SPRING IO

#springio24

The Future of Java Performance in Serverless: Native Java, CRaC and Project Leyden in the modern enterprise

Dan Dobrin
@ddobrin

Abdelfettah Sghiouar
@boredabdel



MAY · 30-31 · 2024

SPRINGIO.NET

The Success

Dev Teams love Serverless

- Fully managed, highly scalable app platforms
- Support for open standards
- Rapid developer onboarding
- Security offered by the infrastructure

The Challenges

There are some

- Improve performance w/more CPU+memory
- Predefined computer configurations
- Scalability control
- Cold starts - initial container boot-up

The Mission

We have to aim for something today

- Understand what can trip us up
- Know what to optimize for in serverless
- What should we adopt *today* for Serverless Java in Prod
- And then there is K8s, does the same apply there?

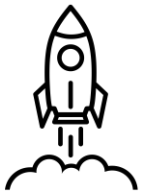


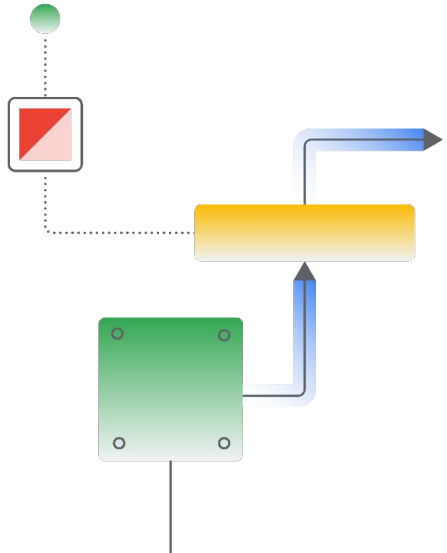
Dan Dobrin
App Architect
@ddobrin



Abdelfettah Sghiouar
Cloud Developer Advocate
@boredabdel

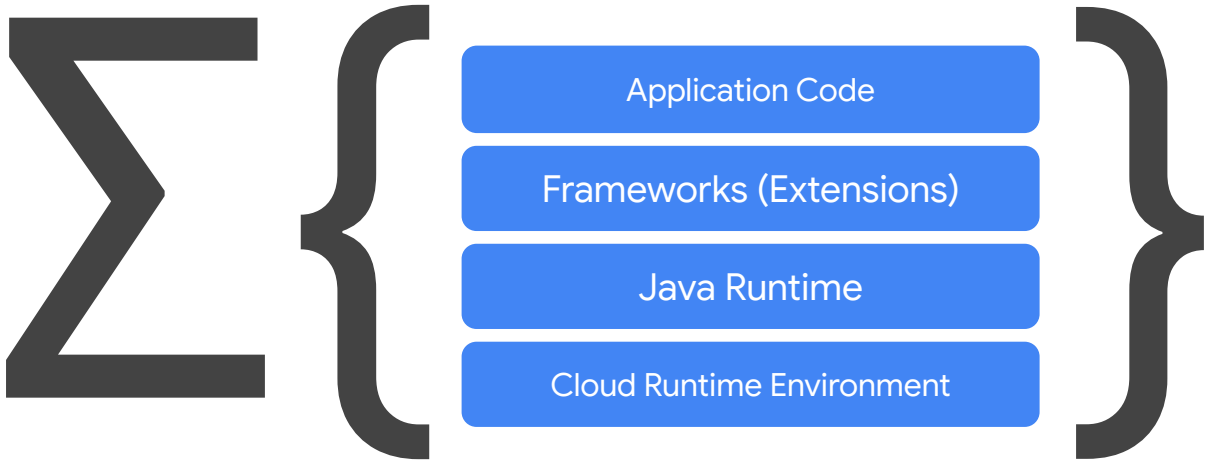
Let’s





The Idea

Basic Arithmetics

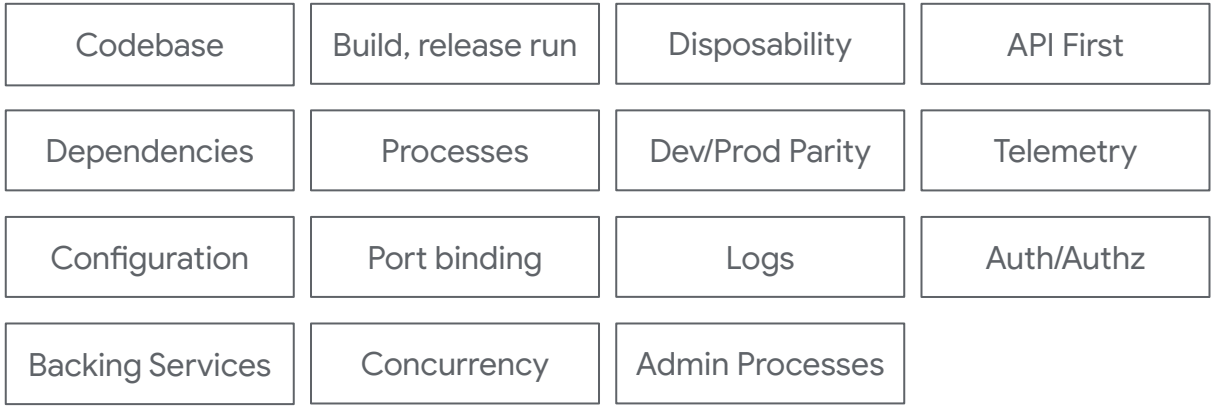


Devs asking

- Why do services start slow in Serverless, is it Java?
- Why this high number of service instances?
- My container is not accepting traffic
- Is there a one-knob optimization?
- How should I think about optimization, any ideas?

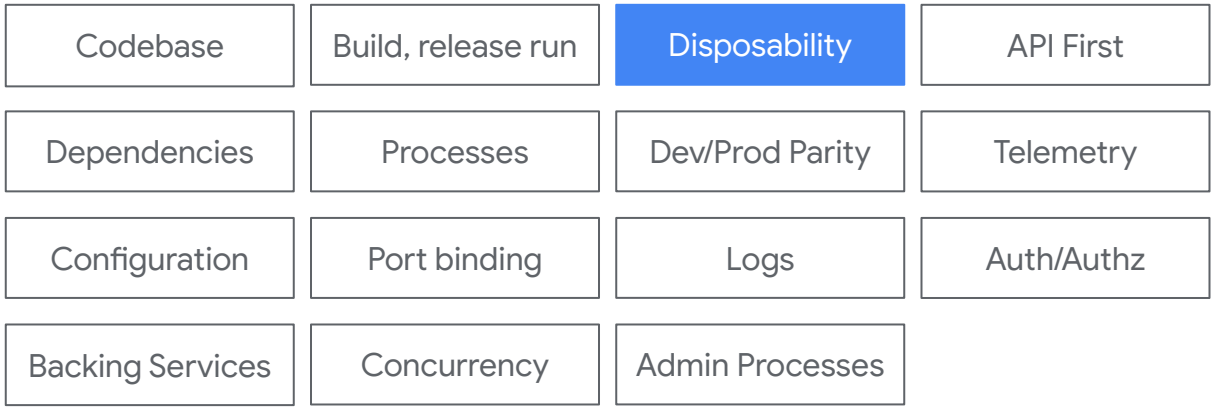
~~12~~15

Factor apps



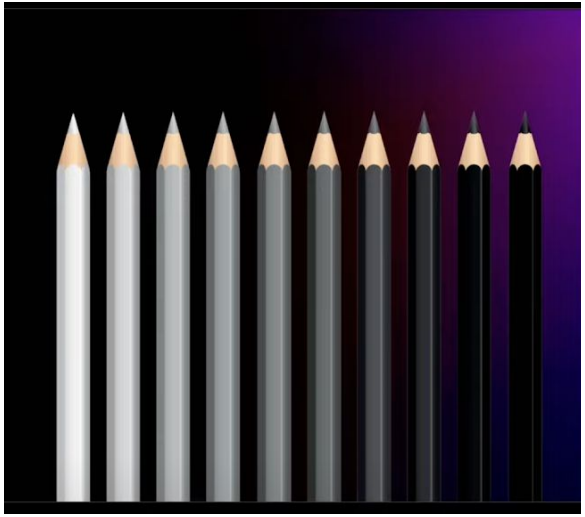
~~12~~15

Factor apps



Finding a balance

- **Startup-time**
- Resource consumption
- Concurrency
- Container image
- Cost
- Maintainability
- **Throughput**

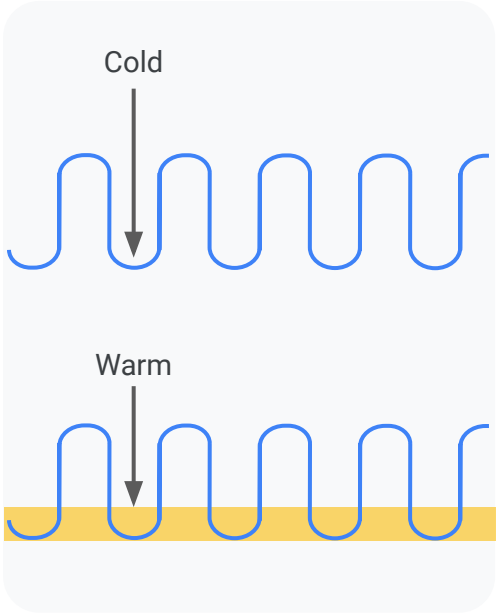


The problem of scale-to-zero is the same as scale-up when it comes to cold-starts !!!

Autoscaling

Impacted by multiple factors

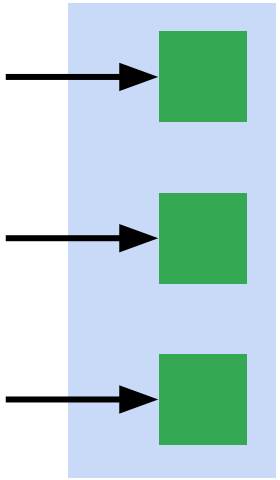
- CPU utilization of existing instances
- Minimum # instances (provisioned)
- Maximum # instances (reserved)
- Request concurrency

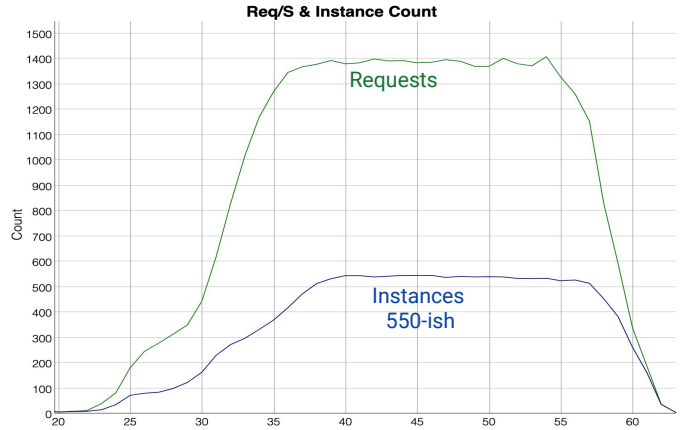


Concurrency

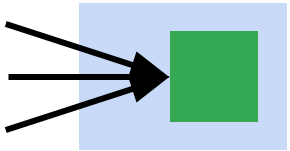
400 Clients making 3 Req/sec

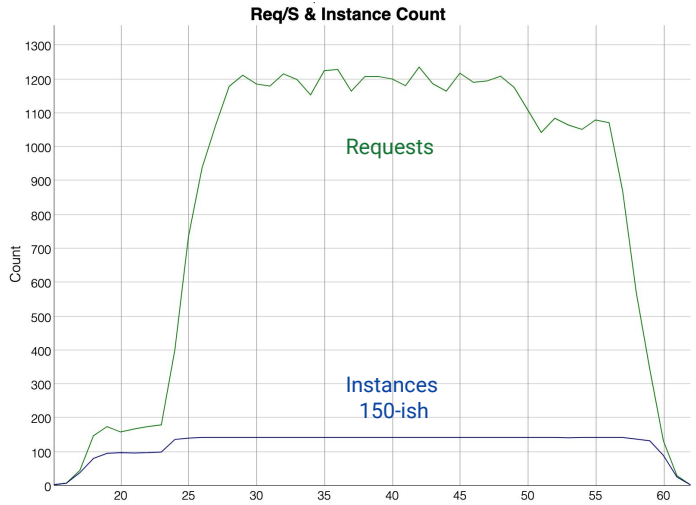
concurrency = 1





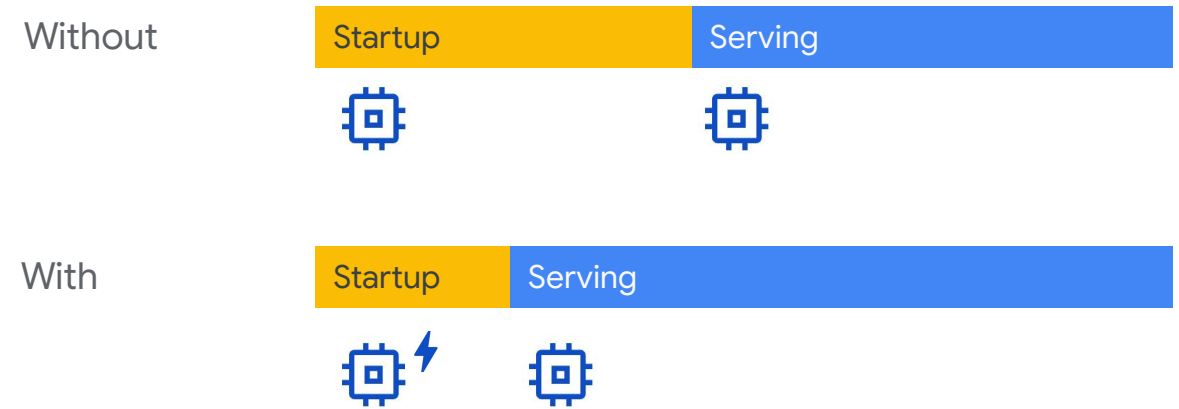
concurrency = 80





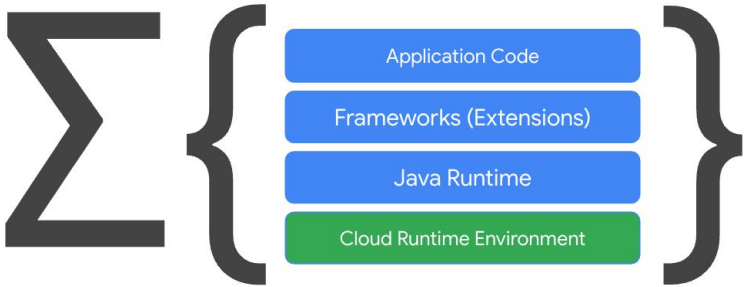
CPU Boost

Google Optimization



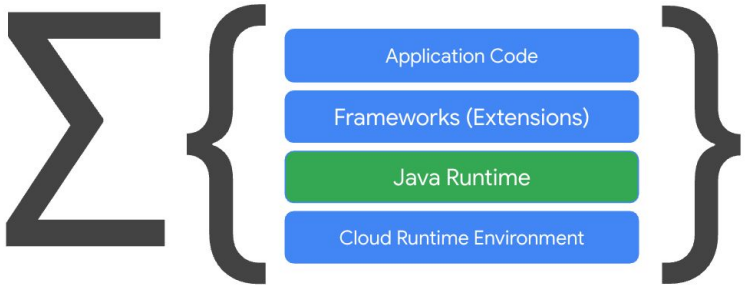
Optimize the Cloud Runtime

- Increase allocated vCPUs
- Use a different runtime environment
- Use CPU-Boost
- Tune Min instances wisely (and proactively)
- Match memory to concurrency



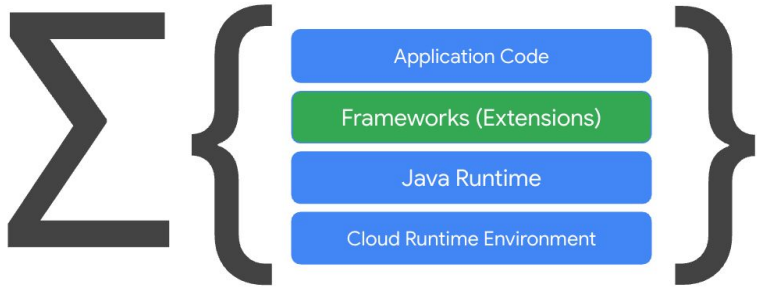
Optimize the JVM

- Use the latest Java LTS version
- Never rely on default Java Ergonomics, set them yourself
- Set memory as a % of container memory
- Set the garbage collector explicitly
- Reduce the thread stack size



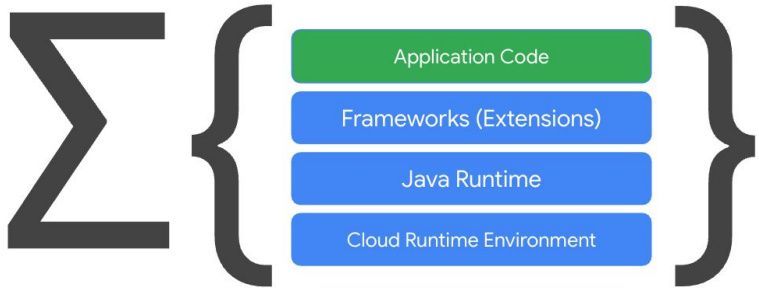
Optimize Spring Boot

- Limit unnecessary auto-configuration
- Optimize classpath scanning
- Optimize dependency management
- Fine-tune logging
- Disable actuator endpoints you don't need



Optimize the Application

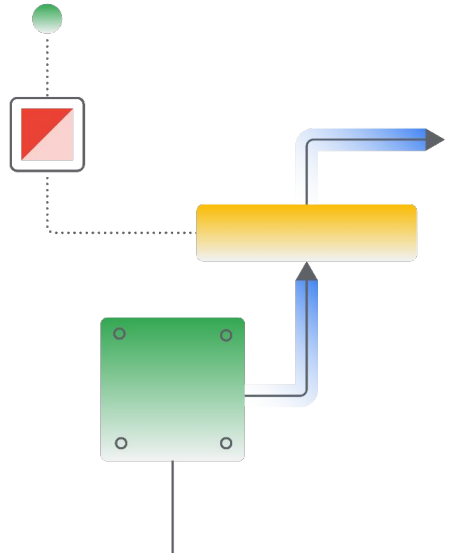
- Eliminate unnecessary dependencies
- Minimize the container image
- Layer your images
- Avoid calling other services during start-up
- Load data from a cache vs DB, on demand



Demo
An optimization journey



Step += 1



Class Data Sharing (AppCDS)

- JVM feature, helps reduce the startup time and memory footprint between multiple JVMs
- Class metadata cached in an *archive* file ⟶ pre-load quickly into a fresh JVM instance
- Caches system classes as well as your application classes
- Relies on a *training* run to exercise the startup and warmup code paths to be captured

JEP 310: Application Class-Data Sharing

Why AppCDS

- Production-ready, mature technology, constantly improving
- AppCDS does not cached the initialized values of static variables
- More secure due to no risk of leaking secrets in the CDS archive
- Less constraints and side-effects than GraalVM or Project CRaC, with an unchanged app behaviour

AppCDS in Spring

- Dev friendly, w/great performance improvements in Boot 3.3.0
- Spring Boot default unpacked layout is optimal and CDS friendly
- Starts and immediately exits Spring app before lifecycle start
  -Dspring.context.exit=onRefresh
- Creates the CDS archive on exit
  -XX:ArchiveClassesAtExit=application.jsa

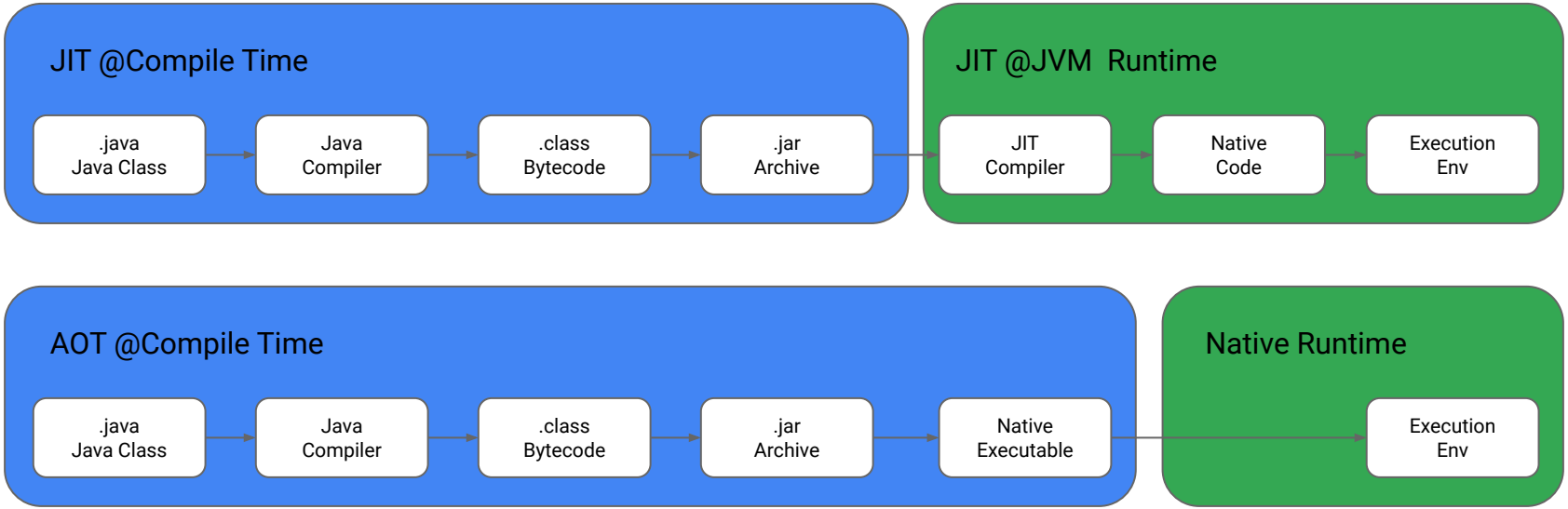
Any trade-offs?

- Improvements not as significant as GraalVM or Project CRaC
- Requires that JDK and Classpath used in archive creation to be identical to the one at application startup

Our Serverless Java App has a SLA < 1s

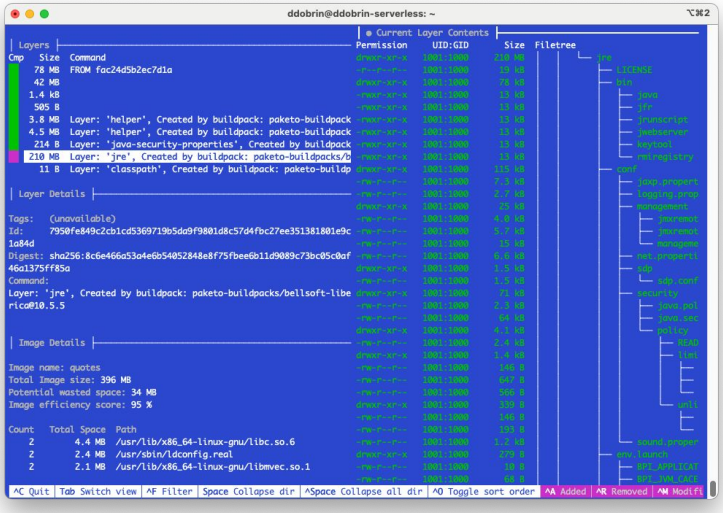
We need next-level performance!

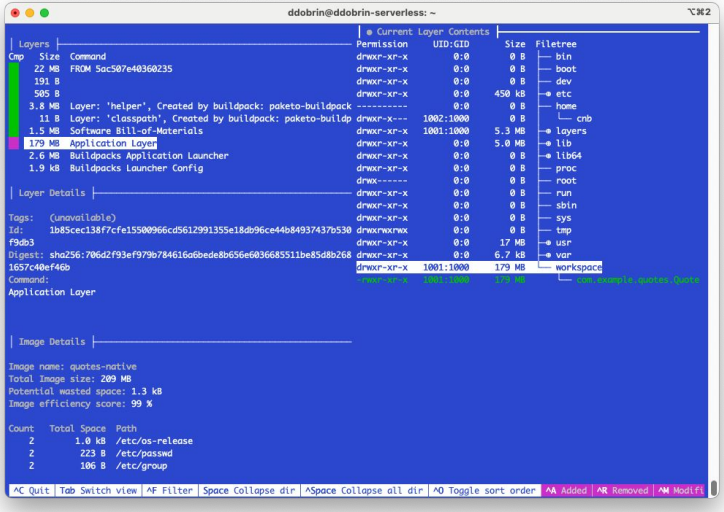
JIT vs AOT


JIT @Compile Time
.java
Java Class
Java
Compiler
.class
Bytecode
.jar
Archive
JIT @JVM Runtime
JIT
Compiler
Native
Code
Execution
Env
AOT @Compile Time
.java
Java Class
Java
Compiler
.class
Bytecode
.jar
Archive
Native
Executable
Native Runtime
Execution
Env

What are Native Images

- Self-contained executables, built directly from Java apps
- Run without the need of a JVM
- Specific to individual OS/Machine Architecture
- Compiled Ahead-of-Time(AOT) using [GraalVM]
- Smaller Images, consume less CPU, they're green(er) too





Why use Native Images with GraalVM

				
Super-fast start	Peak performance from first request	Smaller container images	Lower CPU and memory usage	Smaller attack surface

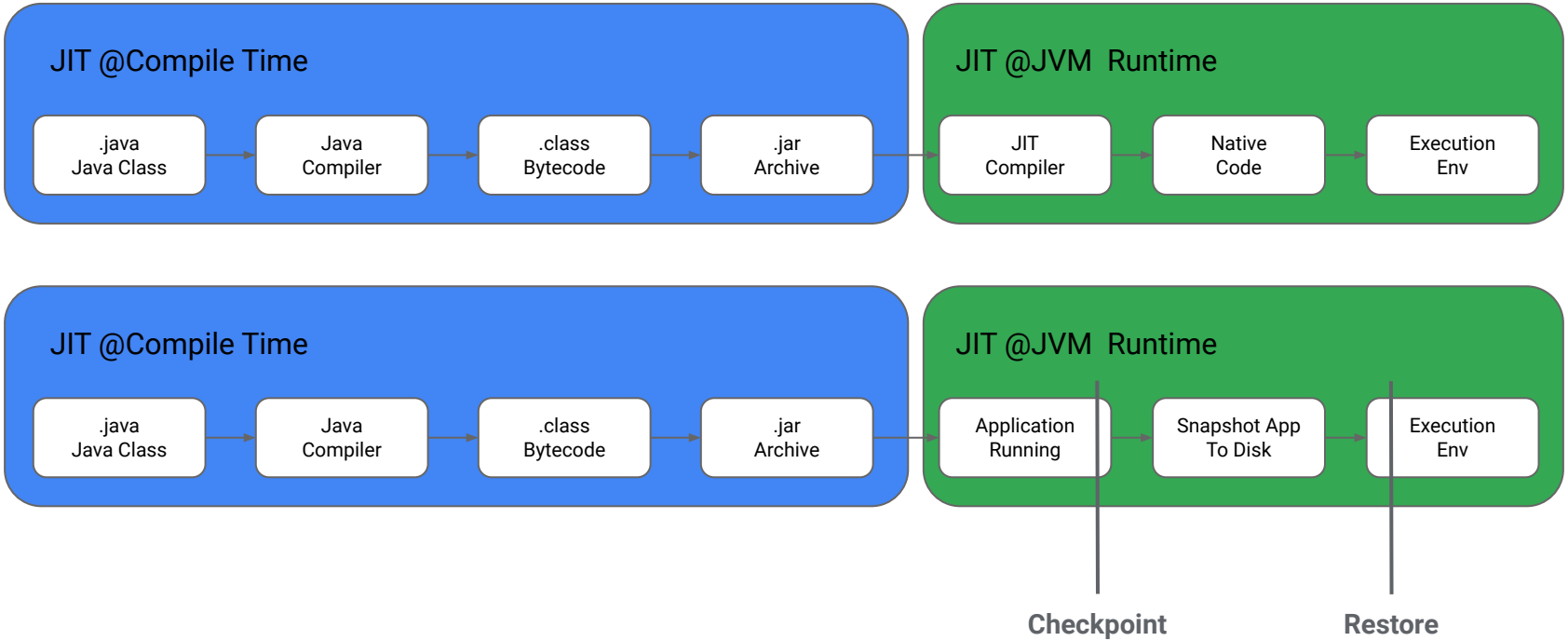
Any trade-offs?

		
Slow compile time (secs to mins)	Closed-world assumptions with AOT	Additional Shared Metadata required for some 3rd party libs

Shared Metadata Repository

What is the alternative to Native Java?

JIT vs CRaC


JIT @Compile Time
.java
Java Class
Java
Compiler
.class
Bytecode
.jar
Archive
JIT @JVM Runtime
JIT
Compiler
Native
Code
Execution
Env
JIT @Compile Time
.java
Java Class
Java
Compiler
.class
Bytecode
.jar
Archive
JIT @JVM Runtime
Application
Running
Snapshot App
To Disk
Execution
Env
Checkpoint
Restore

Why use Project CRaC

		
Super-fast startup time	Peak performance from first request, if checkpointing a fully warmed up image	Easy developer onboarding

Project CRaC trade-offs?

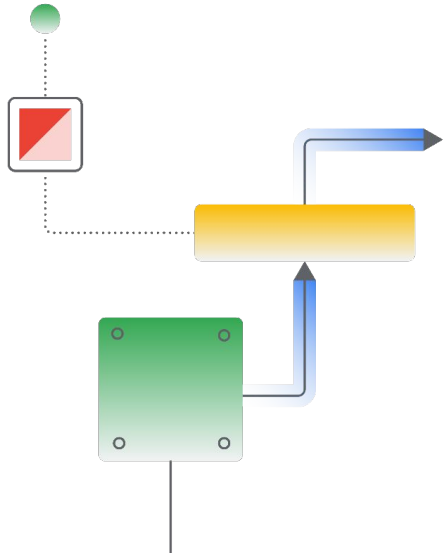
				
Required to checkpoint/store Java application upfront	Peak performance dependent on checkpoint timing in app lifecycle	Must close and reopen files, connection pools, sockets	Linux runtime only due to CRIU dependency	Sensitive data potentially leaked in snapshots

Demo
Next-level performance



OK?

...How about ?



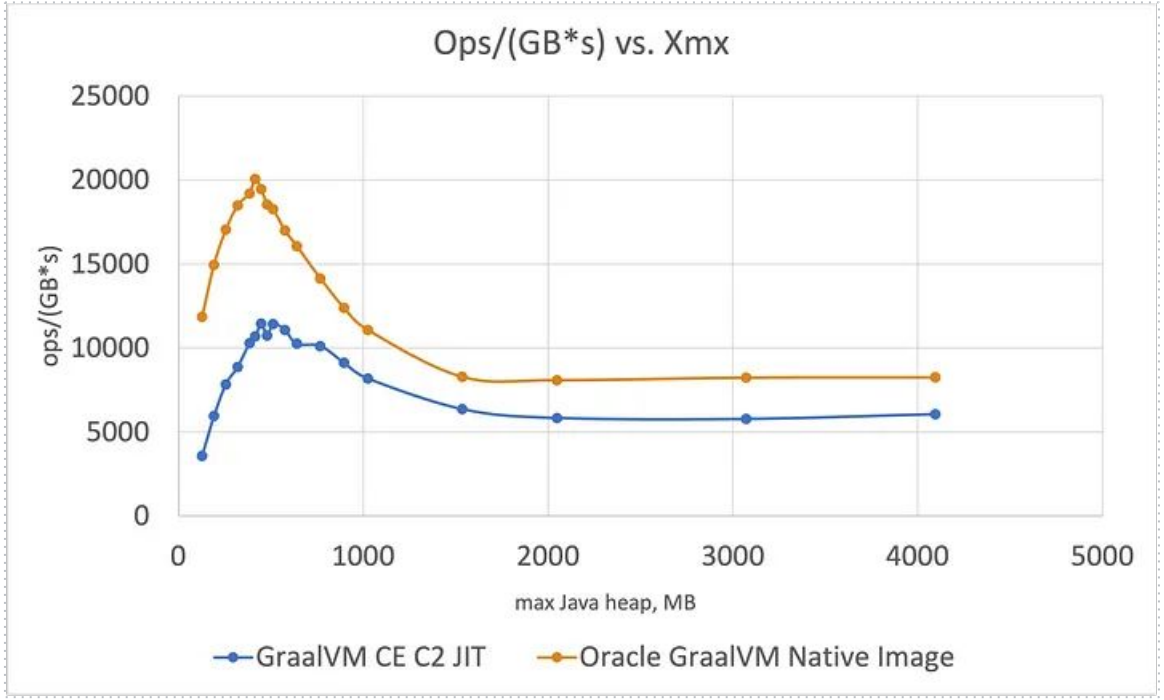
PGO

Why do I need it?

- **JIT** compiler: ability to analyze the runtime behaviour of an app by acquiring a “profile” for your app and passes it to a Tier2
- **AOT** compiler: does not have profiling information
- **PGO**: give the AOT compiler both a static view and a dynamic profile of the application
- **End-result**: optimized app performs better than the default AOT-compiled app

Peak performance

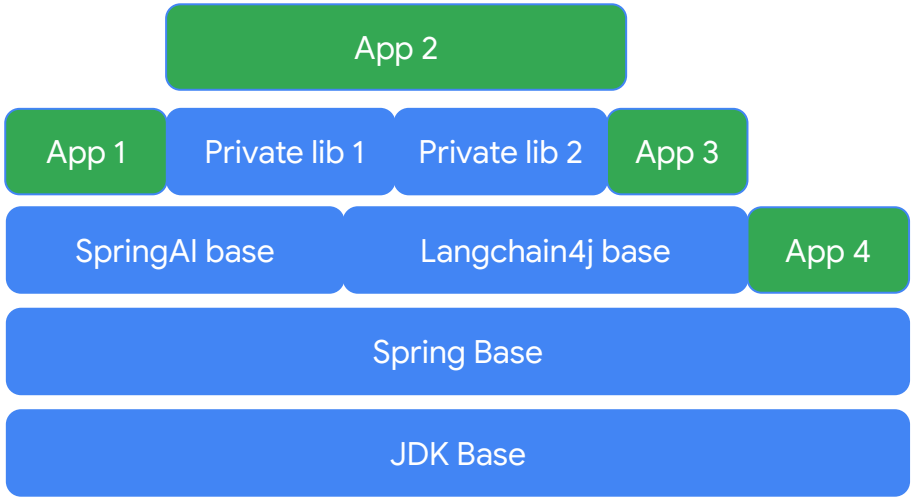
PGO vs Hotspot



Courtesy of GraalVM team

Layered Native Images

The missing link?

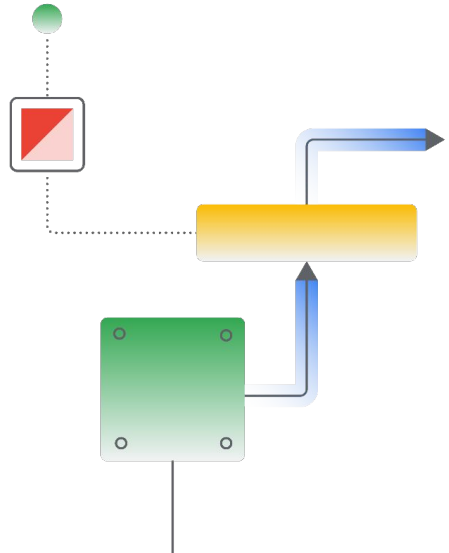


[GR-48103] Layered Native Images

Demo
Throughput w/PGO



Future...



Project Leyden

Magic solution one day?



- **Focus** on improving startup, warmup and footprint
- **Shift** computation earlier to build-time (ex. ahead-of-time compilation, AppCDS time, etc)
- **Constrain** Java's dynamism (ex closed-world assumptions)
- **Preserve** the meaning of the resulting app image as the original
- **Select** which aspects can be shifted and which can be constrained

Project Leyden, Talk

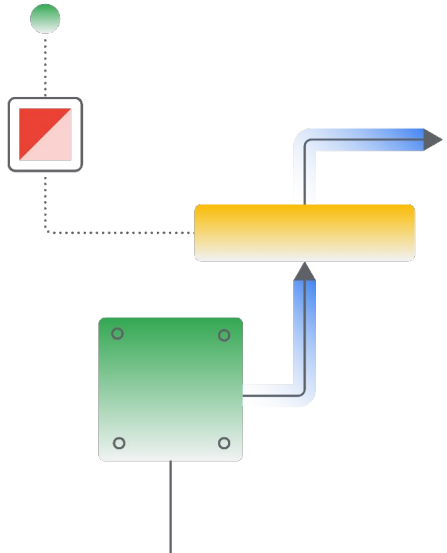
Which technology should you use?

The one knob

If you had a single choice

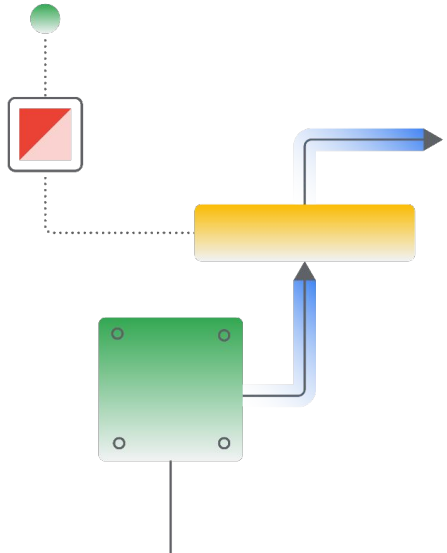
- Java 21 + GraalVM + PGO + G1 + (-O3)
- Production-ready ✅

And there is K8s, does the same apply?



Quick recap on **resource management**

in Kubernetes



Google

Resource Requirements

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Container resource needs are expressed in a **Pod Template** with resource **requests** and **limits**

Google

Resource Requirements

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Requests resources are **reserved** by the kubelet; they are also **used** in a **scheduling decision**

Google

Resource Requirements

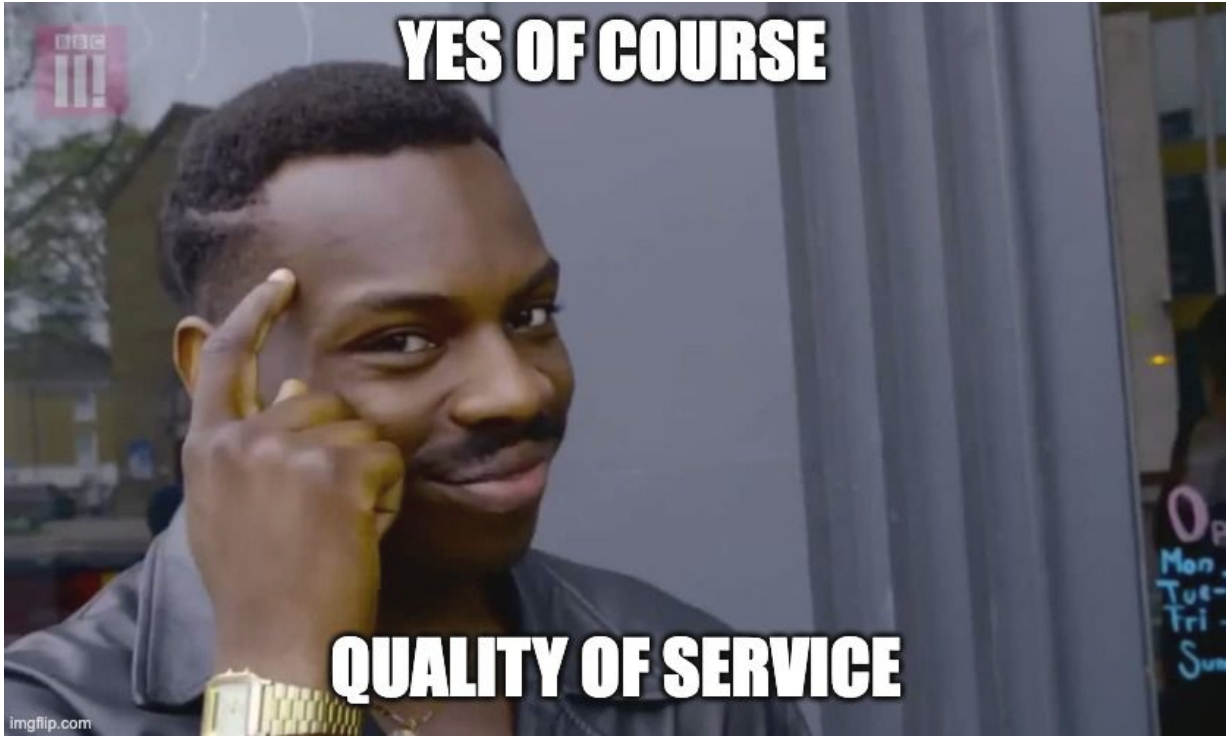
```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Limits resources are **enforced** by the kubelet, so the **container is not allowed** to use more than the limit

We saw **resource reservation**,

enforcement and **scheduling decisions**.

Is there anything else?



Google Cloud

Google

Pod Quality of service

Guaranteed

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "64Mi"
        cpu: "250m"
```

For **every** container in a POD:

- Requests & limits defined
- Requests = limits

Google

Pod Quality of service

Guaranteed

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "64Mi"
        cpu: "250m"
```

For **every** container in a POD:

- Requests & limits defined
- Requests = limits

Burstable

```
...
  containers:
  - name: nginx-one
    image: nginx:1.14.2
  - name: nginx-two
    image: nginx:1.14.2
    resources:
      limits:
        memory: "128Mi"
        cpu: "1"
  - name: nginx-three
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
```

- Doesn't fall into Guaranteed class
- **At least one** container **has** memory or CPU **request or limit**

Google

Pod Quality of service

Guaranteed

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "64Mi"
        cpu: "250m"
```

For **every** container in a POD:

- Requests & limits defined
- Requests = limits

Burstable

```
...
  containers:
  - name: nginx-one
    image: nginx:1.14.2
  - name: nginx-two
    image: nginx:1.14.2
    resources:
      limits:
        memory: "128Mi"
        cpu: "1"
  - name: nginx-three
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
```

- Doesn't fall into Guaranteed class
- **At least one** container **has** memory or CPU **request or limit**

Best Effort

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx-one
    image: nginx:1.14.2
  - name: nginx-two
    image: nginx:1.14.2
  - name: nginx-three
    image: nginx:1.14.2
```

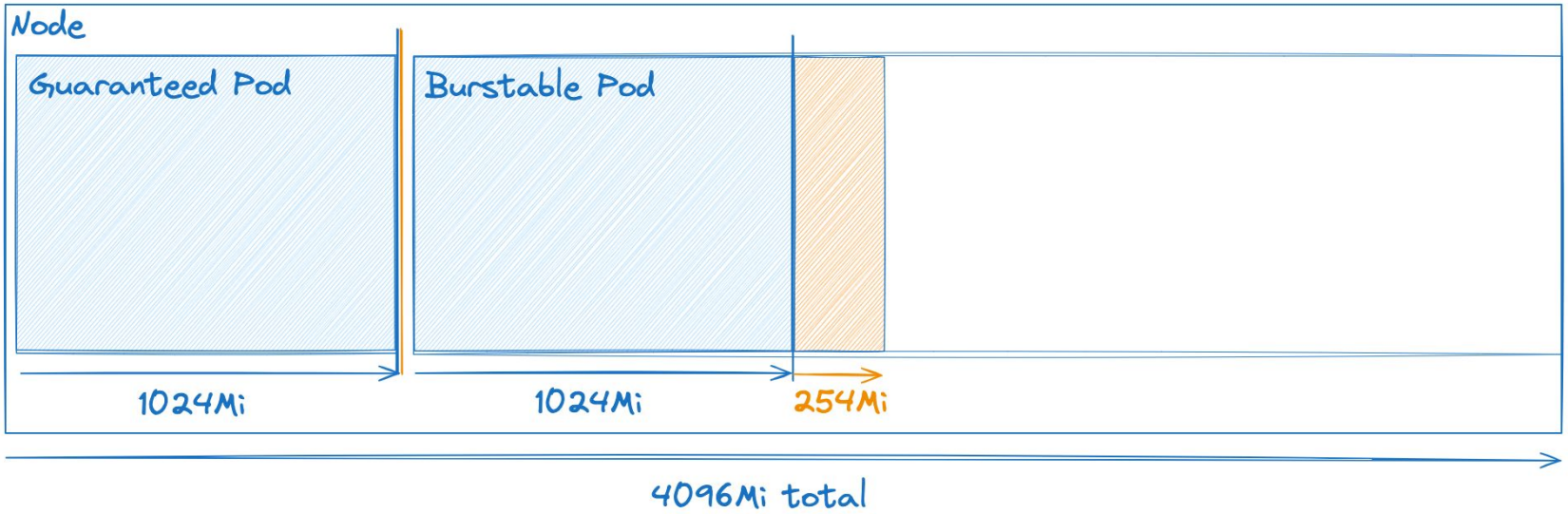
Doesn't fall into Guaranteed nor Burstable classes

Kubelet can **evict PODs** when **Node Pressure** occurs.

The **QOS class** of a POD is **used in a decision**.

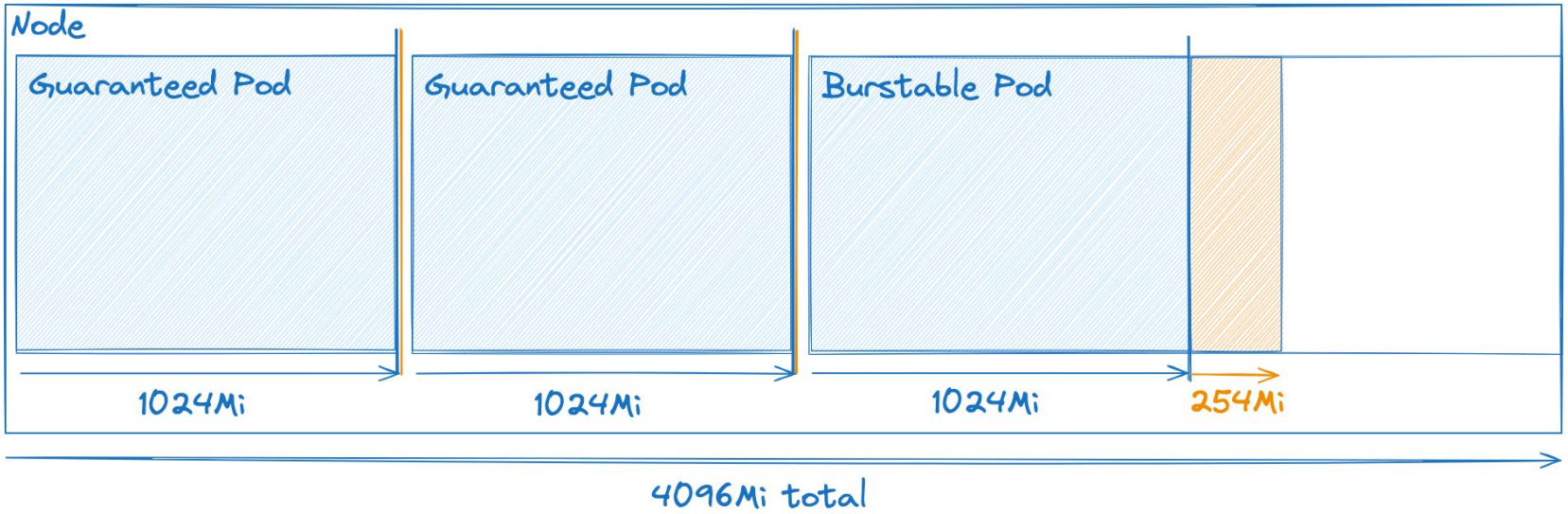
Google

Pod Quality of Service



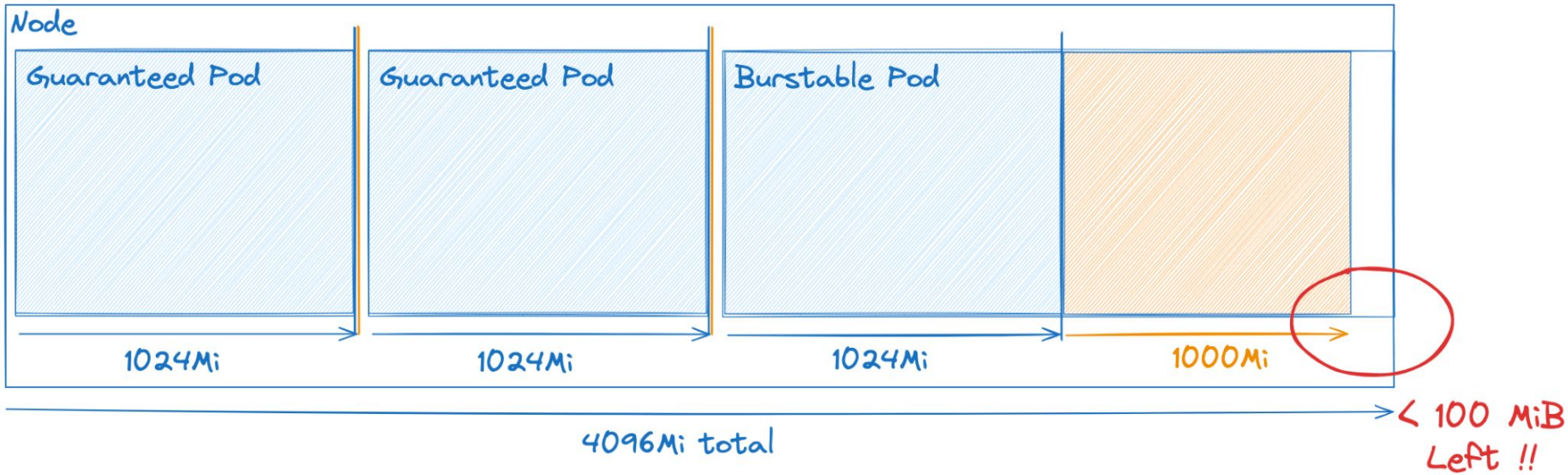
Google

Pod Quality of Service



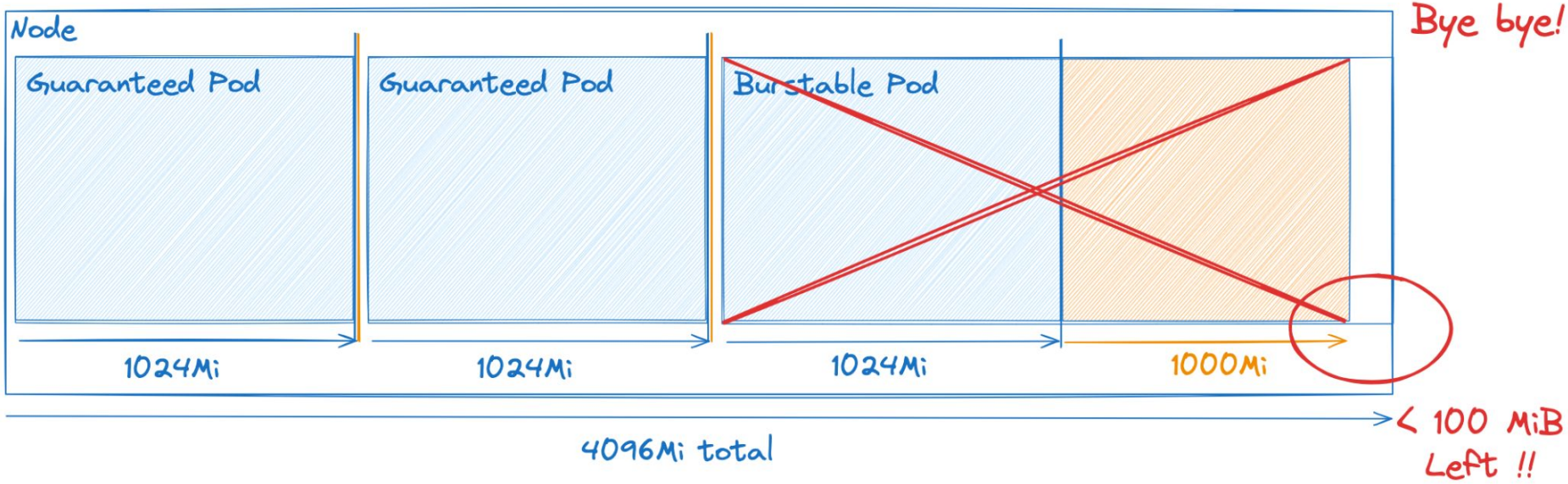
Google

Pod Quality of Service



Google

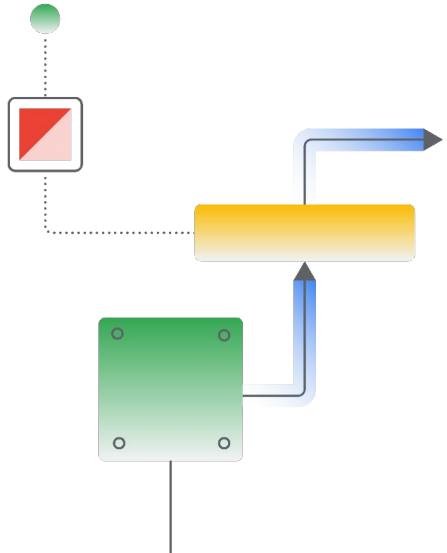
Pod Quality of Service



For **CPU** resources **overrun**,

PODs are not evicted but **throttled** instead

How does the **Kubernetes** **resource requirements** translate to **containers**?



Google

Container Resources + cgroups

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Google

Container Resources + cgroups

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

```
$ crictl inspect 7caaf821bbbb6 | \
  jq '.info.runtimeSpec.linux.resources | \
  {cpu: .cpu, memory: .memory}"
{
  "cpu": {
    "shares": 256,
    "quota": 50000,
    "period": 100000
  },
  "memory": {
    "limit": 134217728,
    "swap": 134217728
  }
}
```

Google

Container Resources + cgroups

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

```
$ crictl inspect 7caaf821bbbb6 | \
  jq '.info.runtimeSpec.linux.resources | \
  {cpu: .cpu, memory: .memory}"
{
  "cpu": {
    "shares": 256,
    "quota": 50000,
    "period": 100000
  },
  "memory": {
    "limit": 134217728,
    "swap": 134217728
  }
}
```

cpu requests -> cpu shares

250m = 1024 * 0.25 = **256 shares**

Google

Container Resources + cgroups

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

```
$ crictl inspect 7caaf821bbbb6 | \
  jq '.info.runtimeSpec.linux.resources | \
  {cpu: .cpu, memory: .memory}"
{
  "cpu": {
    "shares": 256,
    "quota": 50000,
    "period": 100000
  },
  "memory": {
    "limit": 134217728,
    "swap": 134217728
  }
}
```

cpu requests -> cpu shares

250m = 1024 * 0.25 = **256 shares**

cpu limits -> cpu quota

500m = 100000 * 0.5 = **50000 µs**

Google

Container Resources & cgroups

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

```
$ crictl inspect 7caaf821bbbb6 | \
  jq '.info.runtimeSpec.linux.resources | \
  {cpu: .cpu, memory: .memory}"
{
  "cpu": {
    "shares": 256,
    "quota": 50000,
    "period": 100000
  },
  "memory": {
    "limit": 134217728,
    "swap": 134217728
  }
}
```

cpu requests -> cpu shares

250m = 1024 * 0.25 = **256 shares**

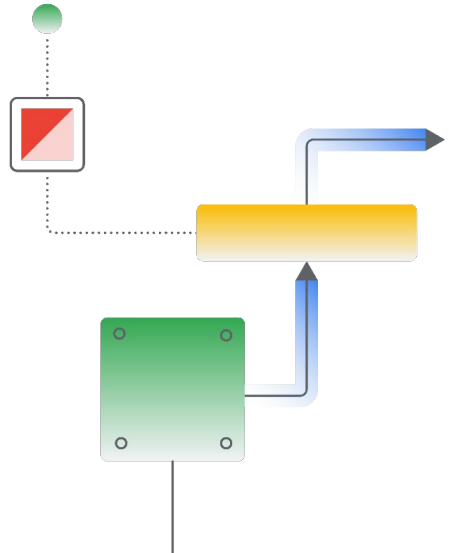
cpu limits -> cpu quota

500m = 100000 * 0.5 = **50000 µs**

memory limits -> memory limit

128Mi = **134217728 bytes**

Java is aware of **container resources**
since **version 10**

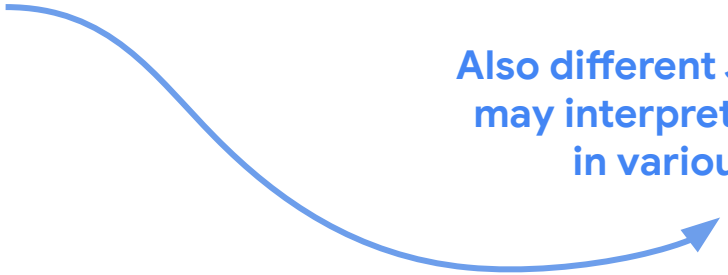


2 approaches to

faster **JVM containers** **startups**

Google

1 Set **high resource limits** or **skip setting limits** at all

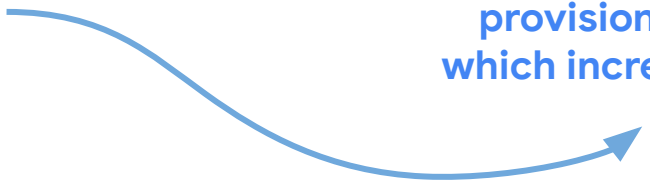


But this does not guarantee faster startup.

Also different Java versions may interpret lack of limit in various ways.

Google

2 Set **high resource requests** to cover startup peak



But this is over provisioning that which increases cost

JVM would benefit from **dynamic resource allocation.**

This is not **possible in Kubernetes.**

JVM would benefit from **dynamic resource allocation.**

This is not **possible in Kubernetes.**



Google

In-place resource resize for Kubernetes **PODs**

- Added in **Kubernetes 1.27** as an **alpha** feature
- Allows to **resize** the CPU & memory **without container restart**

Google

In-place resource resize for Kubernetes **PODs**

- Added in **Kubernetes 1.27** as an **alpha** feature
- Allows to **resize** the CPU & memory **without container restart**


```
kind: Pod
spec:
 containers:
 - name: nginx
   image: nginx:1.14.2
   resources:
     requests:
       memory: "64Mi"
       cpu: "250m"
   resizePolicy:
   - resourceName: cpu
     restartPolicy: NotRequired
   - resourceName: memory
     restartPolicy: NotRequired
status:
 resize: InProgress
 containerStatuses:
 - allocatedResources:
     cpu: 250m
     memory: 64Mi
   resources:
     requests:
       cpu: "1"
       memory: 64Mi
```

Google

In-place resource resize for Kubernetes **PODs**

- Added in **Kubernetes 1.27** as an **alpha** feature
- Allows to **resize** the CPU & memory **without container restart**

Resource requirements are mutable now

```
kind: Pod
spec:
 containers:
 - name: nginx
   image: nginx:1.14.2
   resources:
     requests:
       memory: "64Mi"
       cpu: "250m"
   resizePolicy:
   - resourceName: cpu
     restartPolicy: NotRequired
   - resourceName: memory
     restartPolicy: NotRequired
status:
 resize: InProgress
 containerStatuses:
 - allocatedResources:
     cpu: 250m
     memory: 64Mi
   resources:
     requests:
       cpu: "1"
       memory: 64Mi
```

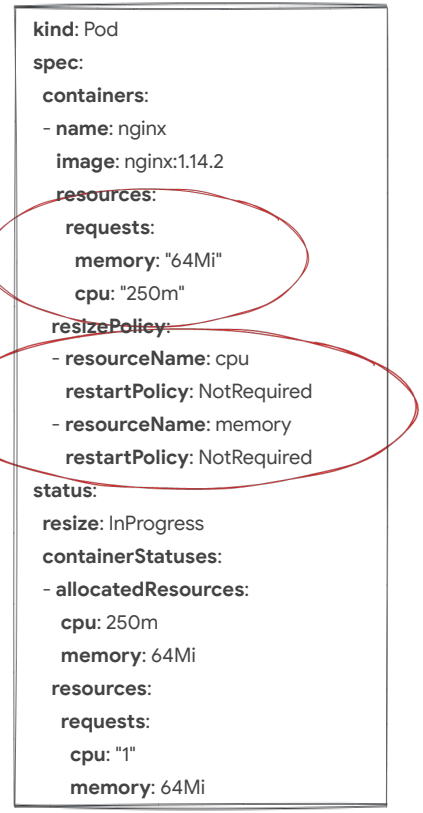
Google

In-place resource resize for Kubernetes PODs

- Added in **Kubernetes 1.27** as an **alpha** feature
- Allows to **resize** the CPU & memory **without container restart**

Resource requirements are mutable now

Containers can specify resize policy for resources



Google

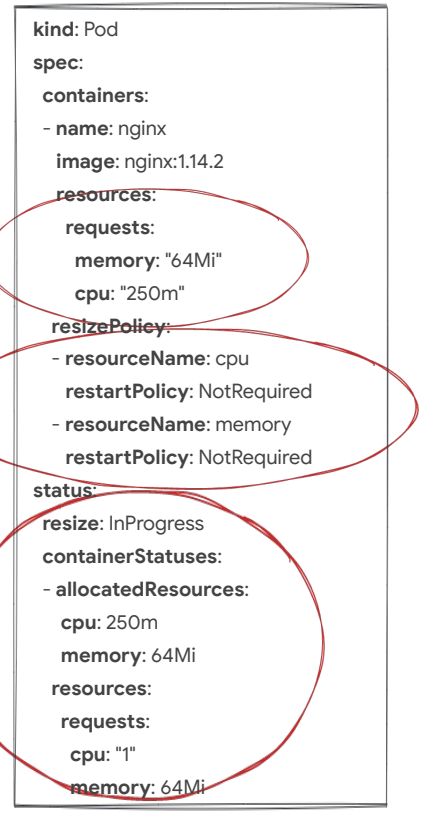
In-place resource resize for Kubernetes PODs

- Added in **Kubernetes 1.27** as an **alpha** feature
- Allows to **resize** the CPU & memory **without container restart**

Resource requirements are mutable now

Containers can specify resize policy for resources

Container statues include information about resize process, resource allocations & actuals

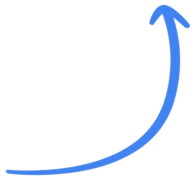


In-place POD resource resize

creates **plenty of possibilities**

In-place POD resource resize

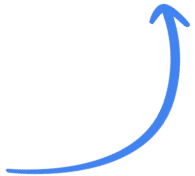
creates **plenty of possibilities**



VERTICAL POD AUTOSCALER

In-place POD resource resize

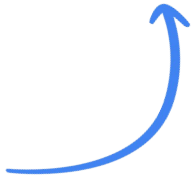
creates **plenty of possibilities**



VERTICAL POD AUTOSCALER
KUBE STARTUP CPU BOOST

In-place POD resource resize

creates **plenty of possibilities**



VERTICAL POD AUTOSCALER

KUBE STARTUP CPU BOOST

Kube Startup CPU Boost

for **workloads** in Kubernetes.

Google Cloud

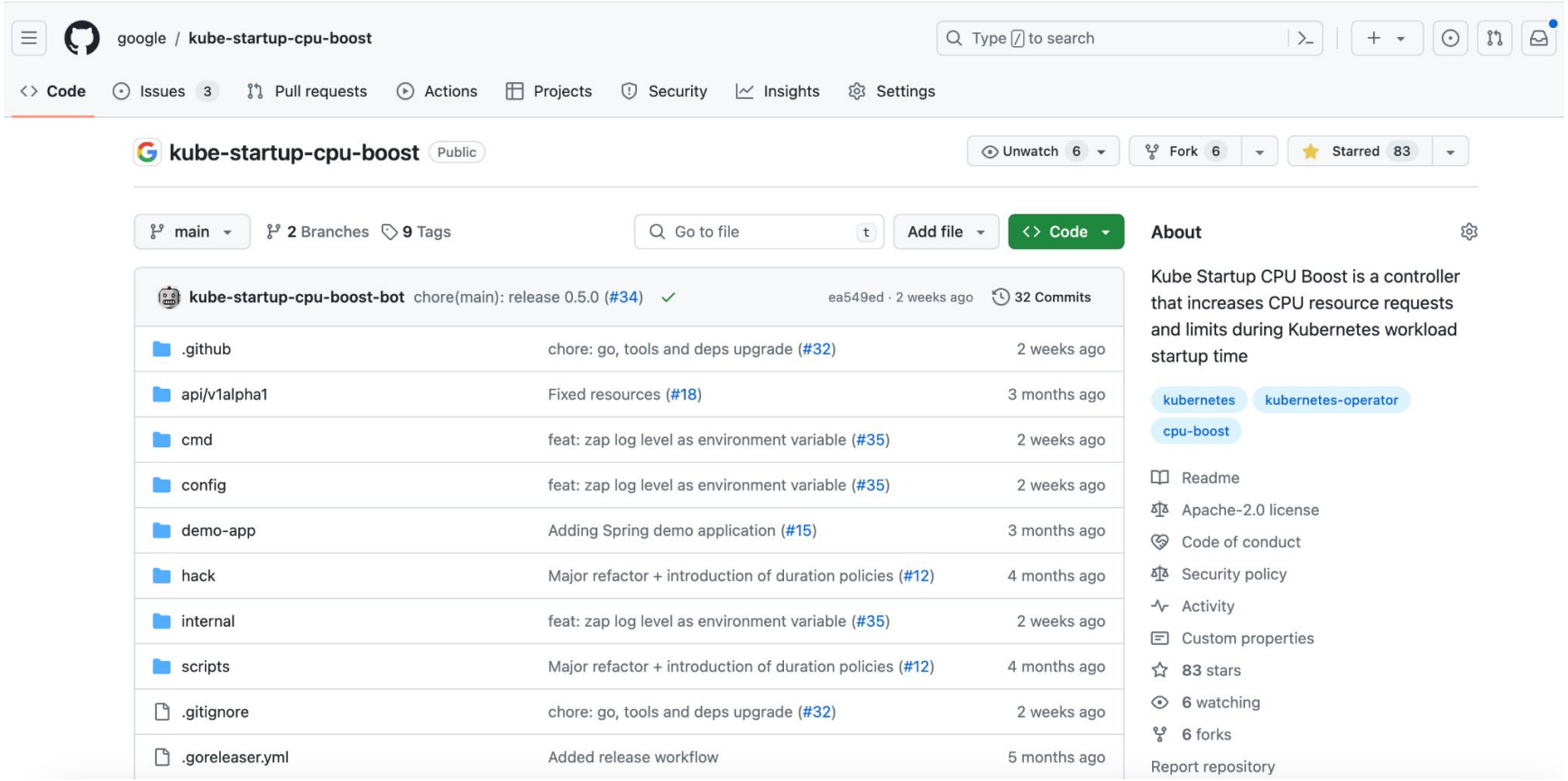
Kube Startup CPU Boost

for **workloads** in Kubernetes.



Open source
released by Google Cloud

Google Cloud



Google Cloud

Kube Startup CPU Boost

increases CPU resources for **container** **startup time**.

Google Cloud

Kube Startup CPU Boost

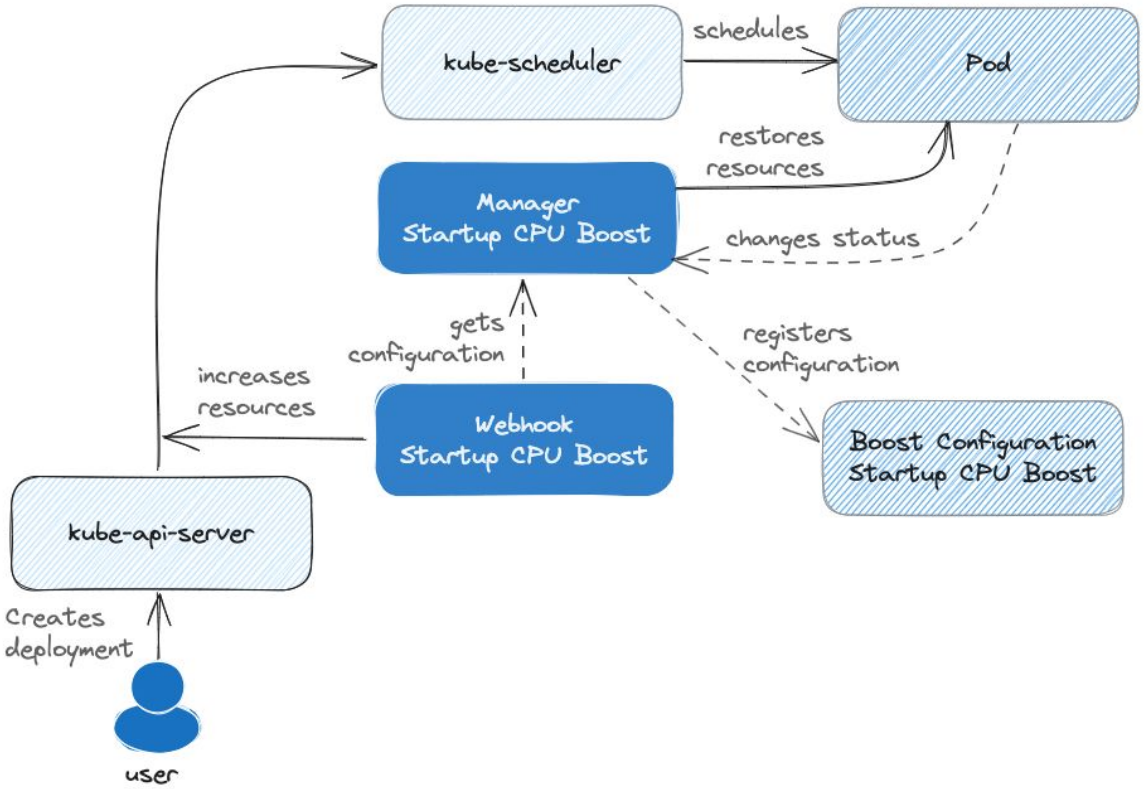
increases CPU resources for **container** **startup time**.



ORIGINAL RESOURCES ARE SET BACK
WITHOUT THE POD RESTART

Google Cloud

Google


kube-scheduler
schedules
Pod
restores
resources
Manager
Startup CPU Boost
changes status
gets
configuration
registers
configuration
increases
resources
Webhook
Startup CPU Boost
Boost Configuration
Startup CPU Boost
kube-api-server
Creates
deployment
user

Google

Startup CPU Boost

Boost spec

```
apiVersion: autoscaling.x-k8s.io/v1alpha1
kind: StartupCPUBoost
metadata:
  name: boost-001
  namespace: demo
selector:
  matchExpressions:
  - key: app.kubernetes.io/name
    operator: In
    values: ["spring-demo-app"]
spec:
  resourcePolicy:
    containerPolicies:
    - containerName: spring-demo-app
      percentageIncrease:
        value: 50
  durationPolicy:
    podCondition:
      type: Ready
          status: "True"
```

How to create the boost?

Google

Startup CPU Boost

Boost spec

```
apiVersion: autoscaling.x-k8s.io/v1alpha1
kind: StartupCPUBoost
metadata:
  name: boost-001
  namespace: demo
selector:
  matchExpressions:
  - key: app.kubernetes.io/name
    operator: In
    values: ["spring-demo-app"]
spec:
  resourcePolicy:
    containerPolicies:
    - containerName: spring-demo-app
      percentageIncrease:
        value: 50
  durationPolicy:
    podCondition:
      type: Ready
          status: "True"
```

How to create the boost?

✓ Add selector

Google

Startup CPU Boost

Boost spec

```
apiVersion: autoscaling.x-k8s.io/v1alpha1
kind: StartupCPUBoost
metadata:
  name: boost-001
  namespace: demo
selector:
  matchExpressions:
  - key: app.kubernetes.io/name
    operator: In
    values: ["spring-demo-app"]
spec:
  resourcePolicy:
    containerPolicies:
    - containerName: spring-demo-app
      percentageIncrease:
        value: 50
  durationPolicy:
    podCondition:
      type: Ready
        status: "True"
```

How to create the boost?

- ✓ Add selector
- ✓ Specify resource policy

Google

Startup CPU Boost

Boost spec

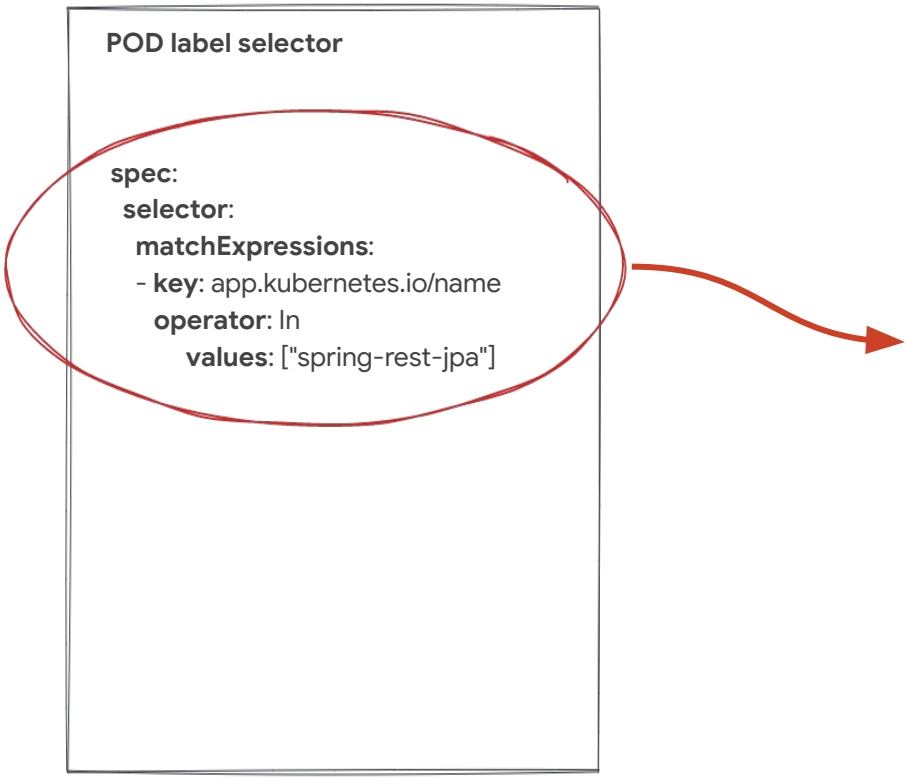
```
apiVersion: autoscaling.x-k8s.io/v1alpha1
kind: StartupCPUBoost
metadata:
  name: boost-001
  namespace: demo
selector:
  matchExpressions:
  - key: app.kubernetes.io/name
    operator: In
    values: ["spring-demo-app"]
spec:
  resourcePolicy:
    containerPolicies:
    - containerName: spring-demo-app
      percentageIncrease:
        value: 50
  durationPolicy:
    podCondition:
      type: Ready
        status: "True"
```

How to create the boost?

- ✓ Add selector
- ✓ Specify resource policy
- ✓ Specify duration policy

Google

Startup CPU Boost



To define the PODs that are subject for resource boost

Google

Startup CPU Boost

Boost resources

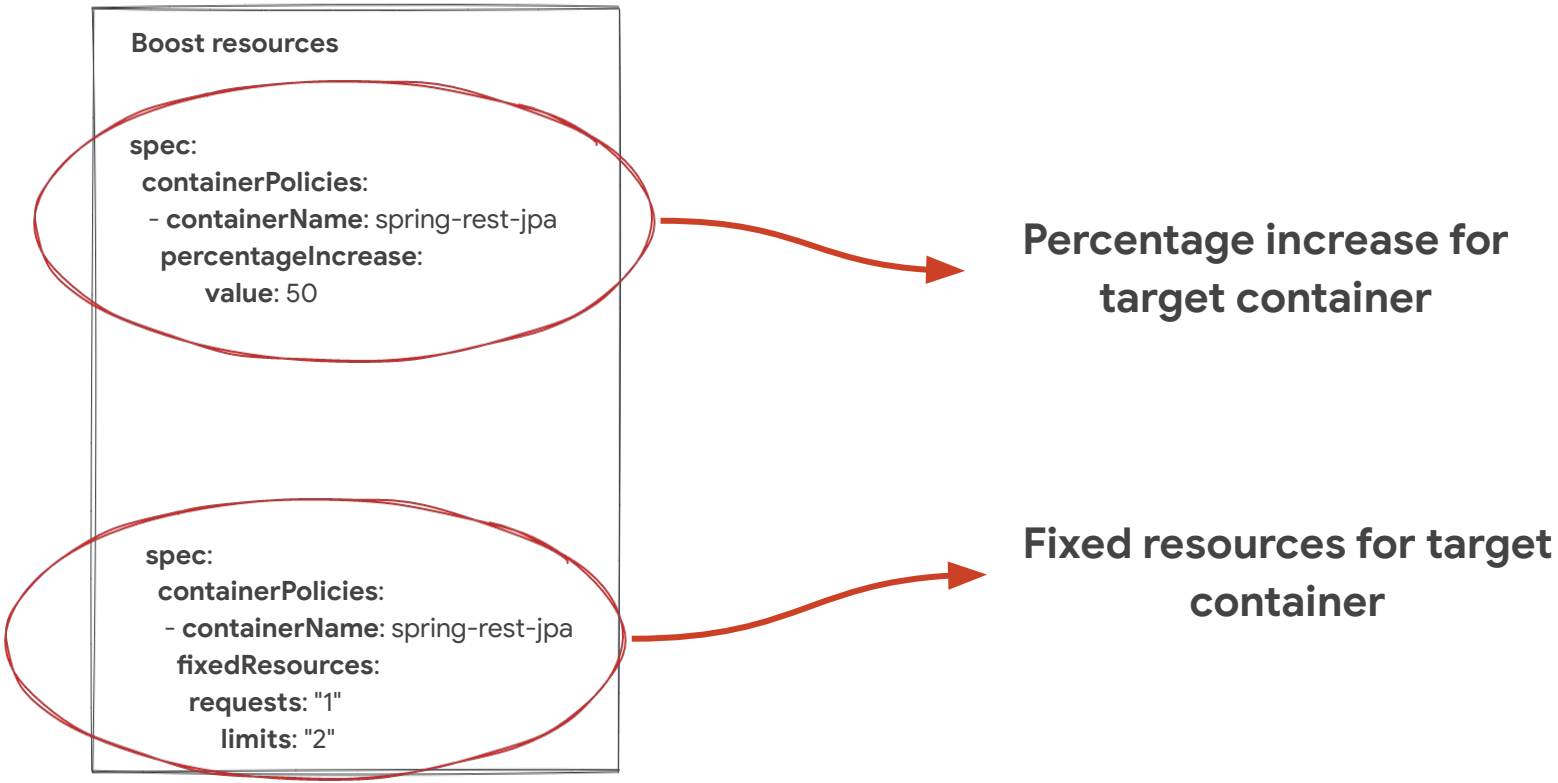
```
spec:
  containerPolicies:
  - containerName: spring-rest-jpa
    percentageIncrease:
        value: 50
```



Percentage increase for target container

Google

Startup CPU Boost



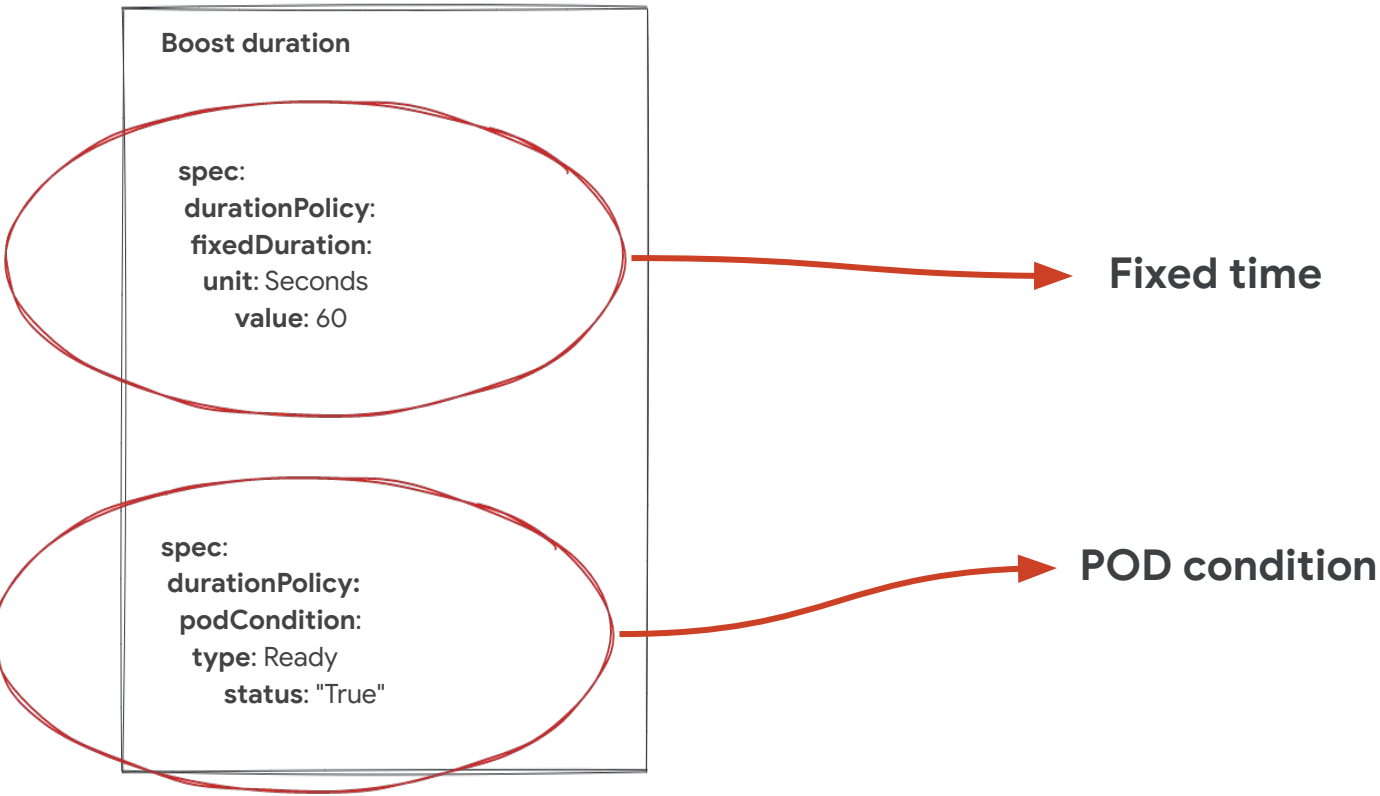
Google

Startup CPU Boost



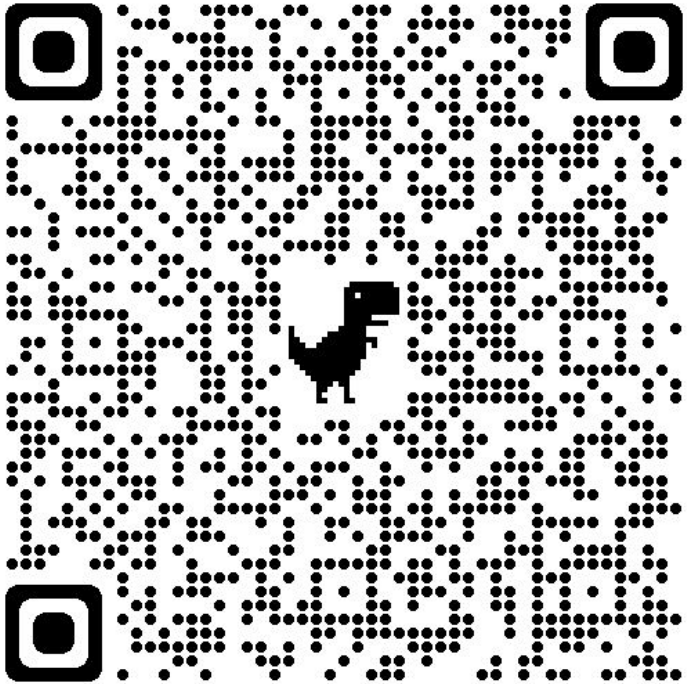
Google

Startup CPU Boost



Google

Startup CPU Boost blog post



Demo
An optimization journey



Live Q&A

An optimization journey



THANKS!!!

Dan Dobrin
@ddobrin

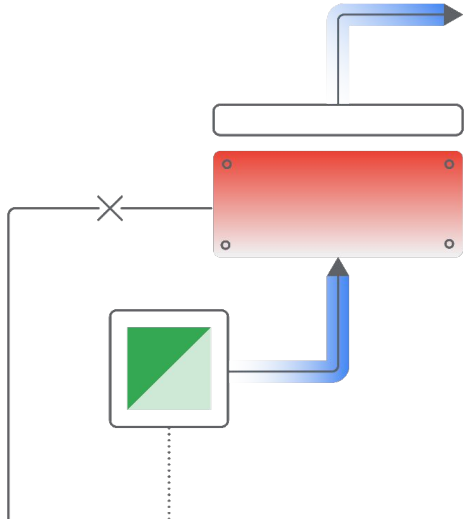
Abdelfettah Sghiouar
@boredabdel

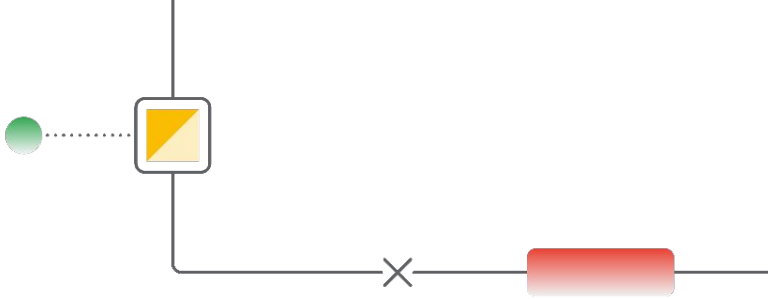


The Future of Java Performance in Serverless

Native Java, CRaC and Project Leyden in the modern enterprise

@ddobrin @boredabdel





Thank you

Dan Dobrin

@ddobrin

Abdelfettah Sghiouar

@boredabdel

