GraalVM Native Images für SpringBoot auf Kubernetes

Dominik Schießl & Hendrik Still

MLP

Finanzen verstehen. Richtig entscheiden.

Disclaimer

- No legal advice! (Oracle-Technology)
- Experimental Use of GraalVM (no productive use)

Offene Stellen

- Fullstack-Software-Entwickler/in (m/w/d)
- Auszubildende (Systemintegration)
- Duale Studenten/innen (Cyber Security / Data Science & KI)

Agenda

01 **Problem Statement**

02 **GraalVM**
Natives Images

03 **Introduction: Time for Code**
Demo

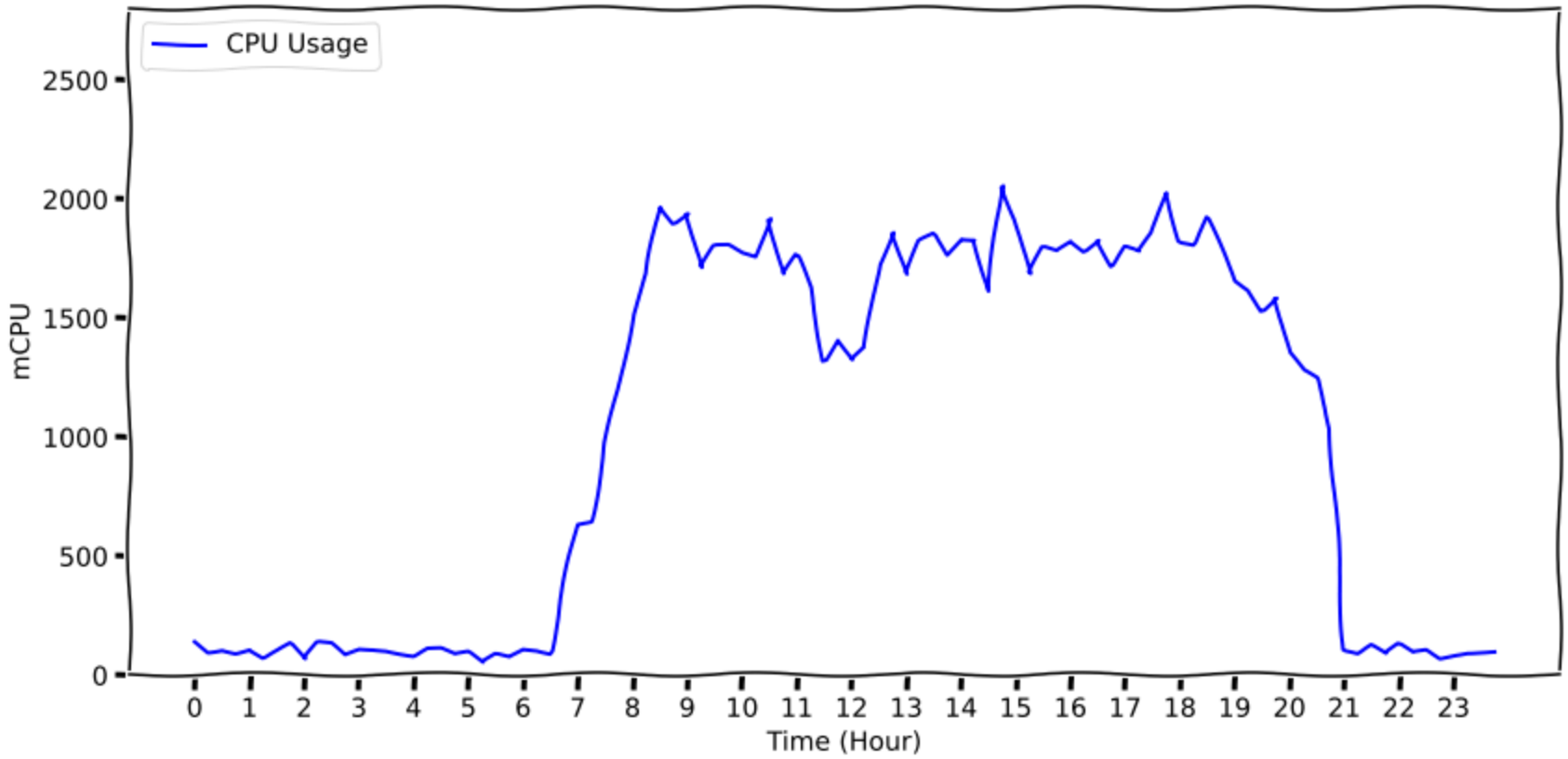
04 **Performance**

05 **Conclusion**

01

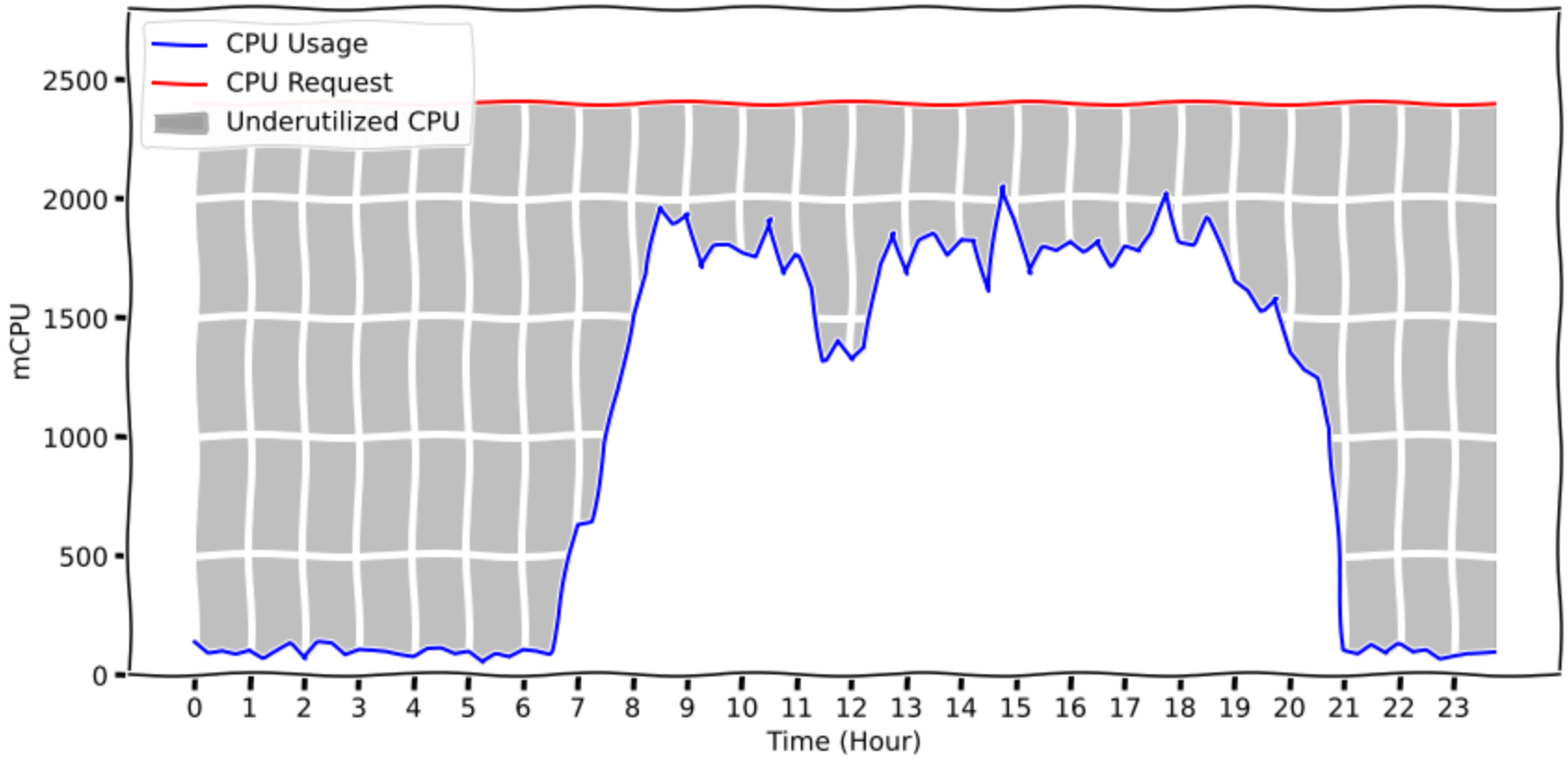
Problem Statement

Capacity Planning



MLP

Capacity Planning



MLP

HPA to the rescue!

- HorizontalPodAutoscaler (HPA):
  - automatically updates number of running pods
  - scaling is based on a target value
  - multiple type of metrics can be used

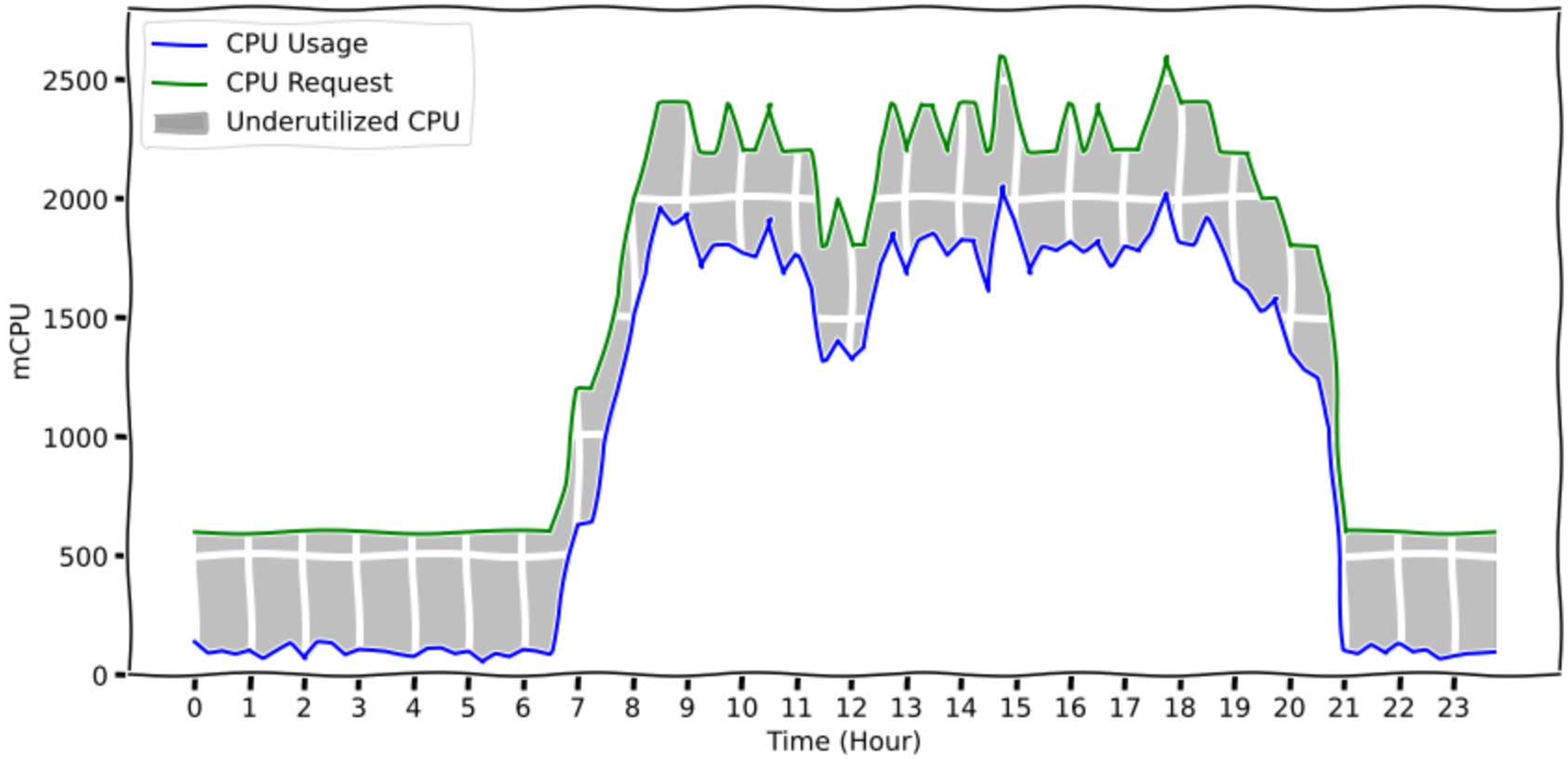
  - **“Autoscaling in Kubernetes – From Zero to Hero” – 16:00 Raum 1**

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: graal-demo
  #...
spec:
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - resource:
        name: cpu
        target:
          averageUtilization: 80
          type: Utilization
      type: Resource
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: graal-demo
```


MLP

Capacity Planning

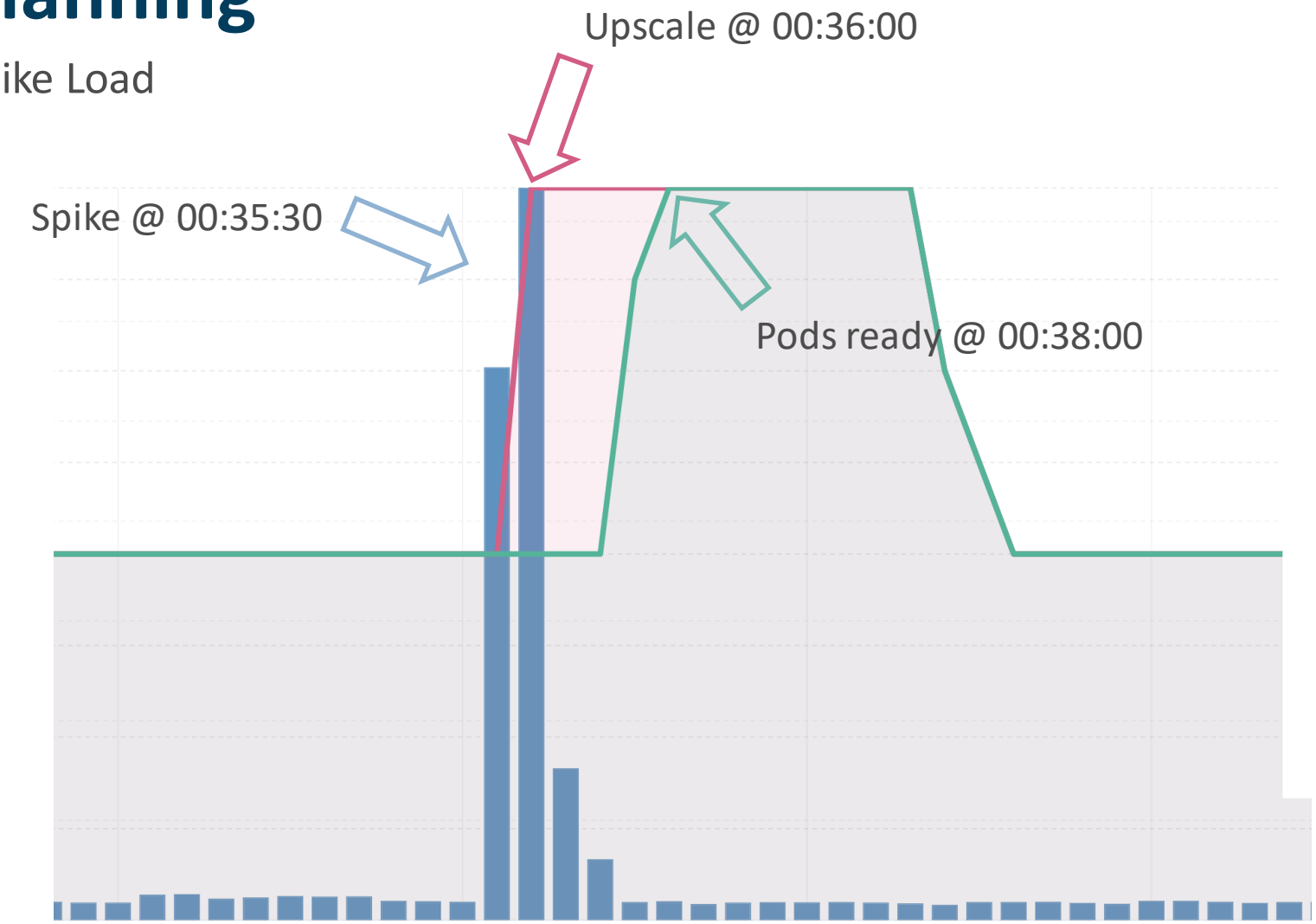
Better, but not perfect!



MLP

Capacity Planning

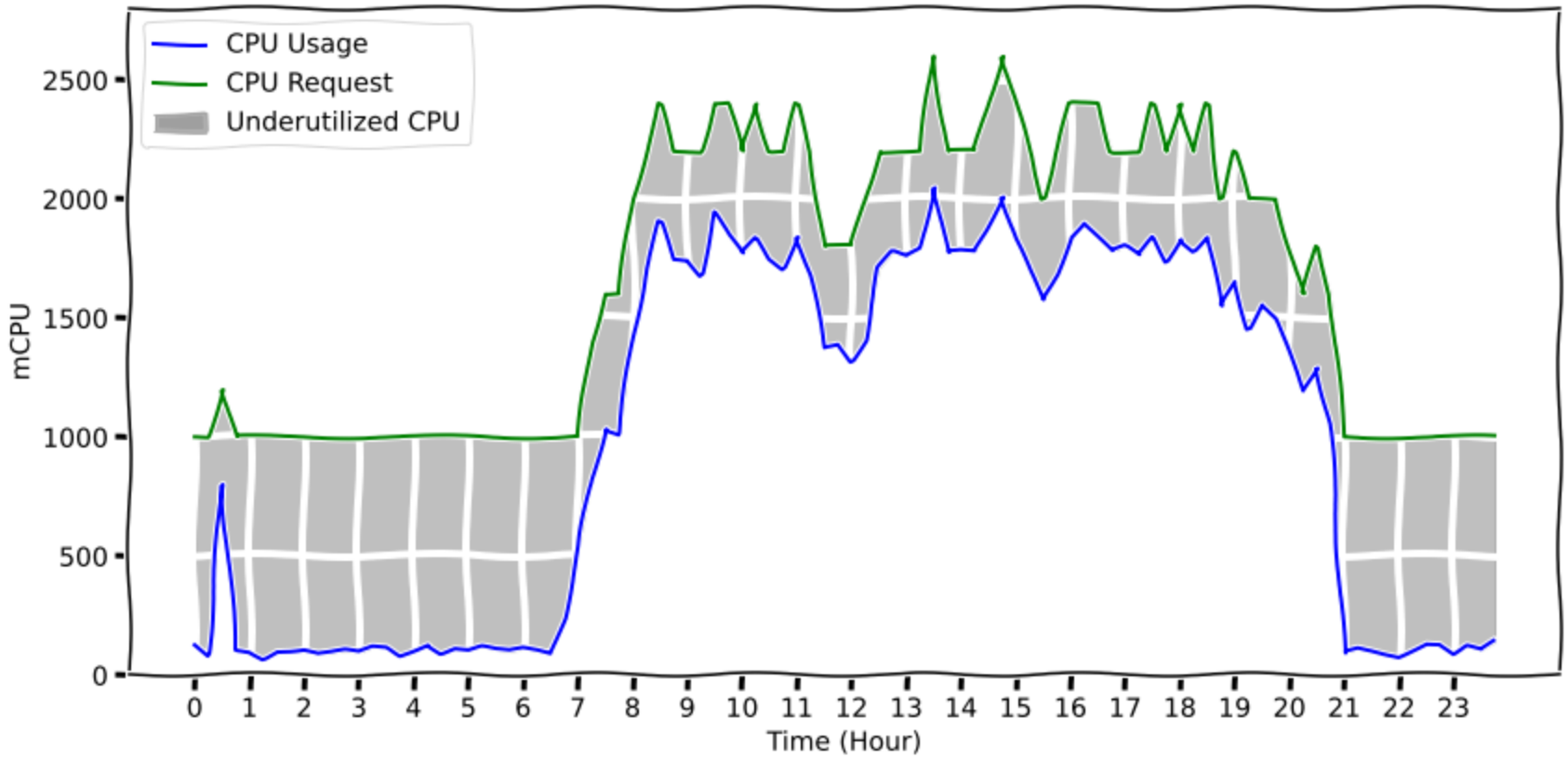
The ugly details: Spike Load



MLP

Capacity Planning

The Reality!



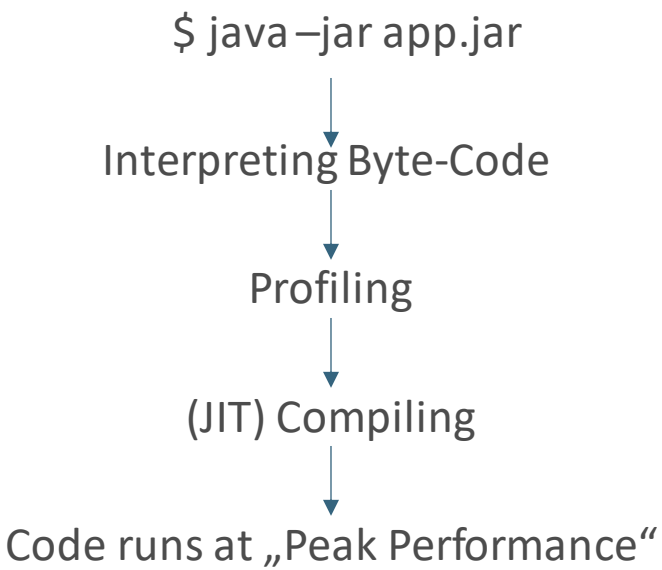
MLP

Whats the Problem?

- Autoscaling still underutilizes ressources
- Time to scale leads to underutilizes ressources
- Spring Boot services on the JVM have very high startup times

Why are our Spring Boot Services so slow at startup?

Just-In-Time Compiling!



MLP

02

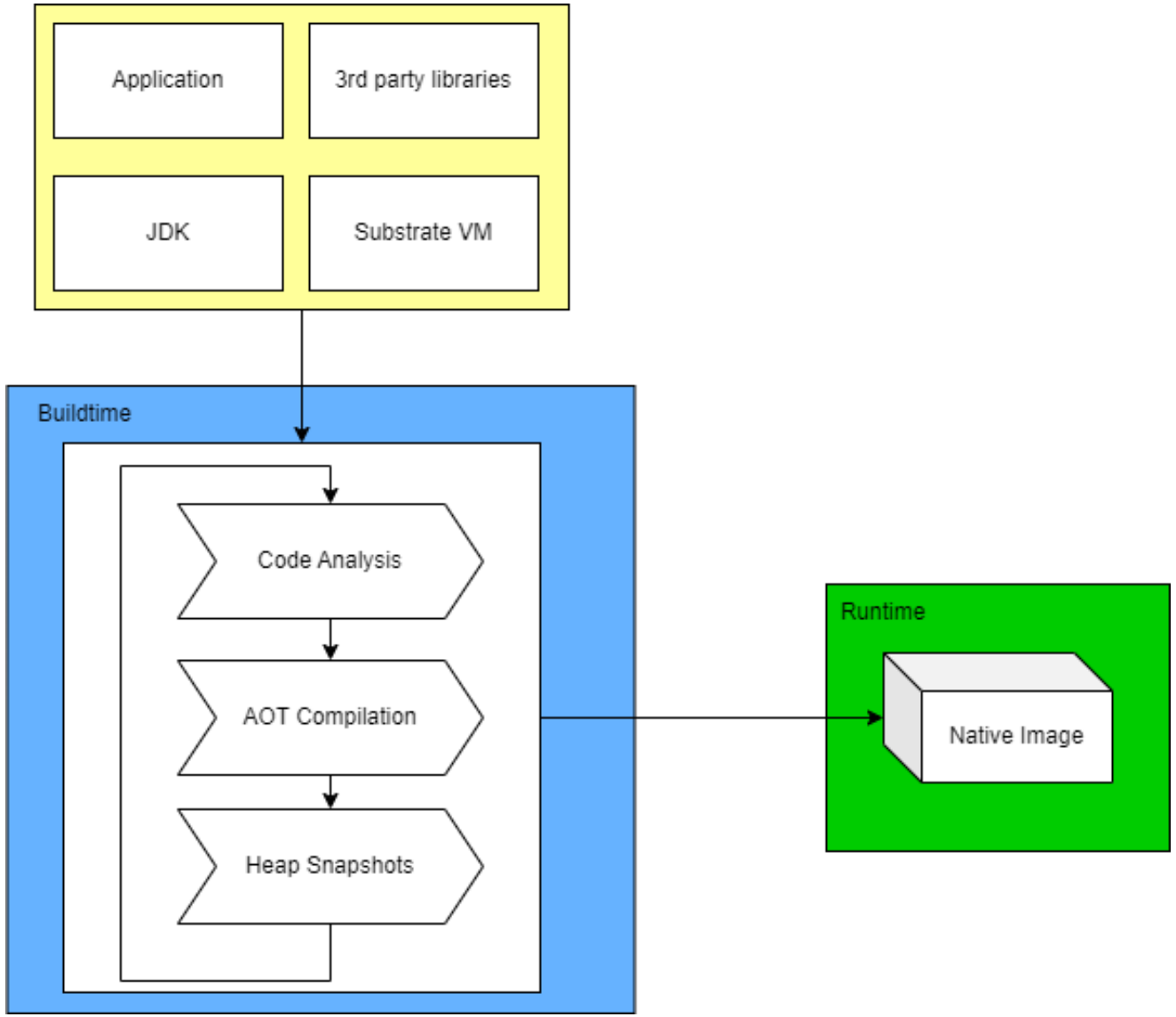
GraalVM

Native Images

GraalVM

- High performance JDK von Oracle
- Origin: Research project 2011
- Polyglot-Support
- Native Image Compilations

GraalVM – Native Image Build



MLP

Native Images

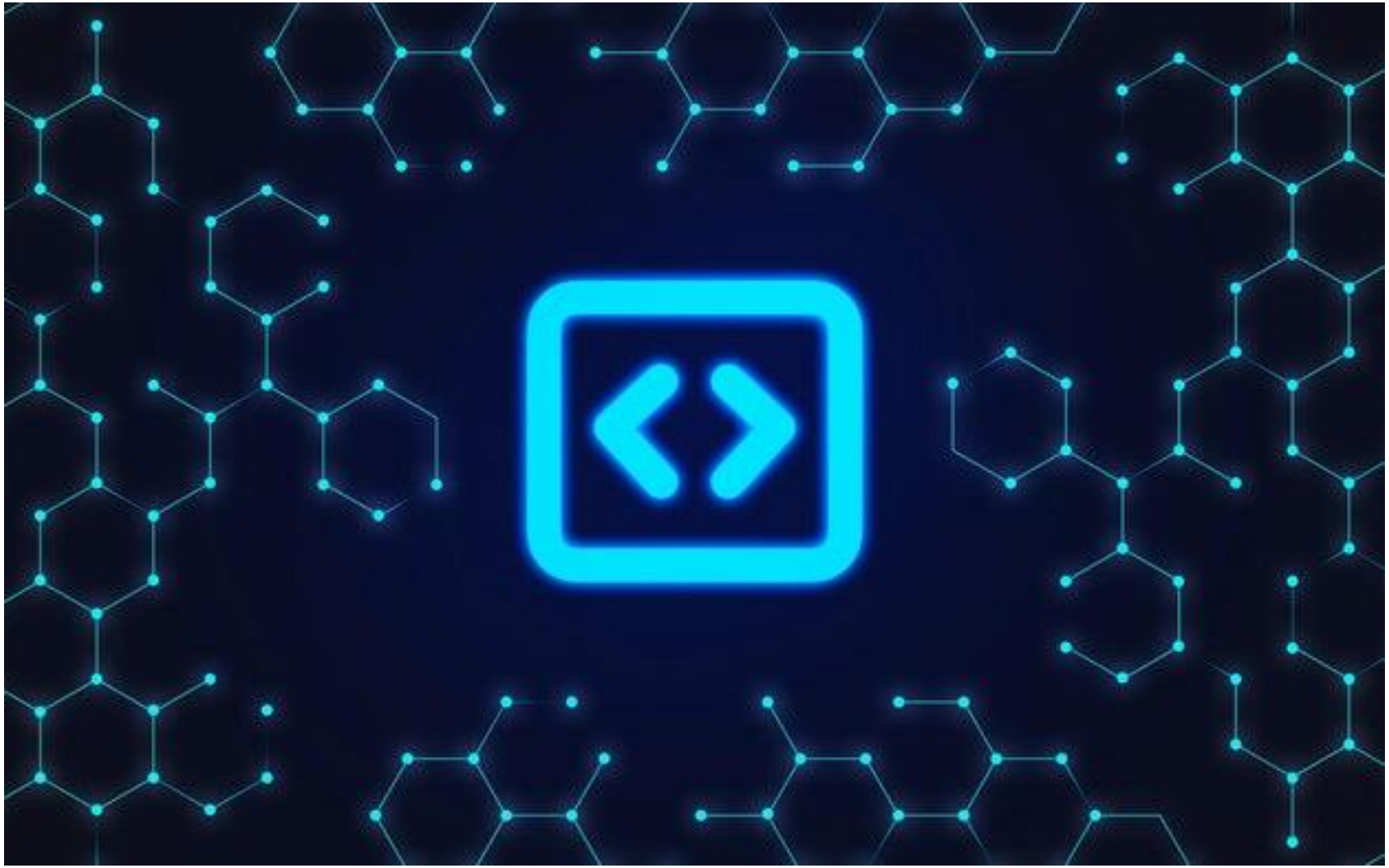
- AOT <-> JIT
- Minimized startup time
- Problem: Dynamic (runtime) features
- Spring-Boot 3: Support 11-2022

03

Introduction: Time for code

Demo

Demo – Hands-On



MLP

Demo

Build & Execution

- Build-Definition & AOT-Sources
- Conditional Beans & Profiles
- Runtime Hints
- Tests / Debugging
- Recommendation: AOT-Mode!

04

Performance

Performance

	JVM	GraalVM
Startup Time per Pod*	51 Sec	**3 Sec**
max CPU Usage per Pod	500m	500m
max Memory Usage per Pod	280 MB	**118 MB**
Container-Image Size	183 MB (Jar 44 MB)	235 MB (Binary 134 MB)
Max Throughput	**2.569 RPM**	1.925 RPM
Build-Time	**~15 Sec**	~16 Min

* from „Scheduled“ until „Ready“

Performance

- Memory Limit/Request: 512mi
- CPU Limit/Request: 500m (0.5 CPU)
- Autoscaling Target: 80% CPU
- Spring-Boot Version: 3.1.3
- JVM: OpenJDK 64-Bit Server VM Temurin-17.0.8.1+1
- GraalVM: GraalVM CE 17.0.8+7.1 (build 17.0.8+7-jvmci-23.0-b15)
- Spiketest: 5 Min

05

Conclusion

Pitfalls

- Initial Setup
- Runtime-Errors
- Build-Time: AOT / Native
- Multi-Maven-Module Projects
- Library Support

Ease of Development

Drop-In	Easy-to-use	Not so easy	Problematic
Actuator	Conditionals	Hibernate	Multiple Data Sources
Generated Classes	Native Tests	Spring Data JPA	Multi-Maven-Modules
Profiles	PathVariables		
Prometheus	Ressourcen		
Serialization / Deserialization			
Spring MVC			

Conclusion

It depends!

- Memory-Footprint down
- Scalability up
- Moderate pain to use

Vielen Dank!

- Kontakt:
  - Dominik.schiessl@mlp.de
  - Hendrik.still@mlp.de

- Slides & Code:
  - https://github.com/MLPschiessl/graal-demo

MLP