11.0.15: Reusing cached layer
OR
[INFO]     [creator]         Using Java version 17.* from BP_JVM_VERSION
[INFO]     [creator]       BellSoft Liberica NIK 17.0.3: Contributing to layer
....
[INFO]     [creator]       Launch Configuration:
[INFO]     [creator]         $BPL_SPRING_CLOUD_BINDINGS_DISABLED  false  whether to auto-configure Spring
properties from bindings
```

2

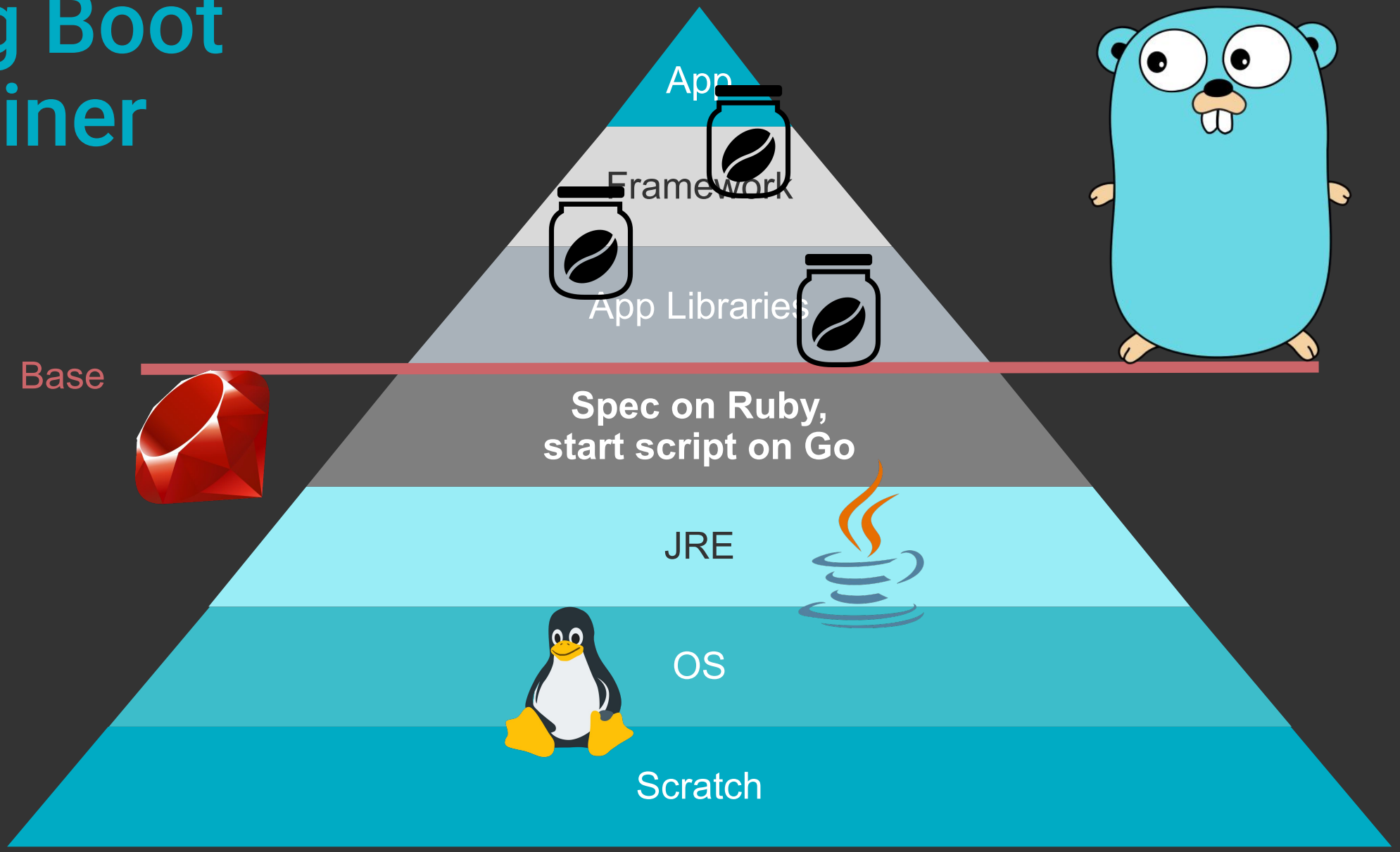
# Builders use stacks

- A stack consists of two images
  - Build image: the environment in which your app is built
  - Run image: the OS layer for your app image
- Stacks are rebuilt whenever a package is patched to fix a CVE + weekly
- Tiny builder uses the Paketo Tiny Stack
  - Build image: bionic with buildpacks for Java, Java Native Image, Go, and Procfile
  - Run image: distroless-like bionic + glibc + openssl + CA certs
- Base builder
  - Run image: ubuntu:bionic + openssl + CA certs

# Start a container

```
Setting Active Processor Count to 96

Calculated JVM Memory Configuration: -XX:MaxDirectMemorySize=10M -Xmx16161921K
-XX:MaxMetaspaceSize=103294K -XX:ReservedCodeCacheSize=240M -Xss1M (Total Memory: 16G, Thread Count: 250,
Loaded Class Count: 15823, Headroom: 0%)

Enabling Java Native Memory Tracking

Adding 127 container CA certificates to JVM truststore

Spring Cloud Bindings Enabled

Picked up JAVA_TOOL_OPTIONS:
-Djava.security.properties=/layers/paketo-buildpacks_bellsoft-liberica/java-security-properties/java-secur
ity.properties -XX:+ExitOnOutOfMemoryError -XX:ActiveProcessorCount=96 -XX:MaxDirectMemorySize=10M
-Xmx16161921K -XX:MaxMetaspaceSize=103294K -XX:ReservedCodeCacheSize=240M -Xss1M
-XX:+UnlockDiagnosticVMOptions -XX:NativeMemoryTracking=summary -XX:+PrintNMTStatistics
-Dorg.springframework.cloud.bindings.boot.enable=true
```

# Spring Boot Container



# What we can tweak

- `BPL_JVM_THREAD_COUNT`
  - 250 (because of tomcat.max-threads)
  - 50 if the application is a reactive web application
- `BPL_JVM_HEAD_ROOM`
- `BPL_JVM_LOADED_CLASS_COUNT`

- `JAVA_TOOL_OPTIONS`
  - Append mode
  - ActiveProcessorCount=0 to restore the default
  - Active Processor Count uses total CPUs
    - It also used to be detected like 4 (fixed) which could cause approx. 2k → 0.5k rps drop

# Resulting images

| REPOSITORY | TAG | SIZE |
|---|---|---|
| bellsoft/liberica-openjdk-alpine-musl | 17.0.3.1-2 | 99.9MB |
| petclinic-jpa-alpine | 0.0.1-SNAPSHOT | 142MB |
| paketobuildpacks/run | tiny-cnb | 17.4MB |
| paketobuildpacks/builder | tiny | 578MB |
| petclinic-jpa-ni | 0.0.1-SNAPSHOT | 153MB |
| petclinic-jpa-jvm | 0.0.1-SNAPSHOT | 229MB |

# Optimize application for containers?

# Optimize containers instead!

# Petclinic Load

- WRK2 + HdrHistogram
- Fixed rate
- Query all default pet owners
  - Low GC pressure
- Max Rate detection
- Warmup study
- Latency measurement
- 10 cores for `wrk`, 100 connections

- Intel(R) Xeon(R) Platinum 8259CL
- 1x96x1
- 88 cores for the cluster
- Target millisecond latencies

```
$ minikube start --driver=docker --extra-config=kubelet.housekeeping-interval=10s \
--cpus 88 --memory 32768

$ taskset -c 88-95 ./wrk -t10 -c100 -d30s -R1100 --latency \
http://192.168.49.2/owners?lastName=
```

# Check target instance variants

- Generic cloud provider list, e.g. EC2
- Horizontal autoscaler limited subset, e.g. for Fargate

| vCPU | Memory |
|---|---|
| .25 vCPU | 0.5 GB, 1 GB, 2 GB |
| .5 vCPU | 1 GB, 2 GB, 3 GB, 4 GB |
| 1 vCPU | 2 GB, 3 GB, 4 GB, 5 GB, 6 GB, 7 GB, 8 GB |
| 2 vCPU | Between 4 GB and 16 GB in 1-GB increments |
| 4 vCPU | Between 8 GB and 30 GB in 1-GB increments |

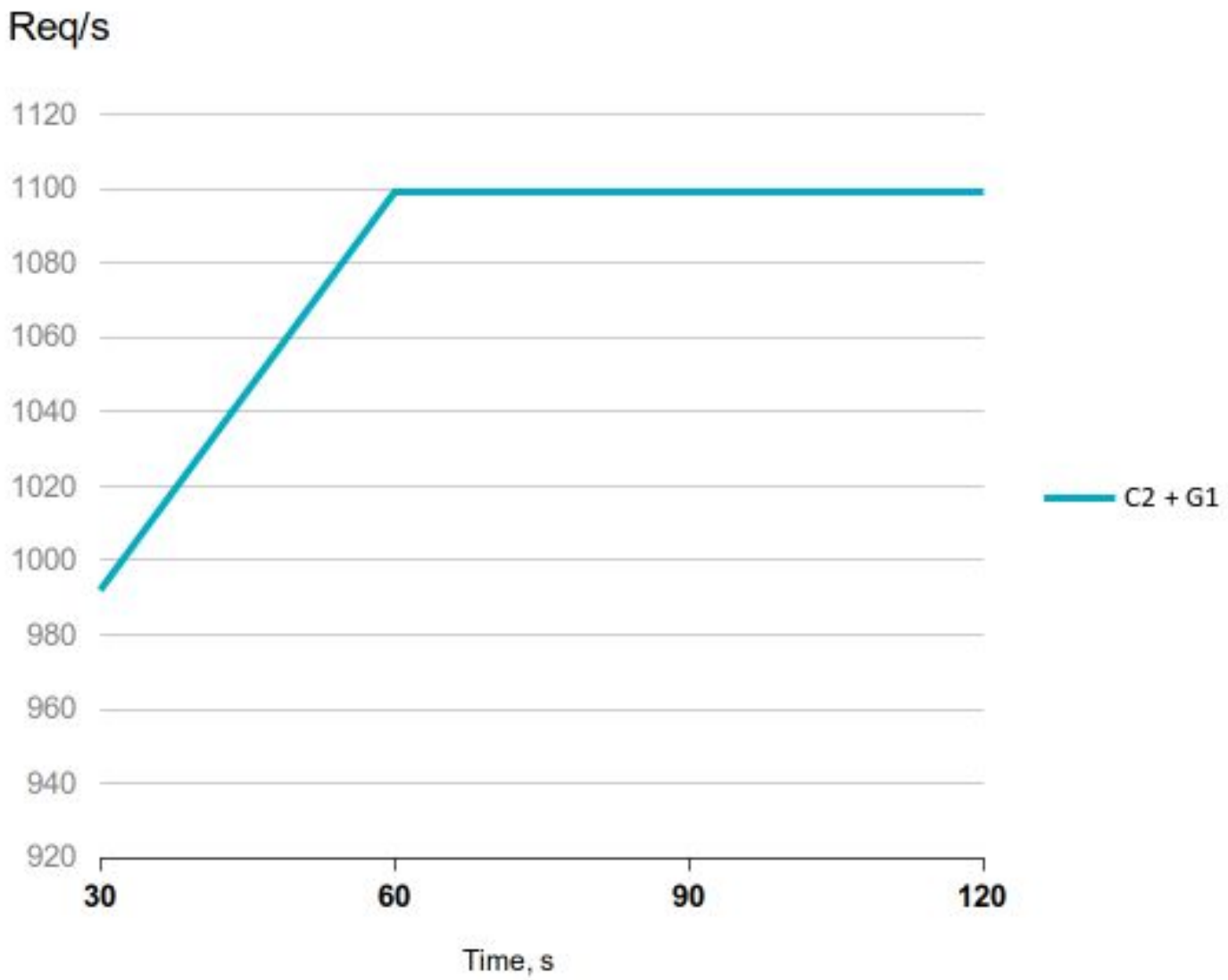
8 CPUs
Xmx16g

# Scalability on bare metal

- JVM buildpack container: 1113 RPS
- Native image buildpack container: 2462 RPS
- The app doesn’t scale vertically anymore
  - Can we reproduce the result in a cluster?
  - Can we scale it horizontally?
  - No goal to optimize the app itself
  - Not all apps are like this

# Warmup

Petclinic. 1200 rps

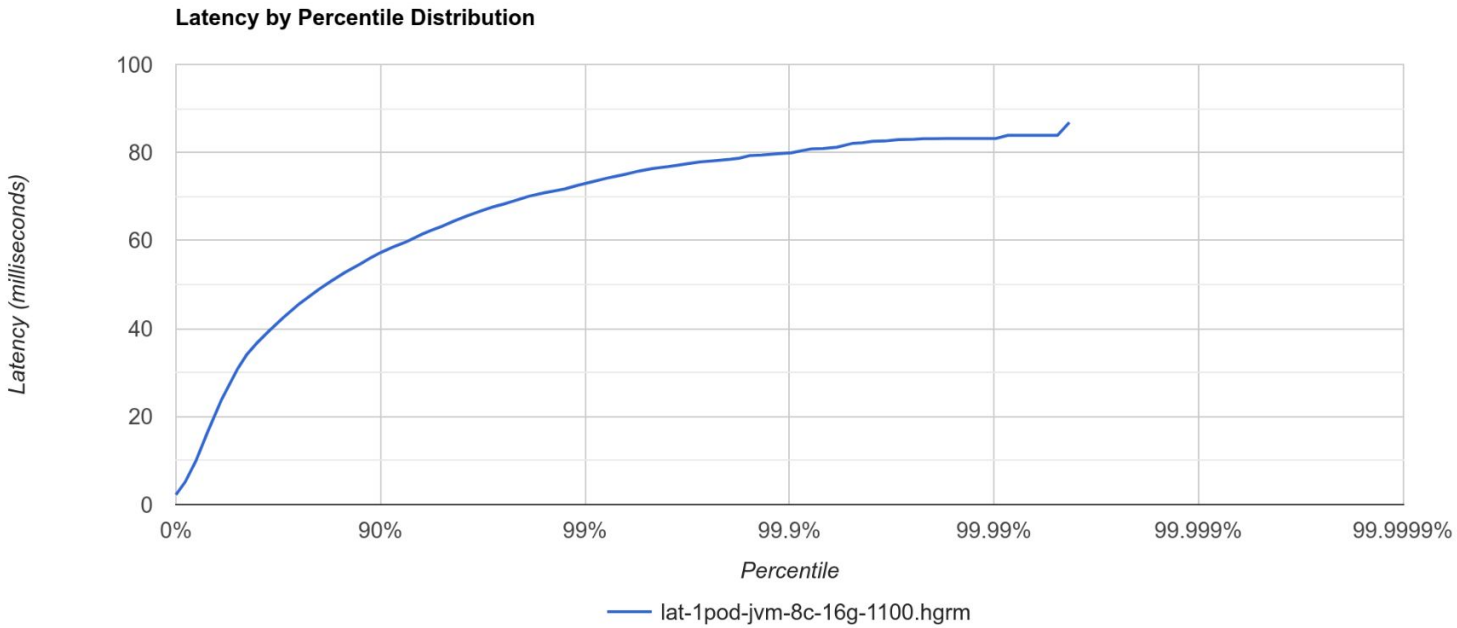


# Latency

Petclinic

**1200 rps**

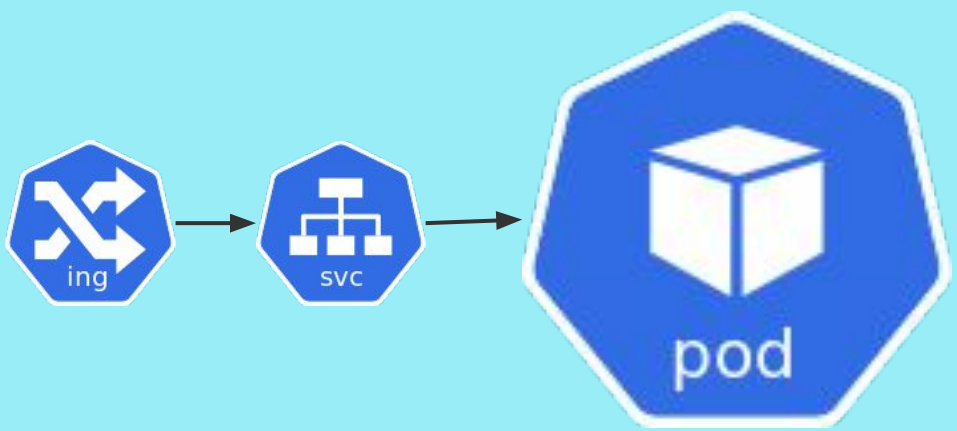
**After 120 s**



8 CPUs
16GB (no swap)

# Scalability in Docker

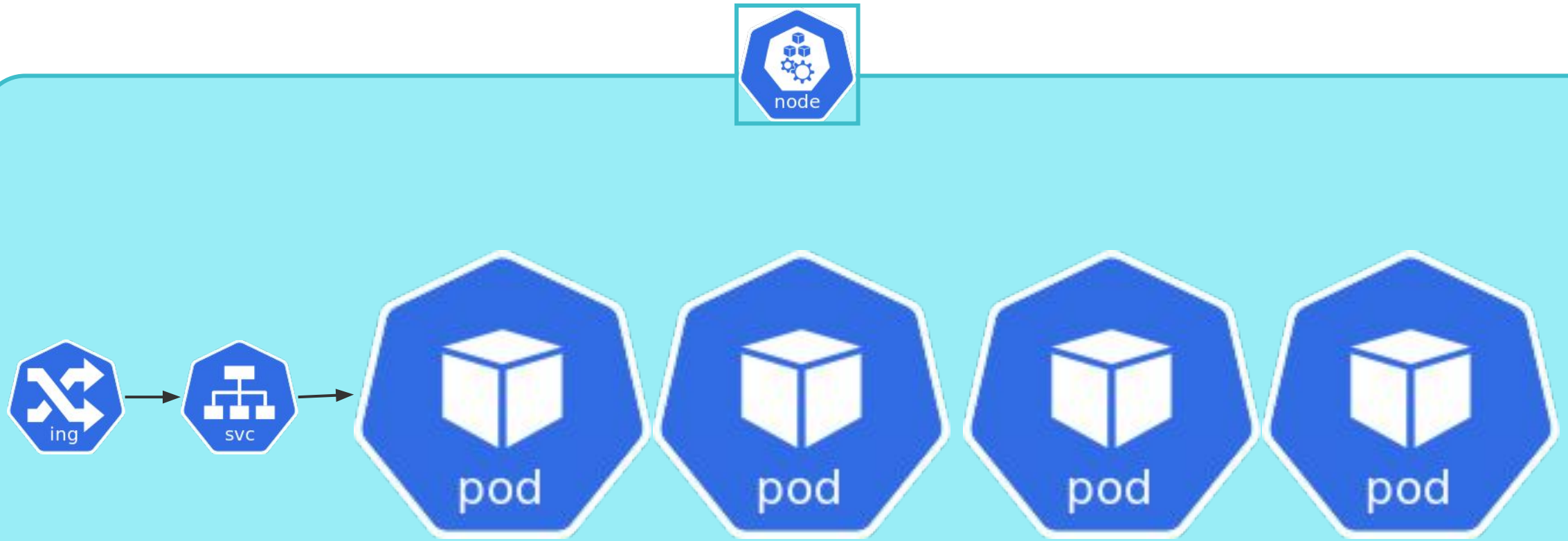
- JVM buildpack container: 1099 RPS




8 CPUs
16Gi

# Scalability in K8s

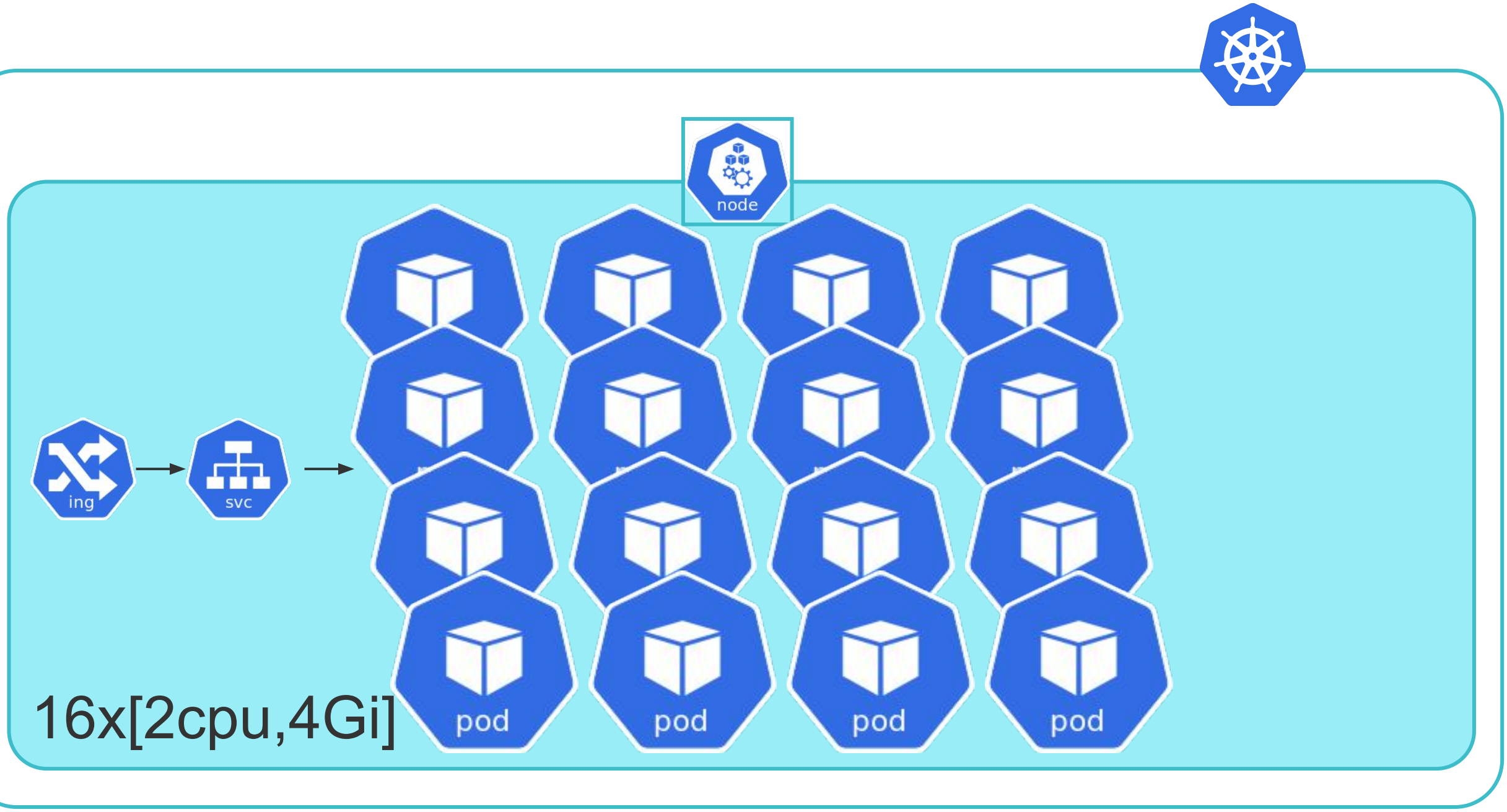
- 1x[8cpu,16Gi] JVM buildpack container: 599 RPS
  - Added ActiveProcessorCount=8
  - 2x worse than 1 bare metal instance
- Same without ingress




4x[8cpu,16Gi]

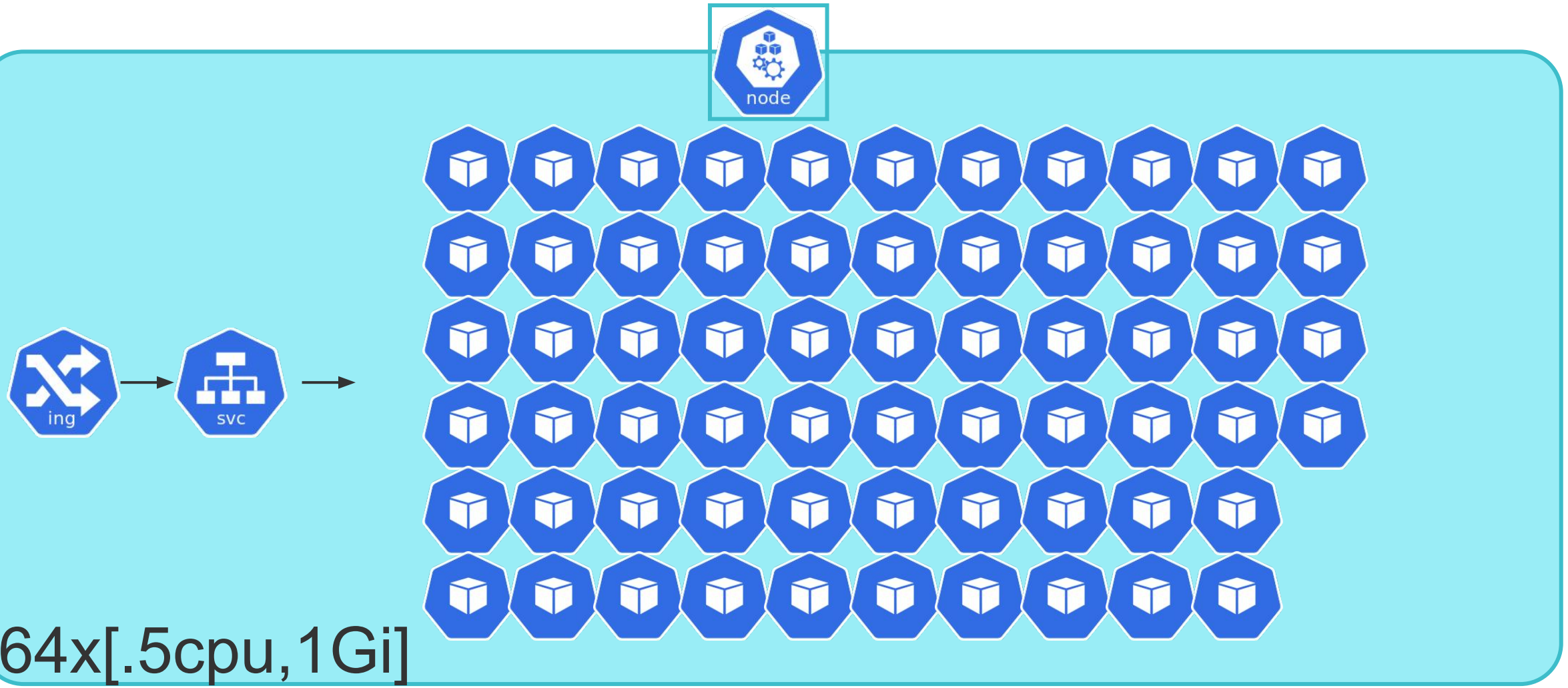
# Scalability in K8s

- 4x[8cpu,16Gi] JVM buildpack containers: 1100 RPS
  - Increased resource consumption but within the same node size
  - Matched bare metal peak performance
  - Matched bare metal warmup
  - Each Java process consumes ~10-70% CPU after warmup
    - Host `top`
    - `kubectl top pod`


node
ing
svc
pod
pod
pod
pod
16x[2cpu,4Gi]

# Scalability in K8s. Best of JVM

- 16x[2cpu,4Gi] JVM buildpack containers: 2200 RPS (!)
  - Added ActiveProcessorCount=2
  - 2x better peak performance than with 1 bare metal instance
  - Each Java process consumes ~200% CPU during warmup
  - Each Java process consumes ~10-70% CPU after warmup
  - It’s still G1
- 16x[2cpu,4Gi] JVM Alpine Dockerfile containers are the same


node
ing
svc
64x[.5cpu,1Gi]

# Scalability in K8s

- 64x[.5cpu,1Gi] JVM buildpack container: 2200 RPS
  - Added ActiveProcessorCount=1
  - 2x better peak performance than with 1 bare metal instance
  - Slooow startup (30s instead of 3s), slooow termination
  - Each Java process consumes ~10-70% CPU after warmup
  - It’s SerialGC
- All improvements are non-linear

# Scalability in K8s. Back to native

- 64x[.5cpu,1Gi] Native image buildpack containers: **5000** RPS
  - 2x better peak performance than with 1 bare metal instance
  - Still instant app startup but slow pod start (30s)
  - Each app process consumes ~20-25% CPU after warmup
- 128x[.25cpu,500Mii] Native image buildpack containers: 3200 RPS
  - Still instant app startup but slow pod start (60s)
- All improvements are non-linear


JVMs and Native Images
they scale in K8s

# Find optimal service scaling strategy

- **Dev complete is not production ready**
- **Load testing, A/B, longevity testing, initial capacity planning**
- **Peak rates and latency detection**
  - Systems will fail, but try to prevent it
  - Peak rate implies high resource utilization
  - Peak rate defines scaling by request count – per service
  - Optimization
- **Instance family selection**
- **CPU and RAM requirements detection**
  - Determined per service
- **Mock services**

# Find CPU and RAM requirements

- Tune and measure at peak performance
- Determine lowest requirements to meet SLOs
- Use stress load harnesses
- OS tools to look at metrics such as RAM consumption
- Use Native Memory Tracking

# Find CPU and RAM requirements

2

Start from critical metrics as parameters, measure all limits

- **Default and custom thread pools**
- **GC threads and JIT threads**
- **App server thread pool (e.g. tomcat.max-threads)**
- **Heap**
- **Stacks**
- **Direct memory**
- **Metaspace**
- **CodeCache**
- **Thread stacks**
- **GC memory**

# Container images

- **Provide memory related options per service**
  - Fixed
  - Elastic (scripted, MaxRAMPercentage etc.)
- **Factor working RAM request per service based on lower boundaries**
  - at least as
    MaxDirectMemorySize
    +MaxMetaspaceSize
    +ReservedCodeCacheSize
    +Xss*threads
    +Heap
    +GC

# Container images

- **Add extra JVM container tuning, e.g.**
  - -XX:+AlwaysActAsServerClassMachine
  - -XX:+PerfDisableSharedMem
  - AppCDS
- **Smaller base images**
  - BellSoft Linux
  - Liberica Lite
- **Typically no sense to request less than 1 vCPU**
  - In throttling case, specify CPU-related JVM options by hand
  - e.g. -XX:ActiveProcessorCount

# Container images

- **Propagate application parameters**
  - Variables
  - Dedicated starter
    - Readymade starters are conservative by default
      e.g. cloudfoundry (used in Spring buildpacks)
- **Static packagers won't perform deep analysis for you**
  - e.g. App2container
  - They help to manage boilerplate

# Containers before deploy

- **Load testing, A/B, longevity testing, capacity planning**
- **Load testing results should match no container case!**
  - Technical overhead is small
- **Mock services**
- **Every container in the pod must have a memory limit and a memory request**
- **Align resource requirements with actually possible node sizing**
  - For pod scheduling
  - Especially for Fargate
- **Take K8s overhead into account**
  - e.g. Fargate takes extra 256 MB RAM
- **Ensure ephemeral storage amount is sufficient**

# Containers after deploy

- **Health monitoring**
  - Memory, cpu, threads, error rate
- **Metrics**
  - Latency, throughput, slow DB, GC stats, load, uptime, log sizes
- **Training**
  - Test services outage, simulate slow latency and network instability
- **Rate limiting**
- **Daily load testing!**
  - Parse logs → Load profile → Run load → Analysis
  - Parameter tuning

# Containers optimization

- Build continuous refinement process
- Involve Devs
- Distinguish types of services
- Optimize per service
- Control JVM options
- Start from non-container environment


# Thank you for your attention!

https://bell-sw.com

dmitry.chuyko@bell-sw.com

@dchuyko