HAW HAMBURG

BACHELORTHESIS
Kosai Alzaeim

Quarkus vs. Spring Boot: Effizienz und Ressourcenverbrauch bei nativen Builds, Containerisierung und Skalierbarkeit

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Kosai Alzaeim

Vergleich von Quarkus und Spring Boot hinsichtlich Effizienz und Ressourcenverbrauch bei nativen Builds, Containerisierung und Skalierbarkeit

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Lars Hamann

Eingereicht am: 23. April 2025

Kosai Alzaeim

Thema der Arbeit

Vergleich von Quarkus und Spring Boot im Kontext eines Promotion-Management-Systems

Stichworte

Java, Quarkus, Spring Boot, Microservices, Performance, Skalierbarkeit, Ressourcenverbrauch, Cloudanwendung, Deployment, DevOps

Kurzzusammenfassung

Im Rahmen dieser Bachelorarbeit werden die Java-Frameworks Quarkus und Spring Boot anhand eines eigens entwickelten Microservice-basierten Promotion-Management-Systems systematisch miteinander verglichen. Ziel ist es, Unterschiede hinsichtlich Performance, Ressourcenverbrauch und Skalierbarkeit unter realitätsnahen Lastbedingungen zu identifizieren. Die Evaluation erfolgt mithilfe von Docker-Containern, Prometheus und Grafana sowie Lasttests mit Apache JMeter. Die Ergebnisse zeigen, dass Quarkus insbesondere in Bezug auf Startzeit, CPU- und RAM-Verbrauch Vorteile gegenüber Spring Boot aufweist, während Spring Boot unter extremer Last eine etwas höhere Stabilität zeigt. Die Arbeit bietet somit eine praxisorientierte Entscheidungsgrundlage für die Wahl des geeigneten Frameworks in modernen Cloud-nativen Anwendungen.

Kosai Alzaeim

Title of Thesis

Comparison of Quarkus and Spring Boot in the Context of a Promotion Management System

Keywords

Quarkus, Spring Boot, Microservices, Performance, Scalability, Resource Usage, Cloud-Services, Deployment, DevOps

Abstract

This bachelor thesis presents a systematic comparison of the Java frameworks Quarkus and Spring Boot using a self-developed microservice-based promotion management system. The goal is to analyze differences in performance, resource consumption, and scalability under realistic load conditions. The evaluation was conducted using Docker containers, Prometheus and Grafana monitoring, as well as load testing with Apache JMeter. The results indicate that Quarkus provides advantages in startup time, CPU and RAM usage, whereas Spring Boot demonstrates slightly higher stability under extreme load. This work offers a practice-oriented basis for selecting a suitable framework for modern cloud-native applications.

Inhaltsverzeichnis

Inhaltsverzeichnis ..... **v**

Abbildungsverzeichnis ..... **viii**

Tabellenverzeichnis ..... **x**

1 Einleitung **1**

1.1 Motivation und Hintergrund 1

1.2 Zielsetzung und Forschungsfrage 3

1.3 Aufbau 4

2 Theoretischer Hintergrund **6**

2.1 Spring 6

2.1.1 Historie und Entwicklung 6

2.1.2 Architektur und Kernkomponenten 7

2.1.3 Dependency Injection (DI) in Spring 8

2.1.4 Spring MVC – Webentwicklung mit Spring 9

2.2 Spring Boot 10

2.1.1 Historie und Entwicklung 10

2.1.2 Hauptfunktionen von Spring Boot 11

2.1.3 Vergleich: Spring vs. Spring Boot 12

2.1.4 Einsatz 13

2.2 Quarkus 14

2.2.1 Historie und Entwicklung 14

2.2.2 Architektur und Kernkomponenten 15

2.2.3 Hauptmerkmale von Quarkus 17

2.3 Empirische Vergleichsstudien zu Quarkus und Spring Boot 18

3 Methodik **21**

3.1 Zielsetzung des Vergleichs 21
3.2 Vergleichskriterien und Metriken 22
3.2.1 Buildzeit 22
3.2.2 Startzeit 22
3.2.3 Imagegröße 22
3.2.4 Antwortzeit (Latenz) 23
3.2.5 Durchsatz (Requests pro Sekunde) 23
3.2.6 Ressourcenverbrauch 24
3.2.7 Skalierbarkeit 24
3.3 Versuchsaufbau (Testumgebung) 24
3.4 Testdurchführung (Testplan) 27
4 Implementierung 29
4.1 Beschreibung des Promotion-Management-Systems 29
4.1.1 Kurze Einführung in die Domäne: 29
4.1.2 Begründung für die Auswahl als Vergleichssystem: 30
4.2 Kontextsicht des Systems 31
4.2.1 Fachliche Kontextsicht: 31
4.2.2 Technische Kontextsicht: 32
4.3 Bausteinsicht 32
4.4 Laufzeitsicht 34
5 Analyse, Vergleich und Bewertung der Ergebnisse 35
5.1 Vergleich grundlegender Metriken (Store-Service) 35
5.1.1 Buildzeit-Vergleich 35
5.1.2 Startzeit-Vergleich 36
5.1.3 Imagegröße-Vergleich 37
5.2 Performanzvergleich 38
5.2.1 Anzahl der Samples 39
5.2.2 Antwortzeit (Response Time) 39
5.2.3 Durchsatz und Stabilität 41
5.3 Ressourcenverbrauch 43
5.3.1 CPU-Verbrauch 43

5.3.2 Speicherverbrauch 45
5.4 Skalierung 46
5.4.1 Anzahl der Samples nach der Skalierung 47
5.4.2 Antwortzeit nach der Skalierung 48
5.4.3 Durchsatz und Fehlerquote nach der Skalierung 50
5.4.4 Ressourcenverbrauch nach der Skalierung 52
6 Zusammenfassung und Ausblick 55
Literaturverzeichnis 57

Abbildungsverzeichnis

Abbildung 1: Kernkomponenten von Spring-Framework (Spring, 2025) 8
Abbildung 2: Spring IoC Container als Object Factory (Geeksforgeeks, 2025) 9
Abbildung 3: MVC in Spring (Spring, 2025) 10
Abbildung 4: Das Spring Boot-Ökosystem auf einen Blick (Mitropolitsky, et al., 2019) 12
Abbildung 5: Quarkus Architektur und Kernkomponenten (Štefanko & Martiška, 2025) 17
Abbildung 6: Container-First Ansatz von Quarkus (Quarkus, 2025) 18
Abbildung 7: Spring Boot vs. Quarkus vs. Micronaut (Ter, 2024) 19
Abbildung 8: Vergleichskriterien und Metriken 23
Abbildung 9: Software und Hardware der Testumgebung 26
Abbildung 10: Promotion-Management-System Workflow 30
Abbildung 11: Fachliche Kontextsicht 31
Abbildung 12: Technische Kontextsicht 32
Abbildung 13: Bausteinsicht des Promotion-Management-Systems 33
Abbildung 14: Laufzeitsicht Beispielprozess (Angebot hinzufügen) 34
Abbildung 15: Buildzeit-Vergleich (Store-Service) 36
Abbildung 16: Startzeit-Vergleich (Store-Service) 37
Abbildung 17: Imagegröße-Vergleich (Store-Service) 38
Abbildung 18: Vergleich typischer Antwortzeiten zwischen Quarkus und Spring Boot 40
Abbildung 19: Minimale und maximale Antwortzeiten (ms) unter Last 41
Abbildung 20: Durchsatzvergleich unter Lastbedingungen 42
Abbildung 21: JMeter - Summary-Report Testplan 3 Quarkus 43

Abbildung 22: CPU-Verbrauchsvergleich 44

Abbildung 23: RAM-Verbrauch 46

Abbildung 24: Anzahl der Samples nach der Skalierung..... 48

Abbildung 25: Speedup der Antwortzeiten nach der Skalierung 49

Abbildung 26: Durchsatz- und Fehlerquotevergleich nach der Skalierung 51

Abbildung 27: Relative Änderung des Ressourcenverbrauchs nach Skalierung 53

Tabellenverzeichnis

Tabelle 1: Spring Boot Version-Übersicht 11
Tabelle 2: Vergleich Spring Boot vs. Spring 14
Tabelle 3: Übersicht der Testpläne für die Lasttests 28
Tabelle 4: Anzahl der verarbeiteten Anfragen 39

1 Einleitung

1.1 Motivation und Hintergrund

Die fortschreitende Digitalisierung und die steigenden Anforderungen an moderne Softwareanwendungen haben zu einem Wandel in der Softwareentwicklung geführt. Monolithische Architekturen weichen zunehmend Microservice-Architekturen, die eine höhere Flexibilität, Skalierbarkeit und Wartbarkeit bieten. In diesem Kontext spielen Java-basierte Frameworks eine entscheidende Rolle, da Java nach wie vor eine der meistgenutzten Programmiersprachen in der Unternehmenswelt ist.

Spring Boot hat sich in den letzten Jahren als De-facto-Standard für die Entwicklung von Microservices etabliert und bietet eine umfassende Infrastruktur sowie ein reichhaltiges Ökosystem, das Entwicklern ermöglicht, schnell produktionsreife Anwendungen zu erstellen. Dennoch stehen Spring-Boot-Anwendungen häufig vor Herausforderungen wie langen Startzeiten und hohem Ressourcenverbrauch, insbesondere in Cloud- und Container-Umgebungen, in denen Effizienz und Skalierbarkeit entscheidend sind.

Mit dem Aufkommen von Cloud-nativen Technologien und der zunehmenden Bedeutung von Containern und orchestrierten Umgebungen wie Kubernetes wächst der Bedarf an leistungsfähigeren und ressourcenschonenderen Alternativen. Quarkus wurde speziell für Cloud-native Anwendungen entwickelt und verspricht durch seine schnelle Startzeit und den geringen Speicherverbrauch deutliche Vorteile gegenüber Spring Boot. Durch die Unterstützung der nativen Kompilierung mit GraalVM können Anwendungen als native Executables bereitgestellt werden, was die Performance weiter optimiert und den Ressourcenbedarf minimiert.

Die Wahl des richtigen Frameworks hat erhebliche Auswirkungen auf die Effizienz, Skalierbarkeit und Kosten von Anwendungen im Produktivbetrieb. Während Spring Boot eine ausgereifte und bewährte Plattform bietet, könnten die Performance-Vorteile von Quarkus in bestimmten Szenarien zu erheblichen Verbesserungen führen. Entwickler und Unternehmen stehen daher vor der Herausforderung, abzuwägen, welches Framework besser zu ihren spezifischen Anforderungen passt.

Obwohl die theoretischen Unterschiede zwischen Quarkus und Spring Boot bekannt sind, fehlen bislang umfassende praxisnahe Studien, die einen direkten Vergleich unter realen Bedingungen ermöglichen. Insbesondere gibt es nur wenige Analysen, die Startzeit, Ressourcenverbrauch, Container-Größe und Skalierbarkeit systematisch untersuchen.

Vor diesem Hintergrund ist es von besonderem Interesse, eine RESTful Microservice-Anwendung für das Promotion-Management-System zu implementieren und beide Frameworks anhand praxisrelevanter Kriterien zu evaluieren. Dieses Fallbeispiel bildet typische Anforderungen moderner Anwendungen ab und bietet ausreichend Komplexität für aussagekräftige Messungen. Ziel dieser Arbeit ist es, eine fundierte Entscheidungsgrundlage für Entwickler und Unternehmen bereitzustellen, um die Stärken und Schwächen von Quarkus und Spring Boot in Cloud-nativen Szenarien objektiv bewerten zu können.

1.2 Zielsetzung und Forschungsfrage

Diese Arbeit verfolgt das Ziel, die beiden Java-basierten Frameworks Quarkus und Spring Boot hinsichtlich ihrer Effizienz und ihres Ressourcenverbrauchs zu vergleichen. Dabei sollen praxisnahe Erkenntnisse gewonnen werden, die Entwicklern und Unternehmen eine fundierte Entscheidungsgrundlage für die Auswahl des passenden Frameworks bieten. Der Fokus liegt auf der Bewertung der Performance in Cloud-nativen Umgebungen, insbesondere in Bezug auf schnelle Startzeiten, geringen Ressourcenverbrauch und effiziente Skalierbarkeit.

Zur Untersuchung dieser Aspekte wird eine RESTful Microservice-Anwendung für das Promotion-Management-System entwickelt. Diese dient als Testfall, um beide Frameworks unter realistischen Bedingungen zu vergleichen. Die Analyse konzentriert sich auf Startup-Zeit, Ressourcenverbrauch, Container-Größe, Skalierbarkeit und Entwicklererfahrung, um ein umfassendes Bild der jeweiligen Stärken und Schwächen zu zeichnen. Ziel ist es, herauszufinden, in welchen Szenarien Quarkus signifikante Vorteile gegenüber Spring Boot bietet und ob es Anwendungsfälle gibt, in denen Spring Boot die bessere Wahl ist.

Die zentrale Forschungsfrage dieser Arbeit lautet: *"In welchen Bereichen zeigt Quarkus im Vergleich zu Spring Boot signifikante Vorteile hinsichtlich Effizienz und Ressourcenverbrauch, insbesondere in Bezug auf den nativen Build-Prozess, die Containerisierung und die Skalierbarkeit in Containerumgebungen?"*

Diese Frage wird anhand folgender Kernaspekte analysiert:

- **Nativen Build-Prozess:** Untersuchung der Effizienz nativer Images hinsichtlich Startzeit, Ressourcenverbrauch und Anwendungsleistung.
- **Containerisierung:** Bewertung der Eignung beider Frameworks für containerisierte Umgebungen, insbesondere hinsichtlich Container-Größe und Ressourcennutzung.
- **Skalierbarkeit:** Analyse der Performance unter Last und der Effizienz der horizontalen Skalierung.

Durch die systematische Analyse dieser Kriterien sollen praxisrelevante Erkenntnisse gewonnen werden, die Entwickler und Unternehmen bei der Wahl des optimalen Frameworks unterstützen. Die Ergebnisse bieten eine objektive Grundlage für den Einsatz von Quarkus oder Spring Boot in modernen Cloud- und Container-Umgebungen.

1.3 Aufbau

Kapitel 1 – Einleitung:

Dieses Kapitel führt in die Problemstellung ein, beschreibt die Zielsetzung der Arbeit und formuliert die zentrale Forschungsfrage. Zudem wird ein Überblick über den weiteren Aufbau der Arbeit gegeben.

Kapitel 2 – Theoretischer Hintergrund:

In diesem Kapitel erfolgt eine detaillierte Vorstellung der beiden Frameworks:

- **Spring & Spring Boot:** Architektur und Kernfeatures – Analyse der grundlegenden Konzepte und Funktionsweise von Spring Boot.
- **Quarkus:** Architektur und Kernfeatures – Vorstellung der wesentlichen Eigenschaften und Designprinzipien von Quarkus.
- **Empirische Vergleichsstudien zu Quarkus und Spring Boot** – Überblick über bestehende Forschungsarbeiten und vergleichende Analysen der beiden Frameworks.

Kapitel 3 – Methodik:

Hier wird das methodische Vorgehen beschrieben, das zur Beantwortung der Forschungsfrage herangezogen wird. Es werden die Kriterien für den Vergleich definiert und die Testmethoden erläutert. Zudem wird auf die Entwicklung und Testanforderungen eingegangen, die für die praktische Umsetzung der Experimente relevant sind.

Kapitel 4 – Implementierung:

In diesem Kapitel wird die Umsetzung der RESTful Microservice-Anwendung für das Promotion-Management-System beschrieben. Der Aufbau der Testanwendungen in beiden Frameworks wird detailliert erläutert, einschließlich der Architektur und der verwendeten Technologien.

Kapitel 5 – Analyse, Vergleich und Bewertung der Ergebnisse:

Hier werden die Testergebnisse systematisch analysiert und verglichen. Der Fokus liegt auf den zentralen Leistungsmerkmalen:

- **Performance:** Messung der Startzeiten, Buildzeiten und Laufzeiteffizienz.
- **Ressourcennutzung:** Untersuchung des Speicher- und CPU-Verbrauchs.
- **Skalierbarkeit:** Bewertung der Frameworks unter Lastbedingungen und ihre Eignung für dynamische Skalierung.

Die gewonnenen Ergebnisse werden entlang der definierten Metriken analysiert, gegenübergestellt und im Hinblick auf die Leistungsfähigkeit beider Frameworks eingeordnet.

Kapitel 6 – Zusammenfassung und Ausblick:

Das abschließende Kapitel fasst die wichtigsten Erkenntnisse zusammen und gibt einen Ausblick auf zukünftige Forschungsmöglichkeiten.

2 Theoretischer Hintergrund

Dieses Kapitel widmet sich der detaillierten Vorstellung der beiden Java-basierten Frameworks Quarkus und Spring Boot, die im Rahmen dieser Arbeit miteinander verglichen werden. Dabei werden jeweils die zugrunde liegenden Architekturansätze, charakteristischen Eigenschaften sowie zentrale Designprinzipien erläutert. Darüber hinaus bietet das Kapitel einen Überblick über bestehende empirische Studien und vergleichende Analysen, die sich mit der Leistungsfähigkeit und Ressourcennutzung beider Frameworks beschäftigen und somit die Basis für die eigene experimentelle Untersuchung bilden.

2.1 Spring

2.1.1 Historie und Entwicklung

Das Spring-Framework ist eines der bedeutendsten und am weitesten verbreiteten Frameworks in der Java-Welt. Entwickelt wurde es Anfang der 2000er Jahre von Rod Johnson, der seine Ideen erstmals 2002 in seinem Buch "Expert One-on-One J2EE Design and Development" vorstellte (Spring, kein Datum). Ziel war es, eine leichtgewichtige Alternative zu Java EE (heute Jakarta EE) zu schaffen, da Enterprise JavaBeans (EJB) als zu komplex und schwergewichtig empfunden wurden.

Die erste stabile Version von Spring (1.0) erschien 2004 unter der Apache-2.0-Lizenz. Durch seine modulare Architektur, die flexible Integration mit verschiedenen Technologien und die Möglichkeit, plattformunabhängige Anwendungen zu entwickeln, gewann Spring schnell an Beliebtheit. In den folgenden Jahren wurde das Framework kontinuierlich weiterentwickelt. Es fand breite Unterstützung in der Open-Source-Community und wurde von verschiedenen Unternehmen in großem Umfang eingesetzt. 2009 wurde das Unternehmen hinter Spring (SpringSource) von VMware übernommen, und heute wird es als Teil der VMware Tanzu-Initiative weitergeführt.

2.1.2 Architektur und Kernkomponenten

Spring basiert auf einer modularen Architektur, die es Entwicklern ermöglicht, nur die benötigten Komponenten einzusetzen. Die wichtigsten Module des Spring-Frameworks sind:

1. **Core Container:** Enthält grundlegende Module wie Spring-Core, Spring-Beans und Spring-Context, die für Dependency Injection (DI) und das ApplicationContext-Management verantwortlich sind.
2. **Datenzugriff und Integration:** Beinhaltet Spring-JDBC, Spring-ORM (Integration mit Hibernate, JPA), Spring-TX für Transaktionsmanagement und Spring-Data für eine vereinfachte Datenbankanbindung.
3. **Web-Modul:** Besteht aus Spring-Web und Spring-MVC, die den Aufbau von Webanwendungen und RESTful-Services erleichtern.
4. **AOP (Aspect-Oriented Programming):** Ermöglicht die Trennung von Querschnittsbelangen wie Logging oder Security durch das Spring-AOP-Modul.
5. **Messaging und Events:** Bietet Unterstützung für asynchrone Kommunikation über Spring-Messaging (z. B. RabbitMQ, Kafka) und WebSockets.

6. **Testing:** Stellt Module für Unit-Tests und Integrationstests bereit, z. B. mit JUnit und Mockito.

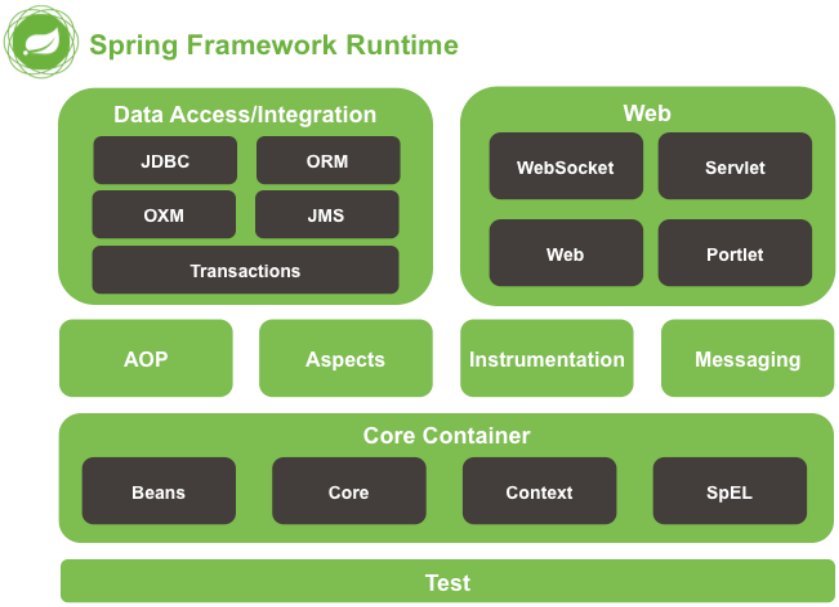


Abbildung 1: Kernkomponenten von Spring-Framework (Spring, 2025)

2.1.3 Dependency Injection (DI) in Spring

Ein zentrales Konzept von Spring ist die Inversion of Control (IoC), die über Dependency Injection realisiert wird. Dadurch wird die Erstellung und Verwaltung von Objekten an den Spring-Container delegiert.

Es gibt zwei Haupttypen des Spring Containers:

BeanFactory:	Eine grundlegende Implementierung für einfache Anwendungen.
Application-Context:	Eine leistungsfähigere Variante mit erweiterten Funktionen wie Event-Handling und Internationalisierung.

Die Konfiguration kann über XML, Java-Config oder Annotations **(@Component, @Service, @Repository**) erfolgen. DI fördert lose Kopplung und erleichtert das Testen.

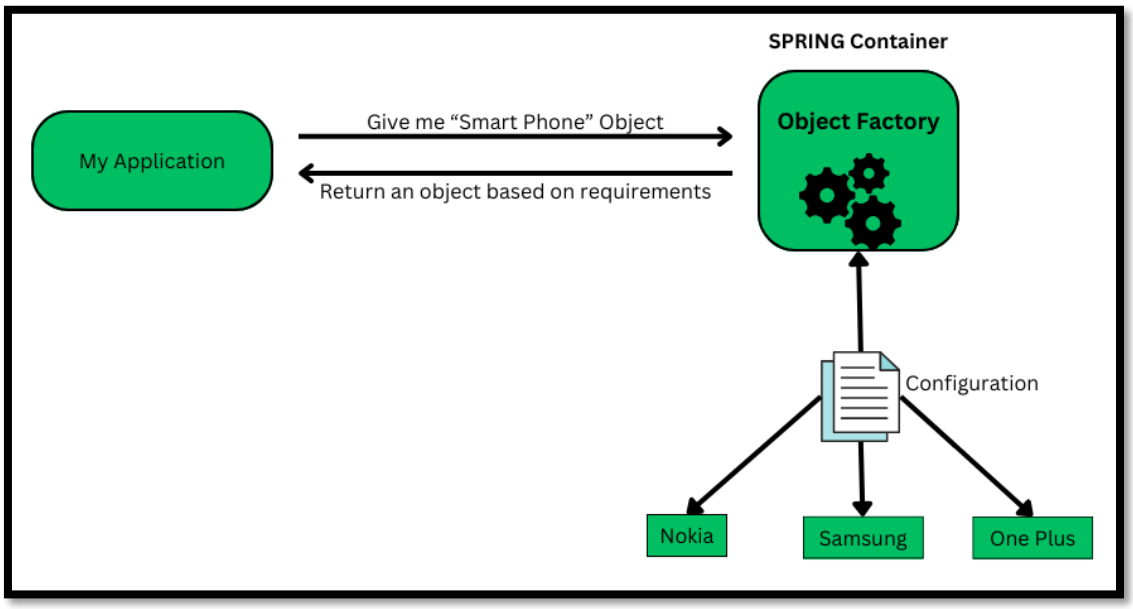


Abbildung 2: Spring IoC Container als Object Factory (Geeksforgeeks, 2025)

2.1.4 Spring MVC – Webentwicklung mit Spring

Spring MVC ist ein flexibles Web-Framework zur Erstellung von Webanwendungen und REST-APIs. Es basiert auf dem Model-View-Controller (MVC)-Prinzip und verwendet einen zentralen DispatcherServlet. Hauptkomponenten sind:

- **Controller (@RestController):** Verarbeiten HTTP-Anfragen.
- **Model:** Daten, die an die View weitergegeben werden.
- **View:** Präsentationsschicht (z. B. Thymeleaf, JSP, JSON-Ausgabe für APIs).

Spring MVC unterstützt verschiedene View-Technologien und bietet eine integrierte Validierung, Form-Handling und Internationalisierung.

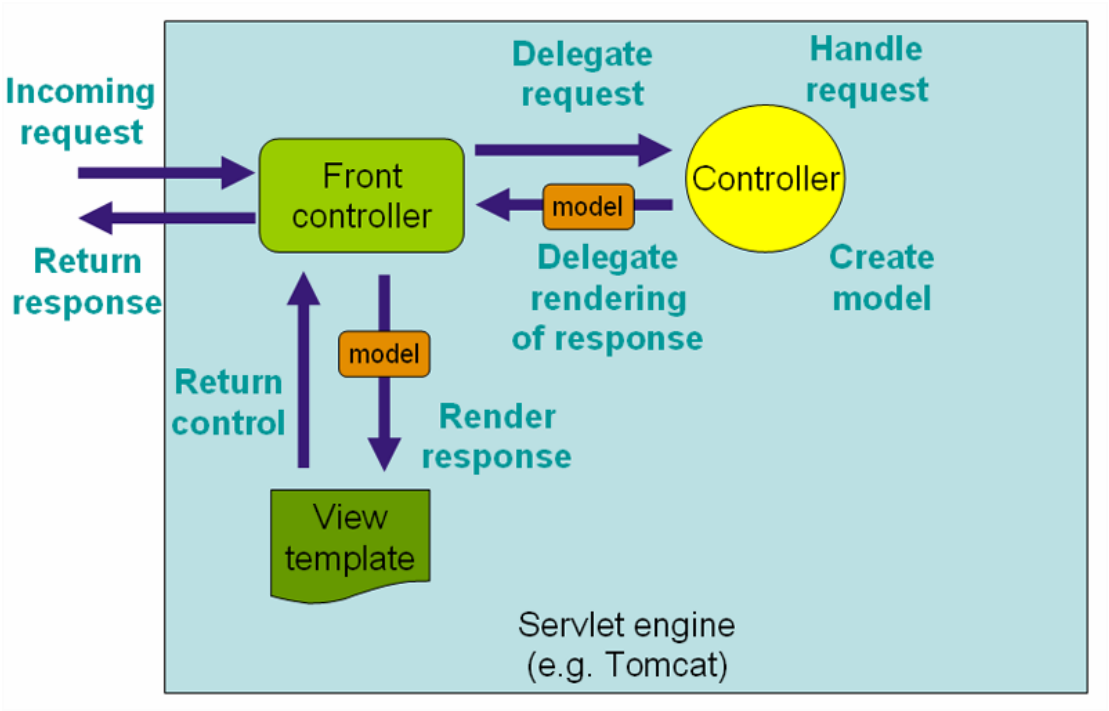


Abbildung 3: MVC in Spring (Spring, 2025)

2.2 Spring Boot

2.1.1 Historie und Entwicklung

Spring Boot ist eine Weiterentwicklung des Spring-Ökosystems, die darauf abzielt, die Entwicklung und Konfiguration von Spring-Anwendungen erheblich zu vereinfachen. Es bietet zahlreiche Funktionen, die Entwicklern helfen, produktive und skalierbare Anwendungen mit minimalem Konfigurationsaufwand zu erstellen. Anstatt zahlreiche XML- oder Java-Konfigurationsdateien manuell zu definieren, übernimmt Spring Boot viele dieser Aufgaben automatisch, was die Entwicklungszeit verkürzt und den Einstieg erleichtert.

Spring Boot wurde erstmals im April 2014 veröffentlicht und hat sich seitdem kontinuierlich weiterentwickelt. Die folgende Tabelle gibt einen Überblick über die Hauptversionen und ihre Veröffentlichungsdaten (Minh, n.d.) :

Version	Veröffentlichungsdatum	Spring Framework Version
1.0.0	April 2014	4.0.3
2.0.0	März 2018	5.0.4
3.0.0	November 2022	6.0.2
3.4.2	23. Januar 2025	6.2.2

Tabelle 1: Spring Boot Version-Übersicht

2.1.2 Hauptfunktionen von Spring Boot

Spring Boot vereinfacht die Entwicklung von Spring-Anwendungen durch mehrere zentrale Funktionen:

Auto-Configuration: Spring Boot erkennt automatisch, welche Bibliotheken und Abhängigkeiten im Projekt vorhanden sind, und konfiguriert diese entsprechend. Dadurch entfällt der Aufwand, viele Einstellungen manuell vornehmen zu müssen.

Standalone-Anwendungen: Anwendungen können eigenständig ausgeführt werden, ohne dass ein separater Webserver installiert werden muss. Spring Boot nutzt eingebettete Server wie Tomcat, Jetty oder Undertow, sodass die Anwendung als ausführbare JAR-Datei gestartet werden kann.

Meinungsbasierte Defaults: Spring Boot stellt eine Reihe von Standardkonfigurationen bereit, die für viele Anwendungen geeignet sind. Entwickler können diese Vorgaben übernehmen oder bei Bedarf anpassen.

Spring Boot Actuator: Ermöglicht das Monitoring und die Verwaltung von Anwendungen durch vorkonfigurierte Endpunkte, die Informationen zur Systemgesundheit, Metriken und laufenden Prozessen liefern.

Die Grafik “Abbildung 4: Das Spring Boot-Ökosystem auf einen Blick” veranschaulicht die Kernkomponenten und den Nutzen von Spring Boot innerhalb des Spring-Ökosystems. Spring Boot baut auf Spring Core, Spring Data und Spring MVC auf und erleichtert die Entwicklung

durch Funktionen wie Auto-Configuration, eingebettete Server und Actuator-Endpunkte zur Überwachung.

Zusätzlich integriert Spring Boot externe Dienste wie Datenbanken, Logging- und Messaging-Systeme sowie Cloud-Provider, was eine flexible und skalierbare Entwicklung ermöglicht. Entwickler profitieren von einer optimierten Konfiguration, schnelleren Entwicklungszyklen und einer nahtlosen Bereitstellung über Container-Technologien.

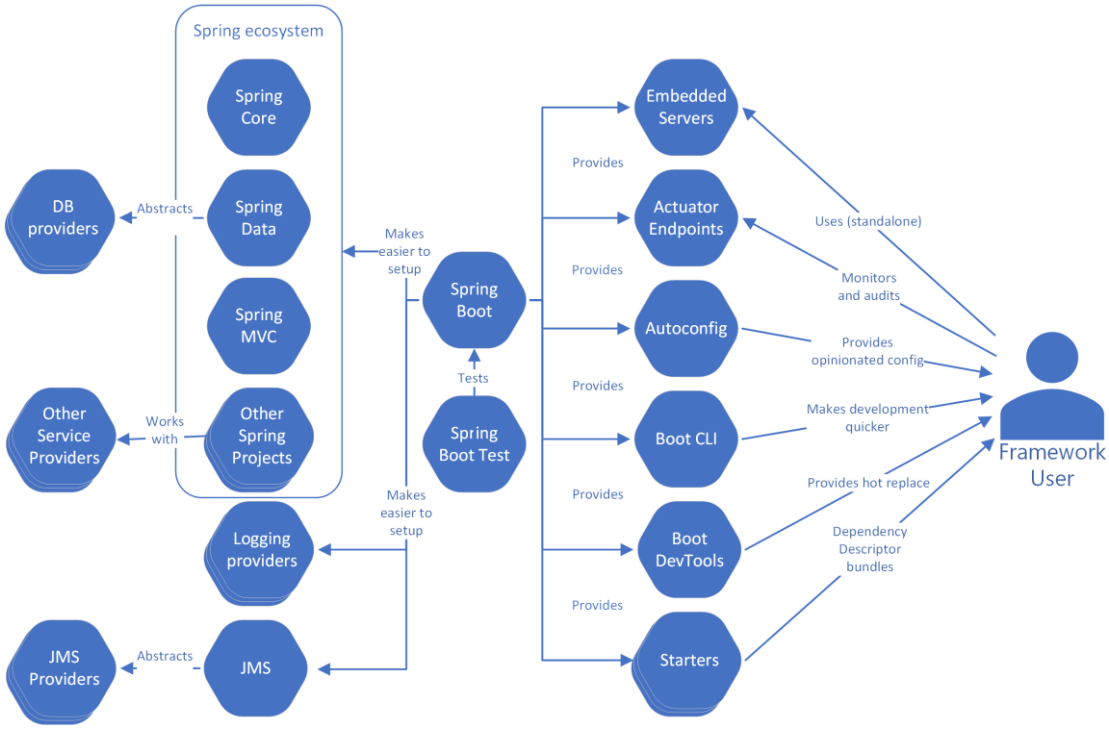


Abbildung 4: Das Spring Boot-Ökosystem auf einen Blick (Mitropolitsky, et al., 2019)

2.1.3 Vergleich: Spring vs. Spring Boot

Spring und Spring Boot sind eng miteinander verbunden, unterscheiden sich jedoch in mehreren wichtigen Aspekten:

- **Konfiguration:** Während das klassische Spring-Framework eine manuelle Konfiguration erfordert, übernimmt Spring Boot viele Konfigurationsaufgaben automatisch, wodurch Entwickler sich mehr auf die Geschäftslogik konzentrieren können.
- **Projektsetup:** Die Einrichtung eines Spring-Projekts kann aufwendig sein, da viele Abhängigkeiten manuell verwaltet werden müssen. Spring Boot bietet hingegen mit Spring Initializr ein Tool, das das Erstellen neuer Projekte mit vorkonfigurierten Abhängigkeiten erleichtert.
- **Flexibilität vs. Konvention:** Spring bietet eine hohe Flexibilität und kann genau an spezifische Anforderungen angepasst werden. Spring Boot hingegen setzt auf Konventionen und vordefinierte Standards, was die Entwicklung beschleunigt, aber gleichzeitig weniger Kontrolle bietet. (Lagnada), 2024)

2.1.4 Einsatz

Spring Boot	**Spring**
Ideal für Microservices, die eigenständig und skalierbar sein müssen	**Geeignet für monolithische Anwendungen mit individuellen Konfigurationen**
Schnellere Entwicklung durch Auto-Configuration	**Mehr Kontrolle über Konfiguration und Architektur**
Eingebettete Server ermöglichen Standalone-Betrieb	**Erfordert externen Application Server**
Vorkonfigurierte Defaults für häufige Anwendungsfälle	**Vollständige Anpassungsfähigkeit an spezielle Anforderungen**

Beste Wahl für Cloud-native und containerisierte Anwendungen	**Gut für Unternehmensanwendungen mit komplexen Integrationen**

Tabelle 2: Vergleich Spring Boot vs. Spring

2.2 Quarkus

2.2.1 Historie und Entwicklung

Quarkus entstand aus dem Bestreben, Java für moderne Cloud- und Container-Umgebungen zu optimieren. Das Framework wurde maßgeblich von Red Hat-Ingenieuren entwickelt und im März 2019 der Öffentlichkeit vorgestellt. Ziel war es, eine Antwort auf die Herausforderungen zu liefern, mit denen klassische Java-Anwendungen in hochskalierbaren und dynamischen Umgebungen häufig konfrontiert waren – insbesondere in Bezug auf Startzeit und Ressourcenverbrauch. (Quarkus, n.d.)

- **Ursprünge und Motivation**:
  Schon in der frühen Planungsphase identifizierten die Quarkus-Entwickler die Notwendigkeit, den klassischen „Java-Stack“ auf seine Essenz zu reduzieren und alle Teile zu entfernen oder zu ersetzen, die in einer Cloud-native-Welt für Performance-Bottlenecks sorgen. Traditionelle Java EE- oder Spring-Anwendungen wiesen oft langsame Startzeiten und relativ hohen Speicherverbrauch auf. In Umgebungen, in denen Microservices skalieren und schnell gestartet werden müssen, war das ein wesentlicher Nachteil.
- **Bedeutende Meilensteine**:
  - **Erste Veröffentlichung (März 2019)**: Vorstellung des Frameworks und Fokus auf schnelle Startzeiten sowie geringe Speicheranforderungen.
  - **Version 1.0 (November 2019)**: Stabilisierung des Kerns, umfangreichere Dokumentation und Erweiterung der Community.
  - **Version 2.0 (Juni 2021)**: Neue Features wie verbesserte Dev-Mode-Unterstützung, optimierte reaktive Programmiermodelle (u.a. mit Mutiny) und verbesserte Zusammenarbeit mit GraalVM.

- **Community und Open-Source**:
  Quarkus ist als Open-Source-Projekt auf GitHub verfügbar und profitiert von einer wachsenden Community aus Entwicklern, Unternehmen und Enthusiasten. Das aktive Feedback und die rege Beteiligung führen zu einer kontinuierlichen Weiterentwicklung und Anpassung an aktuelle Anforderungen – sei es im Bereich Serverless, Microservices oder Container-Orchestrierung. (Quarkus, n.d.)

2.2.2 Architektur und Kernkomponenten

Die Architektur von Quarkus basiert auf dem sogenannten „Container-First"-Ansatz und versucht, möglichst viele Verarbeitungs- und Initialisierungsprozesse bereits in der Build-Zeit durchzuführen. Dadurch werden Laufzeit-Overheads minimiert, was zu schnelleren Startzeiten und geringerem Speicherverbrauch führt. (Quarkus, 2025)

- **Build-Time-Verarbeitung**
  - **Ahead-of-Time (AOT) Kompilierung**: Ein wesentlicher Faktor für die Performance ist, dass Quarkus so viel wie möglich bereits beim Kompilieren der Anwendung erledigt. Klassen werden analysiert, Proxies werden generiert, Reflektionsinformationen werden gesammelt – alles vor dem eigentlichen Anwendungsstart.
  - **Extension- und Plug-in-Mechanismus**: In Quarkus gibt es zahlreiche „Extensions", die bestimmte Funktionalitäten oder Bibliotheken integrieren (z.B. Hibernate, RESTEasy, Kafka, Camel, Vert.x). Jede Extension nutzt die Build-Time-Verarbeitung, um Initialisierungsschritte vorzuziehen und die Laufzeit zu entlasten. (Štefanko & Martiška, 2025)
- **Native Kompilierung mit GraalVM**
  - **Native Images:** Mithilfe von GraalVM lassen sich Quarkus-Anwendungen in sogenannte Native Images kompilieren, also eigenständige Maschinencode-Binaries, die ohne eine klassische JVM starten. Dieses Vorgehen reduziert sowohl die Startzeit auf wenige Millisekunden als auch den Speicherbedarf.
  - **Reflektion und Substitution:** Da GraalVM besondere Anforderungen an Reflektionsaufrufe, dynamische Klassenladungen und Proxy-Erzeugung stellt,

kümmert sich Quarkus während der Build-Phase darum, alle benötigten Metadaten zu erzeugen. Das verhindert Laufzeitfehler und macht Native Images überhaupt erst praktikabel.

- **Reactive Programming**
  - **Reaktive Kernbibliotheken:** Quarkus unterstützt reaktives Programmieren über Bibliotheken wie Mutiny. Dies erlaubt die Entwicklung nicht-blockierender und skalierbarer Anwendungen, die besonders gut in Cloud- und Microservice-Umgebungen funktionieren.
  - **Event-Driven-Ansatz:** Durch die Reaktivität können Anwendungen eingehende Ereignisse (z.B. HTTP-Requests, Messaging-Events) effizient verarbeiten, ohne blockierende Threads oder lange Wartezeiten.
- **Dependency Injection (DI) und CDI**
  - **Quarkus und CDI (Contexts and Dependency Injection):** Das DI-Framework in Quarkus basiert auf dem Jakarta EE-Standard (CDI). Mit „Arc“ als Implementierung wird sichergestellt, dass das Dependency Injection-Konzept leichtgewichtig und modular bleibt.
  - **Lebenszyklus- und Scope-Management:** CDI ermöglicht eine flexible Verwaltung von Beans, deren Lebenszyklus und Kontext (z.B. Request- oder Session-Scopes), was besonders für lose gekoppelte Microservices von Vorteil ist.

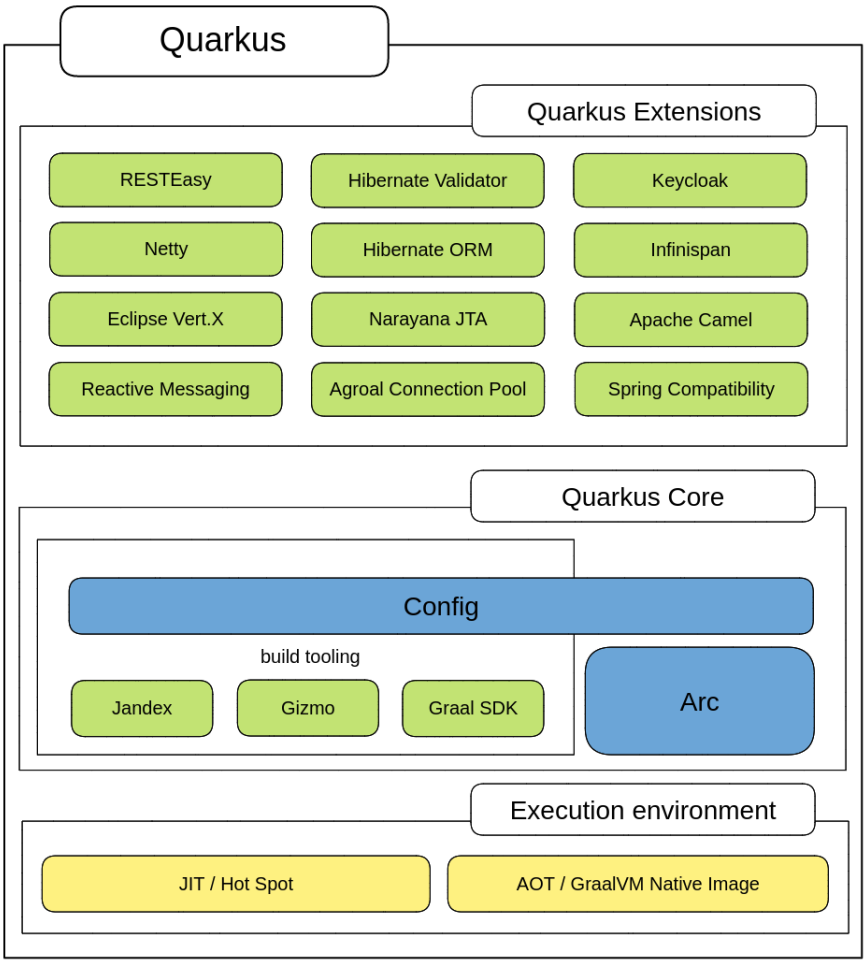


Abbildung 5: Quarkus Architektur und Kernkomponenten
(Štefanko & Martiška, 2025)

2.2.3 Hauptmerkmale von Quarkus

Quarkus bietet eine Vielzahl von Funktionen, die es von traditionellen Java-Frameworks abheben (Quarkus, n.d.):

- **Schnelle Startzeiten und geringer Speicherverbrauch**: Durch die Vorverlagerung von Verarbeitungsaufgaben in die Build-Phase und die Möglichkeit, native Images mit GraalVM zu erstellen, starten Quarkus-Anwendungen in Millisekunden und benötigen weniger Speicher.

- **Entwicklerfreundlichkeit**: Funktionen wie Live Coding ermöglichen es Entwicklern, Änderungen im Code sofort zu sehen, ohne die Anwendung neu starten zu müssen. Dies beschleunigt den Entwicklungszyklus erheblich.
- **Reaktive Programmierung**: Quarkus unterstützt sowohl imperative als auch reaktive Programmiermodelle, was Entwicklern Flexibilität bei der Gestaltung ihrer Anwendungen bietet.
- **Nahtlose Integration mit Kubernetes**: Dank optimierter Container-Bereitstellung und Kubernetes-Nativität lassen sich Quarkus-Anwendungen effizient in Cloud-Umgebungen betreiben.
- **Umfassende Erweiterbarkeit**: Mit über 50 Erweiterungen können Entwickler die Funktionalität von Quarkus leicht an ihre spezifischen Bedürfnisse anpassen.

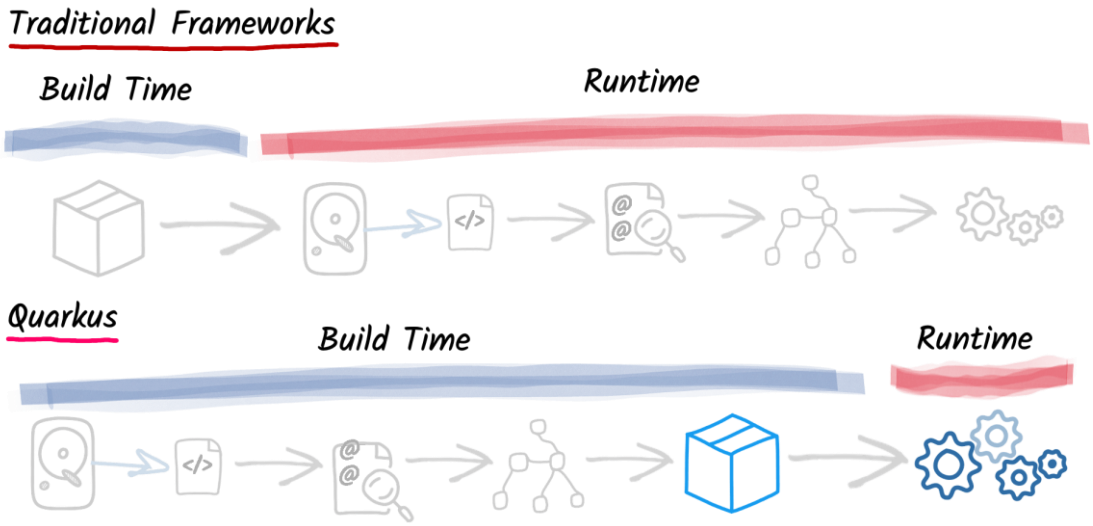


Abbildung 6: Container-First Ansatz von Quarkus (Quarkus, 2025)

2.3 Empirische Vergleichsstudien zu Quarkus und Spring Boot

In der Fachliteratur und in technischen Blogs gibt es zahlreiche Vergleichsstudien zwischen Quarkus und Spring Boot, die sich hauptsächlich mit Performance-Aspekten befassen. Diese Studien liefern wertvolle Einblicke in zentrale Leistungsmerkmale wie Startzeiten,

Speicherverbrauch und Durchsatz, die gerade in modernen Cloud- und Microservice-Umgebungen eine wichtige Rolle spielen.

Ein häufig zitierter Vergleich ist *Spring Boot vs. Quarkus: Performance Comparison for Hello World Case* (C, 2023). Hier wurde eine einfache REST-API mit einem Hello-World-Endpoint in beiden Frameworks implementiert und hinsichtlich ihrer Startzeit und Antwortzeiten untersucht. Die Ergebnisse zeigen, dass Quarkus insbesondere im nativen Modus eine deutlich schnellere Startzeit aufweist und weniger Speicher benötigt. Im Gegensatz dazu bietet Spring Boot im JVM-Modus eine stabilere Laufzeitperformance und kann je nach Szenario Vorteile bei der Durchsatzrate haben.

Ein weiterer relevanter Artikel ist *Quarkus vs. Micronaut vs. Spring Boot: A Comparative Guide for Java Developers* (Ter, 2024). Hier wird Quarkus mit zwei anderen populären Java-Frameworks verglichen, wobei der Fokus auf der Startzeit, der Ressourcennutzung und der Entwicklererfahrung liegt. Auch hier zeigt sich, dass Quarkus im nativen Modus beeindruckend geringe Startzeiten erreicht, während Spring Boot durch seine breite Unterstützung von Bibliotheken und Tooling punkten kann.

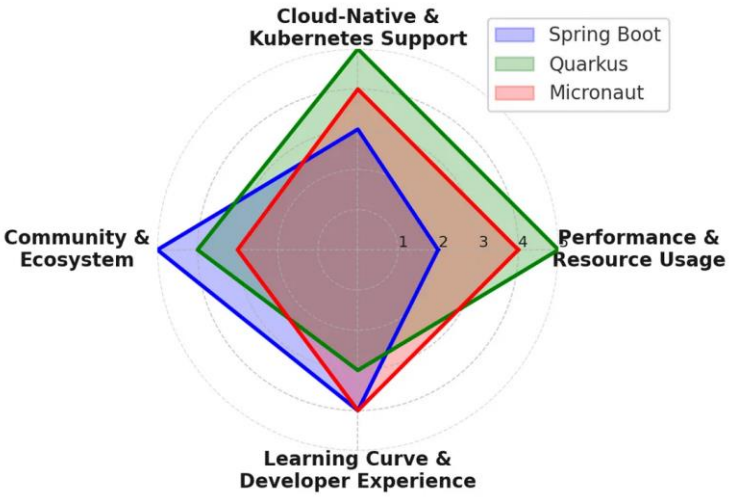


Abbildung 7: Spring Boot vs. Quarkus vs. Micronaut
(Ter, 2024)

Ein dritter relevanter Vergleich, *Spring Boot and Quarkus: Comparing Performance and Usage* (Zanetti, 2024), bestätigt viele der zuvor genannten Erkenntnisse. Hier wurde ebenfalls

die Startgeschwindigkeit und Speicherverbrauch untersucht, wobei Quarkus insbesondere bei geringem Speicherbedarf überzeugt. Spring Boot bleibt weiterhin eine bewährte Wahl für viele Entwickler aufgrund der großen Community und der ausgereiften Infrastruktur.

Die bestehenden empirischen Vergleiche liefern interessante Einblicke in die Stärken und Schwächen beider Frameworks. Während Quarkus durch schnelle Startzeiten und geringe Ressourcennutzung überzeugt, bietet Spring Boot eine ausgereifte Umgebung mit vielen Erweiterungsmöglichkeiten. Viele dieser Tests basieren auf einfachen Beispielanwendungen, was eine erste Orientierung bietet, aber nicht alle Einsatzszenarien vollständig abbildet. Dennoch sind diese Vergleiche hilfreich, um zu verstehen, in welchen Bereichen Quarkus oder Spring Boot ihre jeweiligen Vorteile haben und wie sich diese in einer CRUD-Anwendung auswirken können.

3 Methodik

In diesem Kapitel wird das methodische Vorgehen zur Durchführung des Framework-Vergleichs zwischen Quarkus und Spring Boot beschrieben. Im Mittelpunkt steht die Entwicklung und der gezielte Einsatz eines Microservice-basierten Anwendungssystems, anhand dessen beide Frameworks unter kontrollierten Bedingungen getestet werden. Es werden die Zielsetzung des Vergleichs, die verwendeten Metriken und Bewertungskriterien sowie der Aufbau der Testumgebung detailliert erläutert. Abschließend werden die einzelnen Testverfahren beschrieben, mit denen die Leistungsfähigkeit, Ressourcennutzung und Skalierbarkeit der Frameworks praxisnah untersucht werden.

3.1 Zielsetzung des Vergleichs

Ziel dieser Arbeit ist es, die beiden weit verbreiteten Frameworks Quarkus und Spring Boot im Kontext einer Microservice-basierten, cloudnativen Anwendung miteinander zu vergleichen. Dazu wird ein praxisnahes Anwendungsszenario – ein Promotion-Management-System – implementiert, welches als Grundlage für den technischen Vergleich dient.

Im Fokus des Vergleichs stehen vor allem die Performance, der Ressourcenverbrauch sowie die Skalierbarkeit der beiden Frameworks. Dabei sollen nicht nur theoretische Aspekte beleuchtet, sondern auch empirische Messwerte anhand realer Tests gesammelt und ausgewertet werden.

Die zentrale Forschungsfrage lautet:

Inwiefern unterscheiden sich Quarkus und Spring Boot hinsichtlich Performance, Ressourceneffizienz und Skalierbarkeit in einer containerisierten Microservice-Anwendung?

Um diese Fragestellung zu beantworten, werden identische Microservices mit Quarkus und Spring Boot implementiert. Die Services erfüllen dieselbe Funktionalität (z. B. Benutzerverwaltung, Angebotsverwaltung, Store-Verwaltung) und werden unter vergleichbaren Bedingungen getestet. Die so gewonnenen Ergebnisse sollen dabei helfen, die jeweiligen Stärken und

Schwächen der Frameworks praxisnah zu bewerten und Empfehlungen für deren Einsatz in cloudnativen Architekturen abzuleiten.

3.2 Vergleichskriterien und Metriken

Für eine fundierte Bewertung der beiden Frameworks Quarkus und Spring Boot werden verschiedene technische Metriken herangezogen, die sowohl die Entwicklungs- als auch die Laufzeiteigenschaften der Systeme abbilden. Der Vergleich erfolgt auf Basis objektiv messbarer Kriterien in mehreren Kategorien:

3.2.1 Buildzeit

Die Buildzeit beschreibt die Dauer, die benötigt wird, um aus dem Quellcode ein lauffähiges Artefakt zu erzeugen (z. B. JAR-Datei oder natives Binary bei Quarkus). Dabei werden sowohl die JVM-Buildzeit als auch – im Fall von Quarkus – die Native-Image-Buildzeit betrachtet.

Ziel: Bewertung der Effizienz im Entwicklungsprozess und der CI/CD-Tauglichkeit.

3.2.2 Startzeit

Die Startzeit misst die Dauer vom Start des Containers (z. B. via docker run) bis zur vollständigen Einsatzbereitschaft des jeweiligen Services. Diese Metrik ist besonders relevant für skalierende und kurzlebige Dienste in Cloud-Umgebungen (z. B. beim Serverless Computing oder Auto-Scaling).

Ziel: Analyse des Potenzials für schnelle Bereitstellung und elastisches Skalieren.

3.2.3 Imagegröße

Hier wird die Größe des finalen Container-Images gemessen. Kleinere Images sind vorteilhaft beim Deployment, insbesondere in CI/CD-Pipelines und bei der Übertragung über Netzwerke.

Ziel: Bewertung der Portabilität und Deployment-Effizienz.

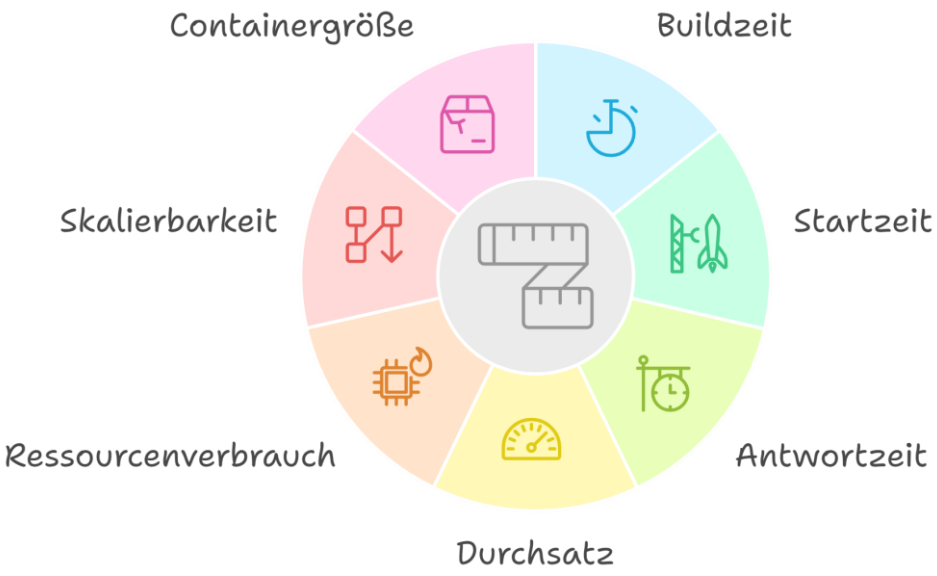


Abbildung 8: Vergleichskriterien und Metriken

3.2.4 Antwortzeit (Latenz)

Die Antwortzeit gibt an, wie lange ein Service benötigt, um eine HTTP-Anfrage zu verarbeiten und eine Antwort zurückzugeben. Diese wird unter kontrollierter Last gemessen und spiegelt die Reaktionsfähigkeit der Anwendung wider.

Ziel: Beurteilung der Interaktivität und Benutzerfreundlichkeit unter realistischen Bedingungen.

3.2.5 Durchsatz (Requests pro Sekunde)

Der Durchsatz zeigt, wie viele Anfragen ein Service pro Sekunde verarbeiten kann. Je höher der Wert, desto leistungsfähiger ist die Anwendung unter hoher Last.

Ziel: Bewertung der Effizienz bei gleichzeitigen Zugriffen und im Dauerbetrieb.

3.2.6 Ressourcenverbrauch

Hier werden der CPU-Verbrauch und der RAM-Verbrauch sowohl im Leerlauf als auch unter Last betrachtet. Diese Werte werden mithilfe von Monitoring-Tools wie Docker Stats oder Prometheus erhoben.

Ziel: Vergleich der Ressourceneffizienz beider Frameworks bei gleichem Anwendungsumfang.

3.2.7 Skalierbarkeit

Die Skalierbarkeit beschreibt, wie gut die Frameworks auf horizontale Skalierung reagieren, d. h. wie sie sich verhalten, wenn mehrere Instanzen gestartet und Last verteilt wird. Dabei wird insbesondere auf die Anwortzeit-, Durchsatz- und Ressourcennutzungsentwicklung bei steigender Instanzzahl geachtet.

Ziel: Einschätzung der Eignung für dynamische Cloud-Umgebungen.

3.3 Versuchsaufbau (Testumgebung)

Zur Durchführung der vergleichenden Analyse von Quarkus und Spring Boot wurde eine standardisierte Testumgebung auf einem dedizierten Entwicklungssystem eingerichtet. Ziel war es, eine kontrollierte und reproduzierbare Umgebung zu schaffen, in der beide Frameworks unter identischen Bedingungen evaluiert werden können.

Die Experimente wurden auf einem lokalen Rechner mit folgenden Hardware- und Softwarekonfigurationen durchgeführt:

- **Betriebssystem:** Microsoft Windows 10 Pro (Version 10.0.19045, Build 19045)
- **Prozessor:** Intel® Core™ i5-6600K CPU @ 3.50 GHz
- **Arbeitsspeicher:** 16 GB RAM
- **Systemtyp:** x64-basierter PC

- **Java-Version:** JDK 23, ergänzt um GraalVM 23.0 für native Builds mit Quarkus
- **Build-Werkzeug:** Apache Maven (Projektverwaltung und Abhängigkeitsmanagement)

Zur Containerisierung der Microservices wurde **Docker Desktop (Version 4.38.0)** eingesetzt. Jeder Service – also der Offer-, Store- und User-Service – wurde als eigenständiger Docker-Container mit eigenem Image ausgeführt. Die zugehörigen SQLite-Datenbanken wurden direkt über Maven-Abhängigkeiten eingebunden und pro Service lokal instanziiert.

Für die Orchestrierung der Container sowie das Service-Routing kam **Docker Compose** in Kombination mit **Traefik** als Reverse Proxy zum Einsatz. Um die horizontale Skalierbarkeit in realistischen Bedingungen zu simulieren, wurde zusätzlich **Docker Swarm** zur Steuerung mehrerer Service-Instanzen verwendet.

Die Überwachung und Analyse des Ressourcenverbrauchs (RAM, CPU) erfolgte über die Kombination von **Prometheus**, **Grafana** und **cAdvisor**, wobei alle drei Tools als Docker-Container (jeweils mit dem latest-Tag) betrieben wurden.

Zur Durchführung der Lasttests wurde das Tool **Apache JMeter (Version 5.6.3)** auf dem Host-System ausgeführt, um gezielt HTTP-Anfragen an die jeweiligen Microservices zu senden und deren Antwortzeiten sowie Durchsatz zu messen.

Die gesamte Testinfrastruktur wurde so konzipiert, dass sie vollständig isoliert und unabhängig vom Entwicklungssystem operieren kann, wodurch eine hohe Aussagekraft und Reproduzierbarkeit der Ergebnisse gewährleistet ist.

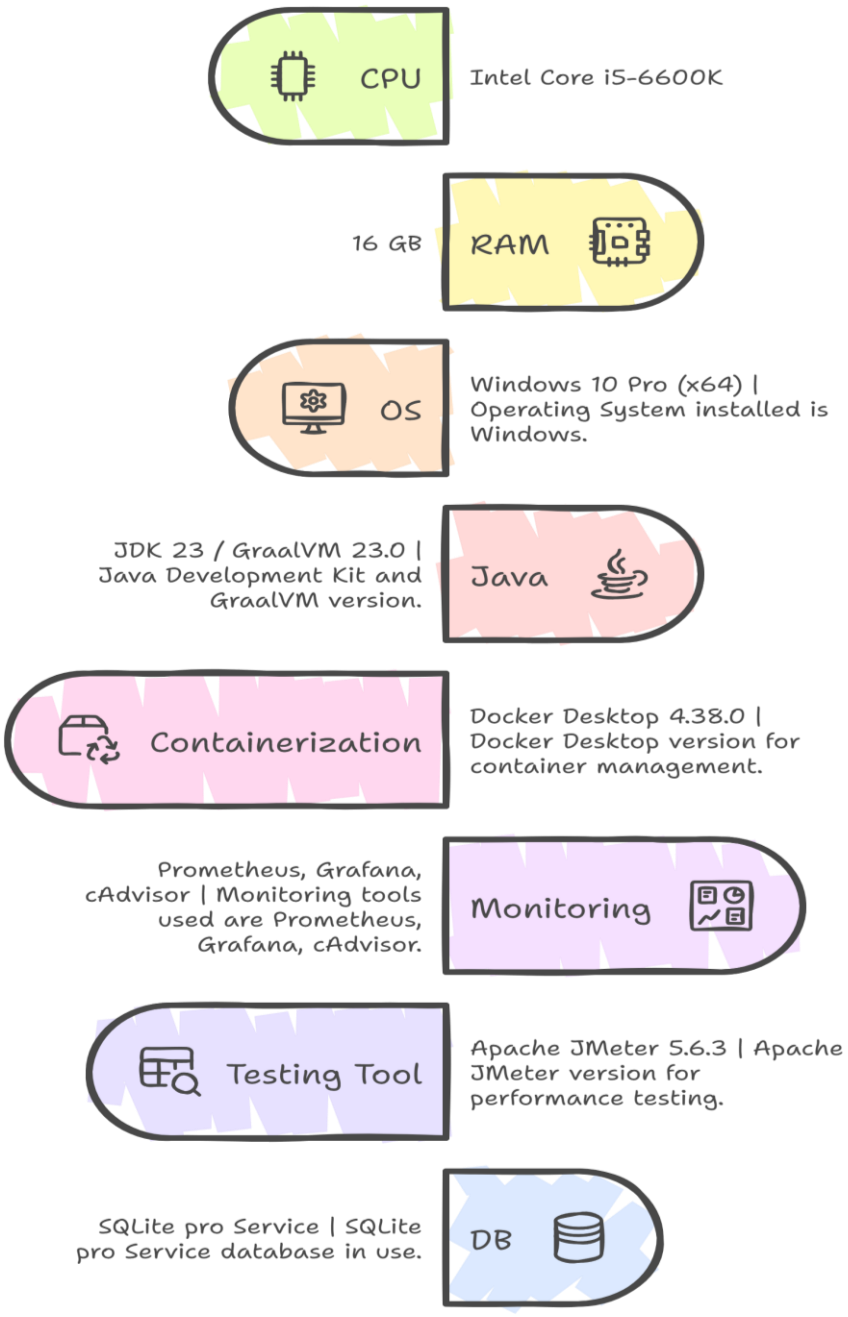


Abbildung 9: Software und Hardware der Testumgebung

3.4 Testdurchführung (Testplan)

Die Durchführung der Lasttests erfolgte anhand dreier unterschiedlicher Testpläne, die mit Apache JMeter realisiert wurden. Die Testpläne variieren hinsichtlich der Anzahl paralleler Nutzer (Threads), der verwendeten Request-Typen (GET- und POST-Anfragen), sowie der Intensität und Dauer der Belastung.

Im ersten Testplan wurden insgesamt 105 parallele Threads eingesetzt, bestehend aus 100 GET-Anfragen auf die Ressource /offers (6000 Requests pro Minute) und 5 parallelen POST-Anfragen auf /stores (300 Requests pro Minute). Die Dauer dieses Szenarios betrug 300 Sekunden bei einer Ramp-up-Zeit von 10 Sekunden.

Der zweite Testplan umfasste insgesamt 806 parallele Threads, verteilt auf vier Thread-Gruppen mit unterschiedlichen Anfragetypen und Durchsatzraten (GET /offers und /stores je 24.000 Requests pro Minute, POST /stores 60 Requests pro Minute, POST /offers 300 Requests pro Minute). Der Test wurde über einen Zeitraum von 300 Sekunden durchgeführt, mit einer Ramp-up-Zeit von 10 Sekunden. Zudem wurde dieser Test nach einer horizontalen Skalierung auf je drei Instanzen der Microservices ausgeführt.

Im dritten Testplan wurden 2070 parallele Threads verwendet, ebenfalls aufgeteilt in vier Thread-Gruppen mit deutlich höherer Last (GET-Anfragen je 60.000 Requests pro Minute, POST /stores 1.200 Requests pro Minute, POST /offers 3.000 Requests pro Minute). Die Testdauer betrug erneut 300 Sekunden mit einer Ramp-up-Zeit von 10 Sekunden. Wie beim zweiten Testplan erfolgte die Durchführung nach horizontaler Skalierung auf jeweils drei Replikate der Offer- und Store-Services, um das Verhalten der Frameworks bei intensiver Belastung und mehreren Service-Instanzen realistisch abzubilden.

Die detaillierte Übersicht zu den verwendeten Testplänen und deren spezifischen Konfigurationen ist in *Tabelle 3* dargestellt.

Test-plan	Beschreibung der Testszenarien	Gleichzeitige Nutzer (Threads)	Dauer	Ramp-up	Skalierung
1	**Gruppe 1:** 100 Threads, GET-Anfragen auf /offers, Durchsatz: 6.000 Requests pro Minute **Gruppe 2:** 5 Threads, POST-Anfragen auf /stores, Durchsatz: 300 Requests pro Minute	**105 Threads** insgesamt (100 GET + 5 POST)	300 s	10 s	Nein
2	**Gruppe 1:** 400 Threads, GET /offers, 24.000 req/min **Gruppe 2:** 400 Threads, GET /stores, 24.000 req/min **Gruppe 3:** 1 Thread, POST /stores, 60 req/min **Gruppe 4:** 5 Threads, POST /offers, 300 req/min	**806 Threads** insgesamt (400 GET + 400 GET + 1 POST + 5 POST)	300 s	10 s	Ja (je 3 Instanzen von Offer & Store)
3	**Gruppe 1:** 1000 Threads, GET /offers, 60.000 req/min **Gruppe 2:** 1000 Threads, GET /stores, 60.000 req/min **Gruppe 3:** 20 Threads, POST /stores, 1.200 req/min **Gruppe 4:** 50 Threads, POST /offers, 3.000 req/min	**2070 Threads** insgesamt (1000 GET + 1000 GET + 20 POST + 50 POST)	300 s	10 s	Ja (je 3 Instanzen von Offer & Store)

Tabelle 3: Übersicht der Testpläne für die Lasttests

4 Implementierung

In diesem Kapitel wird die konkrete Umsetzung der Beispielanwendung für das Promotion-Management-System vorgestellt, welche als Grundlage für den Vergleich zwischen Quarkus und Spring Boot dient. Die Anwendung besteht aus mehreren Microservices, die typische Funktionen wie Angebotsverwaltung, Store-Management und Benutzerverwaltung abbilden. Ziel war es, eine identische fachliche Funktionalität mit beiden Frameworks zu realisieren, um eine objektive Vergleichbarkeit zu ermöglichen. Es wird zunächst auf den fachlichen und technischen Kontext eingegangen, anschließend die Architektur der Anwendung beschrieben und abschließend exemplarisch ein Prozessablauf zur Nutzung des Systems erläutert.

4.1 Beschreibung des Promotion-Management-Systems

4.1.1 Kurze Einführung in die Domäne:

- Das entwickelte Promotion-Management-System dient zur Verwaltung und Anzeige von Angeboten in verschiedenen Läden.
- Benutzer können aktuelle Aktionen einsehen, während Ladenbesitzer neue Angebote erstellen und verwalten.

- Administratoren wiederum sind in der Lage, neue Stores hinzuzufügen oder bestehende zu pflegen.

Abbildung 10: Promotion-Management-System Workflow

4.1.2 Begründung für die Auswahl als Vergleichssystem:

Die Anwendung eignet sich ideal für die Forschungsfrage, da sie typische Merkmale einer Microservice-Architektur aufweist:

- mehrere eigenständige Services (User, Store, Offer),
- REST-Schnittstellen,
- Datenpersistenz mit eigenen Datenbanken,
- realistische Geschäftsprozesse

4.2 Kontextsicht des Systems

4.2.1 Fachliche Kontextsicht:

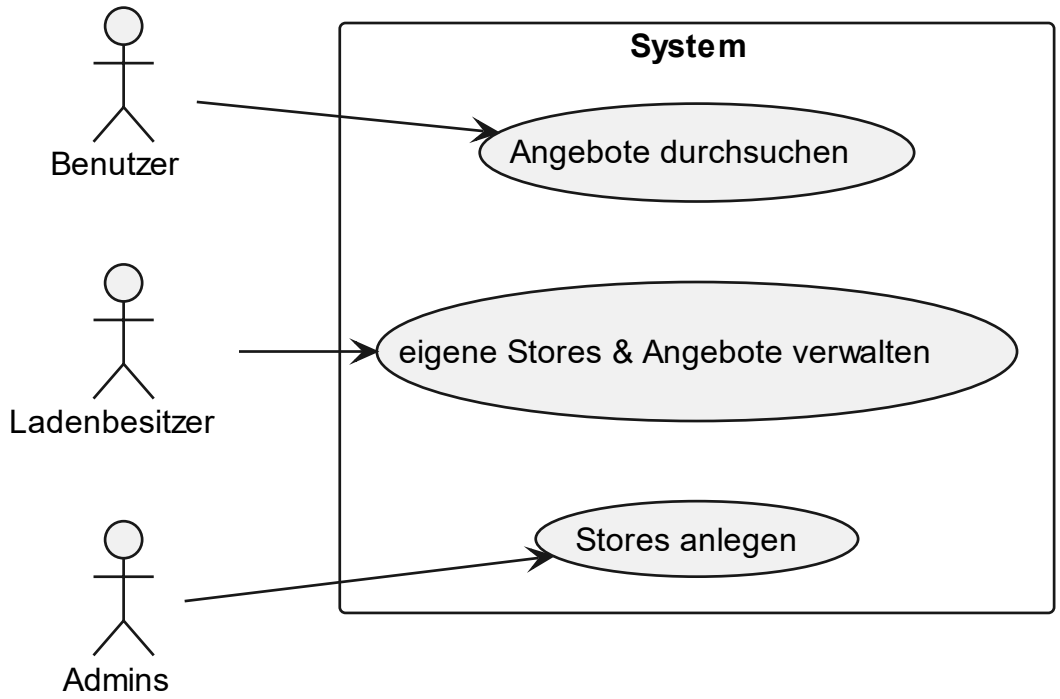


Abbildung 11: Fachliche Kontextsicht

4.2.2 Technische Kontextsicht:

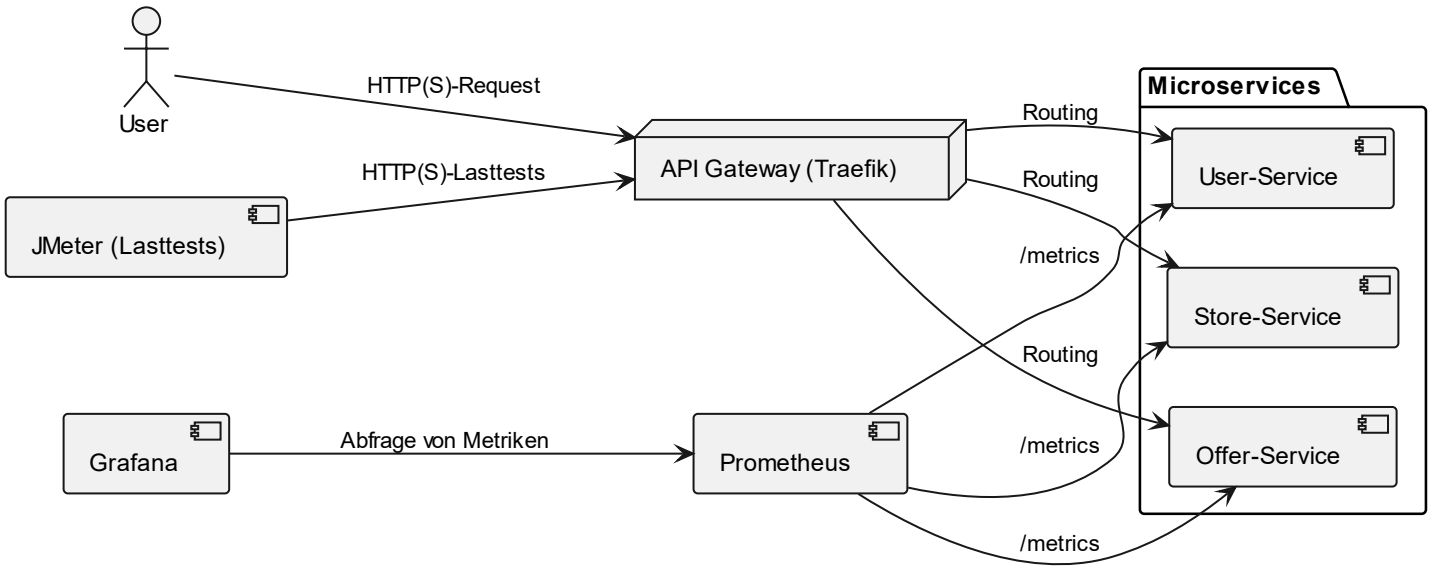


Abbildung 12: Technische Kontextsicht

4.3 Bausteinsicht

Die *Abbildung 13* zeigt die Bausteinsicht des entwickelten Promotion-Management-Systems. Dabei handelt es sich um eine klassische mehrschichtige Architektur, die in allen drei Microservices – Store, Offer und User – einheitlich umgesetzt wurde, sowohl in der Quarkus- als auch in der Spring Boot-Variante.

Die Architektur gliedert sich in vier vertikale Säulen (je Service) und vier horizontale Schichten:

- Die **Controller-Schicht** bildet die REST-Schnittstelle nach außen und verarbeitet HTTP-Anfragen.
- Die **Service-Schicht** enthält die Geschäftslogik und dient als Vermittler zwischen Controller und Datenzugriff.
- Die **Repository-Schicht** kapselt den Datenzugriff auf die persistente SQLite-Datenbank unter Verwendung von Panache (Quarkus) bzw. Spring Data JPA.

- Die unterste Schicht besteht aus separaten **SQLite-Datenbanken**, die jedem Microservice exklusiv zugeordnet sind.

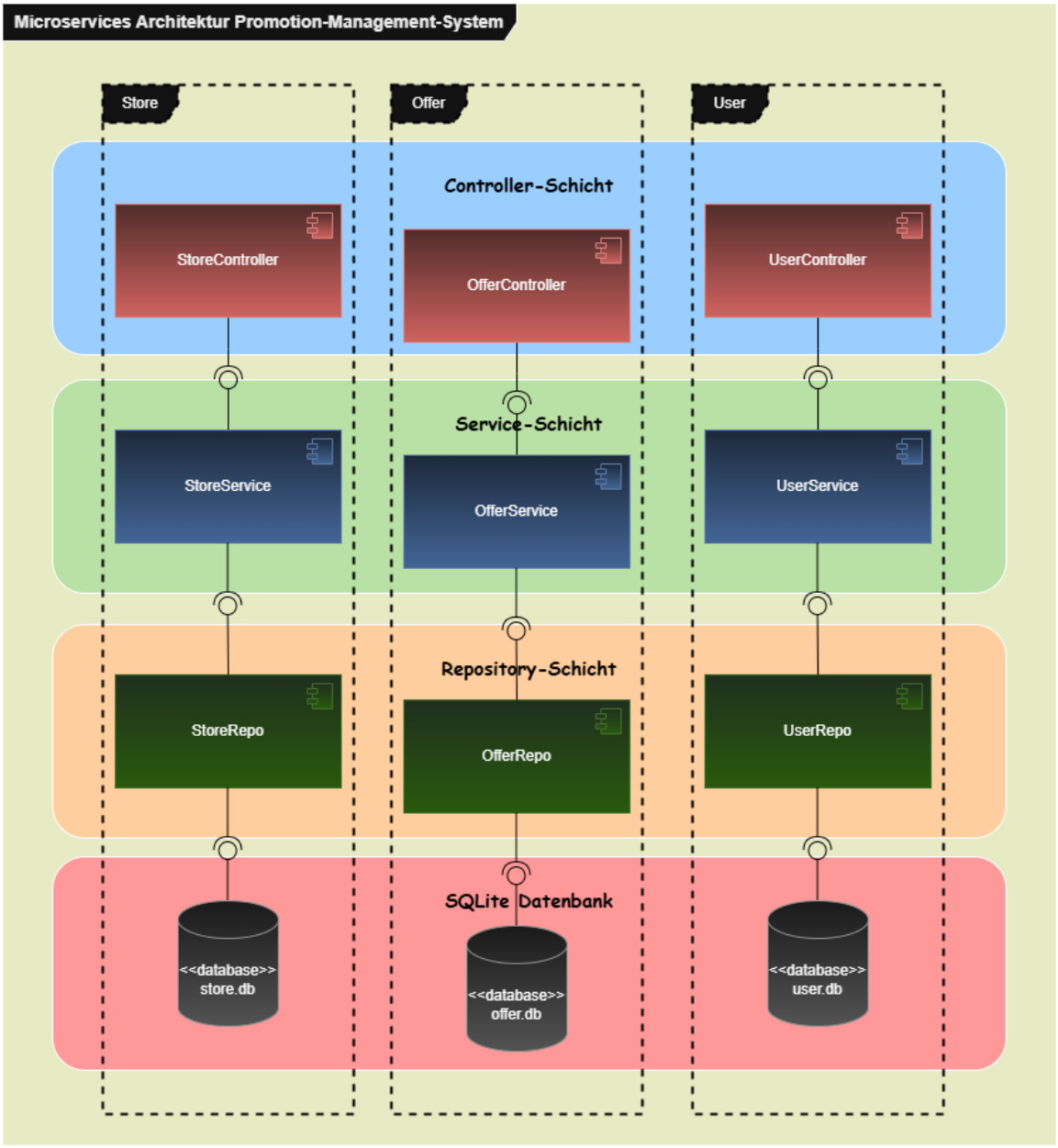


Abbildung 13: Bausteinsicht des Promotion-Management-Systems

4.4 Laufzeitsicht

Die *Abbildung 14* stellt eine Laufzeitsicht des Promotion-Management-Systems dar und veranschaulicht den Ablauf des Prozesses „Angebot hinzufügen“ aus Sicht eines Ladenbesitzers.

Der Ablauf beginnt mit einer POST-Anfrage an den API-Gateway, die ein neues Angebot mit Store-ID, Produkt und Preis enthält. Der Offer-Service leitet die Anfrage weiter, ruft jedoch zunächst den Store-Service auf, um die Gültigkeit der angegebenen Store-ID zu prüfen. Dazu erfolgt eine GET-Anfrage an den Store-Service, der wiederum per SQL-Query die zugehörige Store-DB befragt.

Wenn die Store-ID gültig ist, antwortet der Store-Service mit HTTP 200, woraufhin der Offer-Service das Angebot in der eigenen Datenbank speichert (INSERT). Bei Erfolg wird HTTP 201 Created zurückgegeben, andernfalls – wenn der Store nicht gefunden wurde – ein HTTP 400 Bad Request.

Begleitend wird der Ablauf von Prometheus überwacht, insbesondere im Hinblick auf CPU- und RAM-Verbrauch.

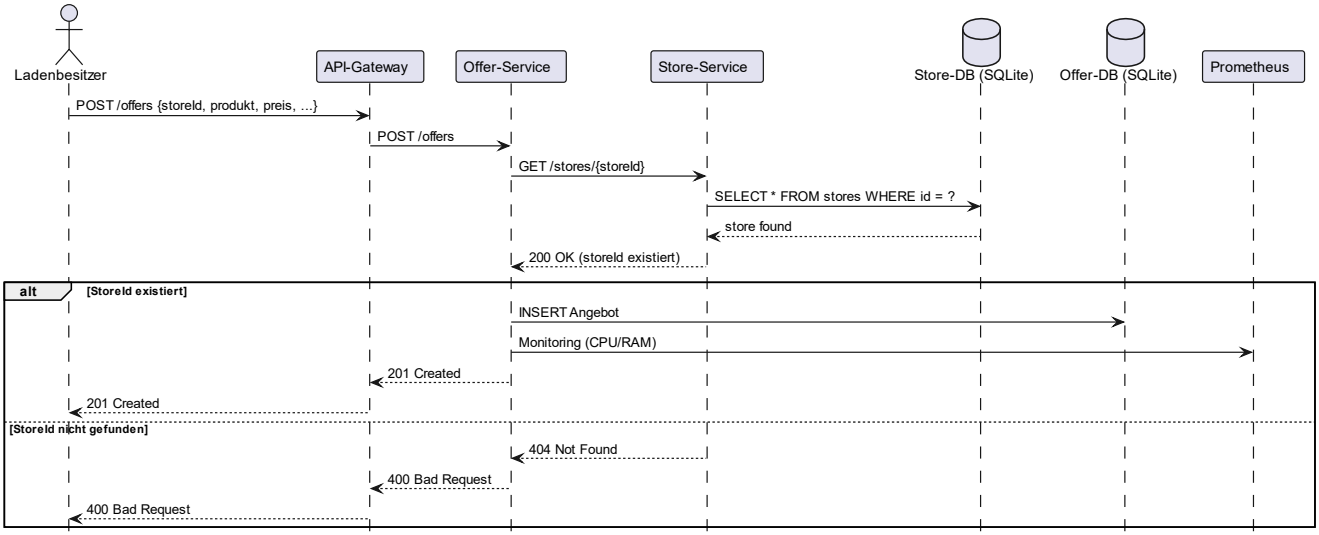


Abbildung 14: Laufzeitsicht Beispielprozess (Angebot hinzufügen)

5 Analyse, Vergleich und Bewertung der Ergebnisse

In diesem Kapitel werden die im Rahmen der Tests gewonnenen Ergebnisse systematisch dargestellt, verglichen und bewertet. Der Fokus liegt dabei auf zentralen Leistungsaspekten wie Startzeit, Antwortzeit, Durchsatz, Ressourcenverbrauch und Skalierbarkeit der beiden Frameworks Quarkus und Spring Boot. Anhand praxisnaher Testpläne wurden die Services in verschiedenen Lastszenarien analysiert – sowohl im Einzelbetrieb als auch in skalierter Ausführung. Die Messergebnisse werden in Form von Tabellen und Diagrammen aufbereitet und kritisch hinsichtlich ihrer Aussagekraft und Relevanz für reale Anwendungsszenarien reflektiert. Ziel ist es, fundierte Aussagen über die Effizienz und Eignung beider Frameworks für Cloud-native Microservice-Architekturen zu treffen.

5.1 Vergleich grundlegender Metriken (Store-Service)

In diesem Abschnitt wird der Store-Service exemplarisch herangezogen, um zentrale Basismetriken zwischen den Frameworks Quarkus und Spring Boot zu vergleichen. Der Store-Service stellt in beiden Varianten die identischen Funktionen zur Verfügung und eignet sich daher ideal für eine objektive Gegenüberstellung. Im Fokus stehen die Buildzeit, die Startzeit sowie die Größe des resultierenden Docker-Images im JVM- und Native-Modus. Die folgenden Diagramme veranschaulichen die jeweiligen Unterschiede.

5.1.1 Buildzeit-Vergleich

Die *Abbildung 15* zeigt deutlich, dass der Buildprozess im Native-Modus bei beiden Frameworks deutlich mehr Zeit beansprucht als im JVM-Modus. Spring Boot (Native) benötigt mit 333,6 Sekunden am längsten, während Quarkus (Native) mit 204 Sekunden etwas schneller

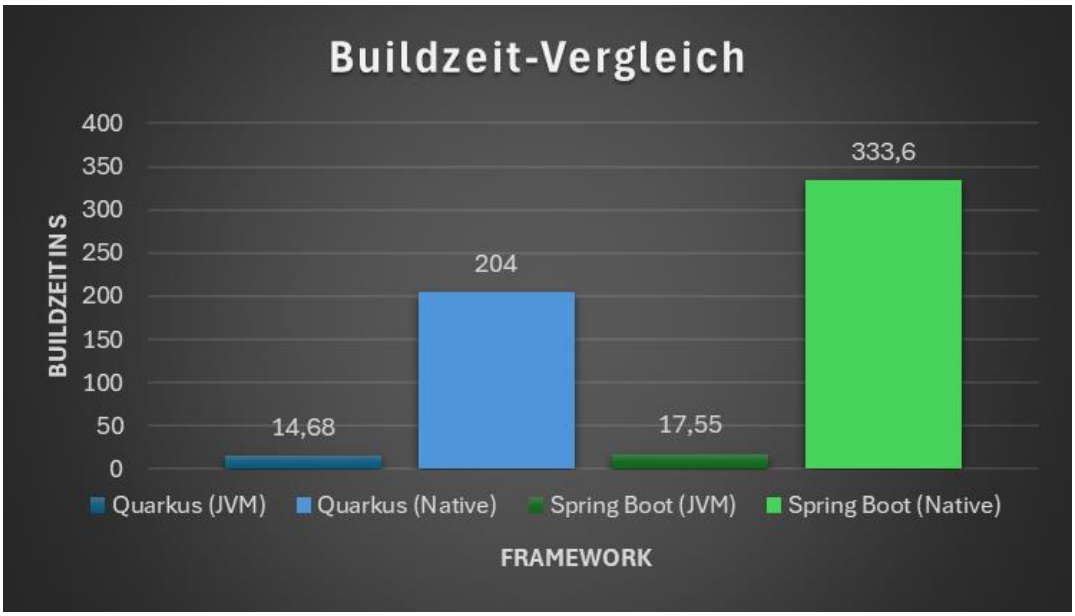


Abbildung 15: Buildzeit-Vergleich (Store-Service)

abschneidet. Im JVM-Modus liegen beide Frameworks eng beieinander (Quarkus: 14,68 s, Spring Boot: 17,55 s).

Die Native-Builds sind wesentlich zeitintensiver – Quarkus jedoch tendenziell effizienter als Spring Boot.

5.1.2 Startzeit-Vergleich

Beim Vergleich der Startzeiten in *Abbildung 16* zeigt sich der größte Unterschied: Quarkus (Native) startet mit nur 0,26 Sekunden extrem schnell, Spring Boot (Native) braucht mit 1,88 Sekunden vergleichsweise mehr Zeit. Im JVM-Modus fällt Spring Boot (22,37 s) deutlich zurück gegenüber Quarkus (5,85 s).

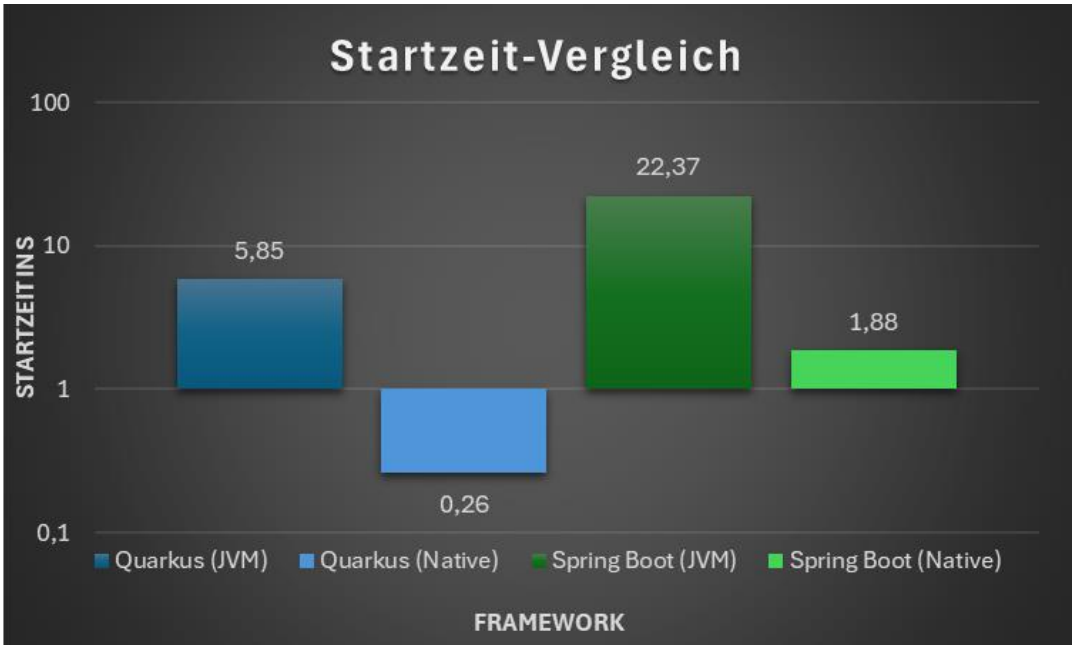


Abbildung 16: Startzeit-Vergleich (Store-Service)

Quarkus überzeugt vor allem durch extrem schnelle Startzeiten im Native-Modus, was besonders für Cloud-Umgebungen (z. B. Serverless) relevant ist.

5.1.3 Imagegröße-Vergleich

Beim Vergleich der Docker-Image-Größen in *Abbdilung 17* zeigt sich, dass beide Frameworks im Native-Modus erheblich kleinere Images erzeugen (Quarkus: 188 MB, Spring Boot: 196,25 MB) im Vergleich zum JVM-Modus (Quarkus: 477,31 MB, Spring Boot: 515,96 MB).

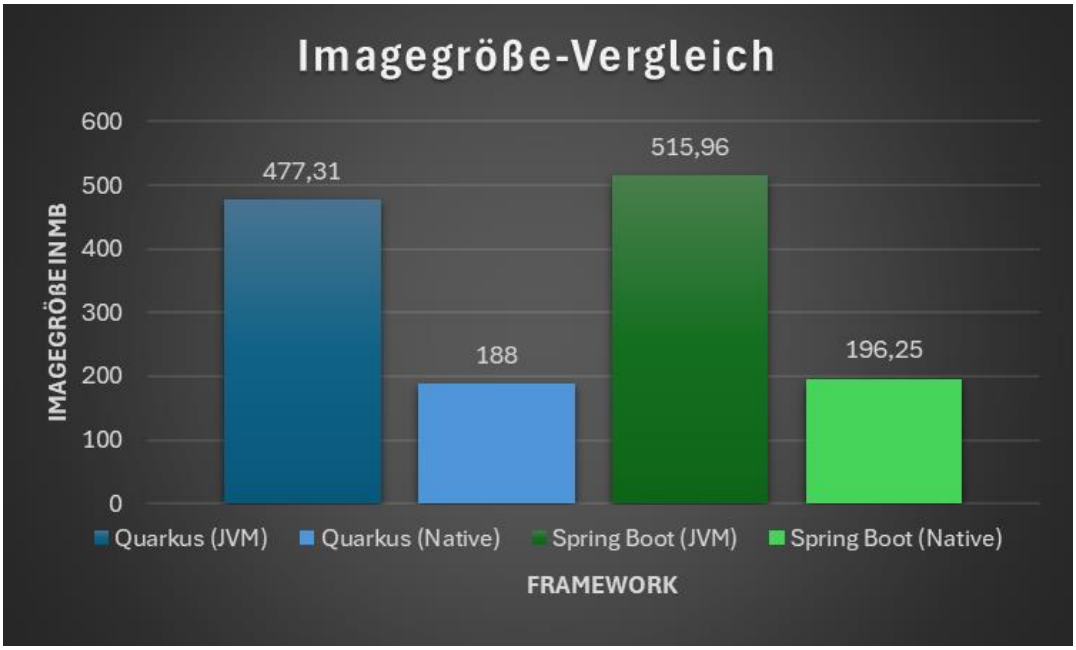


Abbildung 17: Imagegröße-Vergleich (Store-Service)

Native-Images sind erheblich kompakter. Quarkus (Native) erzielt hier den besten Wert und spart gegenüber Spring Boot (JVM) mehr als 60 %.

5.2 Performanzvergleich

In diesem Unterkapitel werden die Ergebnisse der durch Apache JMeter simulierten Lasttests für Quarkus und Spring Boot im Native-Modus systematisch analysiert. Ziel ist es, die Laufzeiteffizienz beider Frameworks unter realitätsnahen Belastungsszenarien zu bewerten. Die Testpläne wurden so konzipiert, dass typische Anwendungsmuster eines Promotion-Management-Systems mit verschiedenen Anfragearten und -frequenzen abgebildet werden. Neben der reinen Antwortzeit (Response Time) werden auch Metriken wie Durchsatz (Throughput) und Fehlerquote betrachtet. Die Ergebnisse bieten eine Grundlage zur Beurteilung, wie beide Frameworks auf steigende Nutzerlast reagieren und welche Leistungsreserven sie unter hoher Auslastung aufweisen.

5.2.1 Anzahl der Samples

Die Anzahl der verarbeiteten Requests, auch als Samples bezeichnet, liefert einen wichtigen Anhaltspunkt für die tatsächliche Last, die während der Tests auf das System ausgeübt wurde. In *Tabelle 4* ist die Gesamtzahl der HTTP-Anfragen aufgeführt, die im Rahmen der drei definierten Testpläne durch die Microservices verarbeitet wurden. Dabei wurden jeweils sowohl der Spring-Boot- als auch der Quarkus-Dienst im Native-Modus ausgeführt. Jeder Testplan hatte eine feste Dauer von fünf Minuten, sodass die Anzahl der Samples im Wesentlichen durch die jeweilige Konfiguration der gleichzeitigen Benutzer, die Anfragetypen sowie die festgelegte Anfragerate bestimmt wurde.

Testplan	Spring Boot	Quarkus
1	31.591	31.605
2	190.986	195.359
3	116.784	127.962

Tabelle 4: Anzahl der verarbeiteten Anfragen

Wie die Tabelle zeigt, sind die Abweichungen zwischen den beiden Frameworks in den Testplänen jeweils gering, was auf eine vergleichbare Testkonfiguration und stabile Antwortverarbeitung in beiden Fällen schließen lässt. Besonders in Testplan 2 und 3, die eine höhere parallele Last aufwiesen, konnte Quarkus insgesamt mehr Anfragen bedienen, was auf eine leicht bessere Durchsatzfähigkeit hindeuten könnte.

5.2.2 Antwortzeit (Response Time)

Die *Abbildung 18* zeigt die durchschnittliche Antwortzeit (Average), den Median sowie die 95%-Line für drei aufeinander aufbauende Lastszenarien (Testpläne 1–3). Die grünen Balken

repräsentieren Spring Boot (Native-Modus), während die blauen Balken die entsprechenden Metriken für Quarkus (Native-Modus) abbilden.

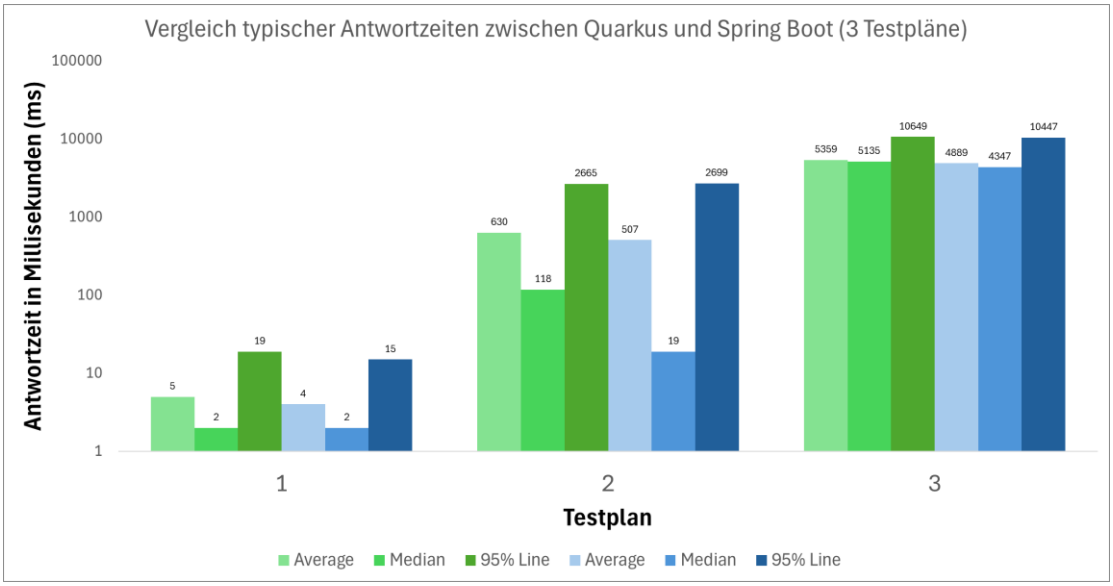


Abbildung 18: Vergleich typischer Antwortzeiten zwischen Quarkus und Spring Boot

Bereits im ersten Testplan mit moderater Last ist ein geringer Vorteil für Quarkus zu erkennen. Dieser Trend verstärkt sich unter zunehmender Belastung (Testpläne 2 und 3): Sowohl Median als auch 95%-Perzentil fallen bei Quarkus deutlich geringer aus, was auf eine stabilere Antwortzeit selbst bei hoher Last schließen lässt. Besonders auffällig ist, dass Quarkus im dritten Testplan – trotz mehr als 127.000 verarbeiteter Anfragen – konsistent niedrigere Werte aufweist.

Die logarithmische Skalierung der Y-Achse verdeutlicht die Unterschiede im höheren Bereich der Antwortzeiten. Insgesamt deuten die Ergebnisse darauf hin, dass Quarkus im Native-Modus insbesondere bei hoher Last reaktionsschneller bleibt und geringere Ausreißer aufweist als Spring Boot.

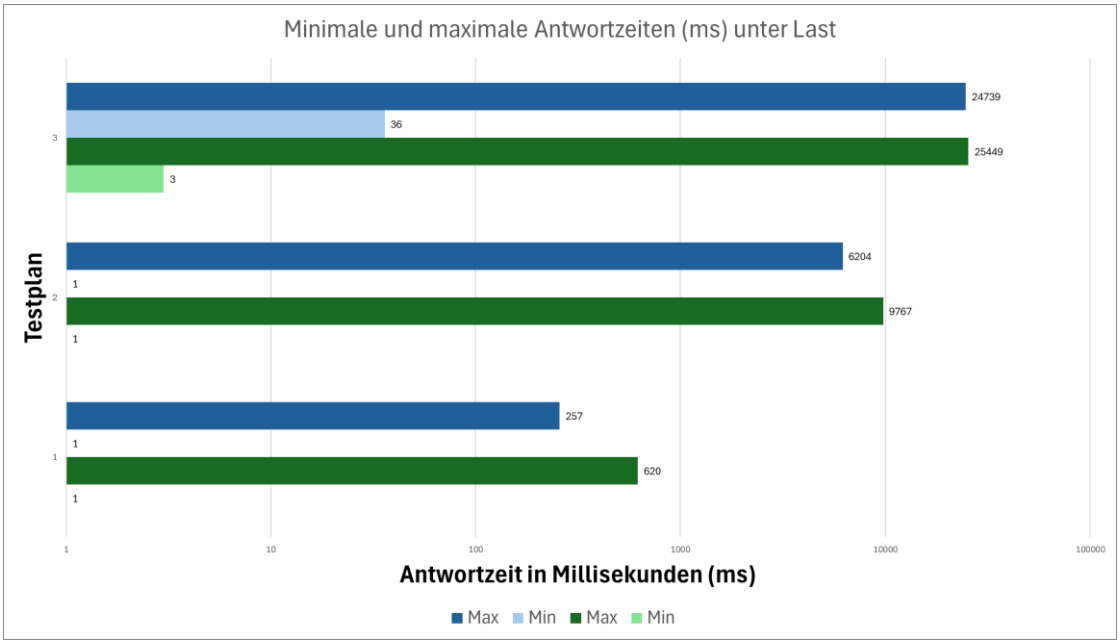


Abbildung 19: Minimale und maximale Antwortzeiten (ms) unter Last

Das Diagramm in der *Abbildung 19* veranschaulicht die minimale und maximale Antwortzeit (in Millisekunden) der getesteten Microservices unter den drei definierten Lastszenarien

Auffällig ist, dass die minimalen Antwortzeiten bei beiden Frameworks konstant sehr niedrig bleiben – im Bereich von 1 bis 3 Millisekunden. Bei der maximalen Antwortzeit hingegen zeigen sich deutliche Unterschiede: Während Quarkus in allen Testplänen leicht geringere Maximalwerte erzielt, steigt insbesondere bei Spring Boot unter starker Last (Testplan 3) die maximale Antwortzeit auf über 25 Sekunden an. Dies weist auf eine geringere Stabilität unter extremer Last hin.

Die Ergebnisse unterstreichen somit die höhere Konstanz von Quarkus bei gleichzeitigem Lastanstieg und belegen ein robusteres Antwortverhalten in Bezug auf Ausreißer.

5.2.3 Durchsatz und Stabilität

Ein zentraler Aspekt bei der Bewertung der Performance eines Frameworks unter Lastbedingungen ist der erzielte Durchsatz, gemessen in Anfragen pro Sekunde (req/s).

In *Abbildung 20* ist der Vergleich der maximal erzielten Durchsatzwerte für Quarkus und Spring Boot zu sehen. Dabei wird deutlich, dass beide Frameworks bei niedriger Last (Testplan 1) nahezu identische Werte von etwa 105 req/s erreichen. Mit zunehmender Last (Testpläne 2 und 3) zeigt sich, dass Quarkus in Testplan 2 leicht geringere Durchsatzwerte als Spring Boot erreicht (629,9 vs. 645 req/s), während es in Testplan 3 mit 369,4 req/s etwas hinter dem Wert von Spring Boot (405,1 req/s) liegt.

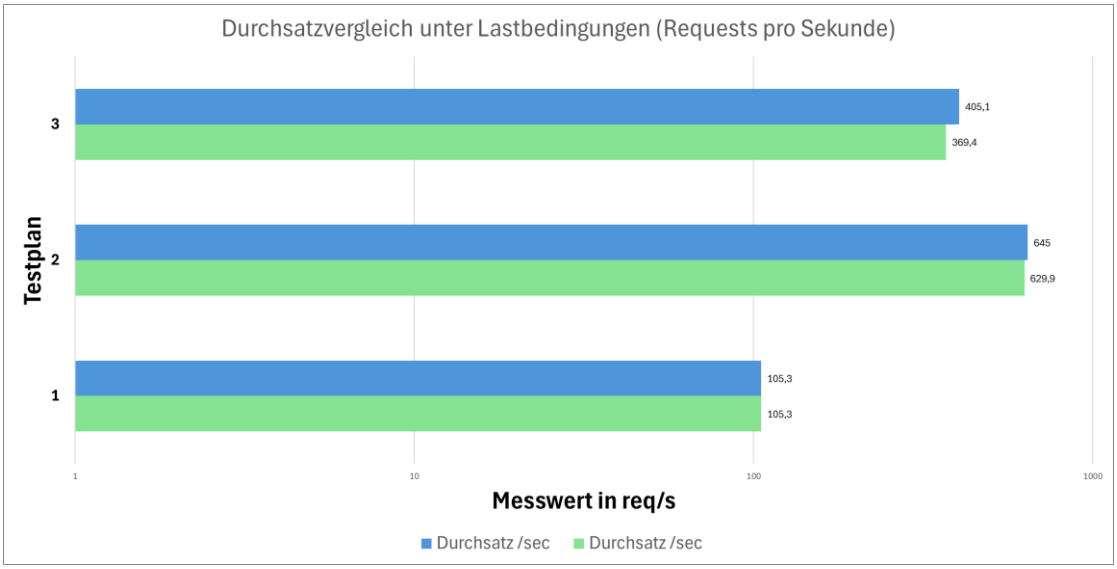
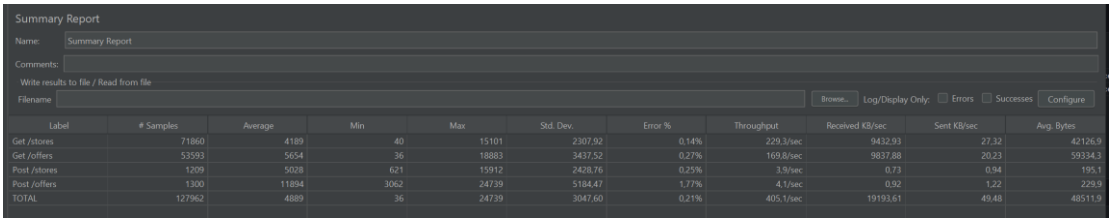


Abbildung 20: Durchsatzvergleich unter Lastbedingungen

Trotz dieser Abweichungen bleibt der Gesamtdurchsatz bei beiden Frameworks in einem vergleichbaren Bereich. Hinsichtlich der Stabilität zeigen sich insgesamt sehr geringe Fehlerraten. In den meisten Testdurchläufen lag die Fehlerquote bei 0 %, was auf eine robuste Verarbeitung der Anfragen hinweist. Eine Ausnahme bildet Testplan 3 mit Quarkus, wo eine Fehlerquote von 0,21 % verzeichnet wurde (*vgl. Abbildung 21* – JMeter Summary Report). Diese Abweichung kann auf die höhere gleichzeitige Last und die Post-Anfragen zurückgeführt werden, welche in diesem Szenario besonders ressourcenintensiv sind.

Der minimale Unterschied in der Fehlerrate bei Quarkus im dritten Testplan bleibt im akzeptablen Bereich und mindert die Gesamtstabilität der Anwendung nicht wesentlich.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
Get /stores	71860	4189	40	15101	2307,92	0,14%	229,3/sec	9432,93	27,32	42126,9
Get /offers	53593	5654	36	18883	3437,52	0,27%	169,8/sec	9837,88	20,23	59334,3
Post /stores	1209	5028	621	15912	2428,76	0,25%	3,9/sec	0,73	0,94	195,1
Post /offers	1300	11894	3062	24739	5184,47	1,77%	4,1/sec	0,92	1,22	229,9
TOTAL	127962	4889	36	24739	3047,60	0,21%	405,1/sec	19193,61	49,48	48511,9

Abbildung 21: JMeter - Summary-Report Testplan 3 Quarkus

5.3 Ressourcenverbrauch

Zur ganzheitlichen Bewertung der Frameworks unter Lastbedingungen wird im Folgenden der Ressourcenverbrauch analysiert. Dabei liegt der Fokus auf der CPU-Auslastung und dem Speicherbedarf der Microservices im Native-Modus über die drei definierten Lastszenarien (Testpläne 1–3). Die Daten wurden mit cAdvisor und Grafana erhoben; für jede Testlaufdauer von fünf Minuten sind der durchschnittliche (Mean) sowie der maximale (Max) Verbrauch ermittelt worden. Diese Kennzahlen geben Aufschluss darüber, wie effizient Quarkus und Spring Boot ihre Rechen- und Speicherressourcen nutzen und wie stabil sie unter zunehmender Last operieren.

5.3.1 CPU-Verbrauch

Abbildung 22 zeigt die gemessene CPU-Auslastung der beiden Microservices (Offer-Service und Store-Service) in den drei Testplänen. Die grünen Balken repräsentieren jeweils die Ergebnisse von Spring Boot (Native-Modus), die blauen Balken die von Quarkus (Native-Modus). Pro Testplan ist jeweils der Mean-Wert (mittlere Auslastung über die 300-Sekunden-Periode) sowie der Max-Wert (Spitzenlast) dargestellt.

- **Testplan 1 (leichte Last):**

Spring Boot erreicht im Offer-Service eine mittlere CPU-Auslastung von 5,45 % und eine maximale von 11,30 %, während Quarkus mit 4,64 % (Mean) bzw. 7,44 % (Max) noch etwas sparsamer agiert. Im Store-Service liegen beide Frameworks sehr niedrig (Spring Boot: 0,86 %/0,90 %; Quarkus: 0,54 %/0,45 %).

- **Testplan 2 (mittlere Last):**

Unter erhöhter Belastung klettert die Auslastung deutlich: Beim Offer-Service erreicht Spring Boot im Mittel 59,5 % CPU mit Spitzen von 130 %, Quarkus liegt hier bei 69,5 % Mean und 126 % Max. Der Store-Service beansprucht im Testplan 2 im Mittel 31,6 % (Spring Boot) bzw. 38,9 % (Quarkus) mit Maximalwerten von 90,3 % bzw. 64 %.

- **Testplan 3 (hohe Last):**

Auch im stärksten Szenario bleibt die Skalierung beider Frameworks funktionsfähig: Die mittlere CPU-Last im Offer-Service beträgt 56,4 % (Spring Boot) und 65,3 % (Quarkus), mit Spitzen von 123 % bzw. 121 %. Beim Store-Service sind es 56,2 %/61,3 % (Mean) und 116 %/113 % (Max) für Spring Boot bzw. Quarkus.

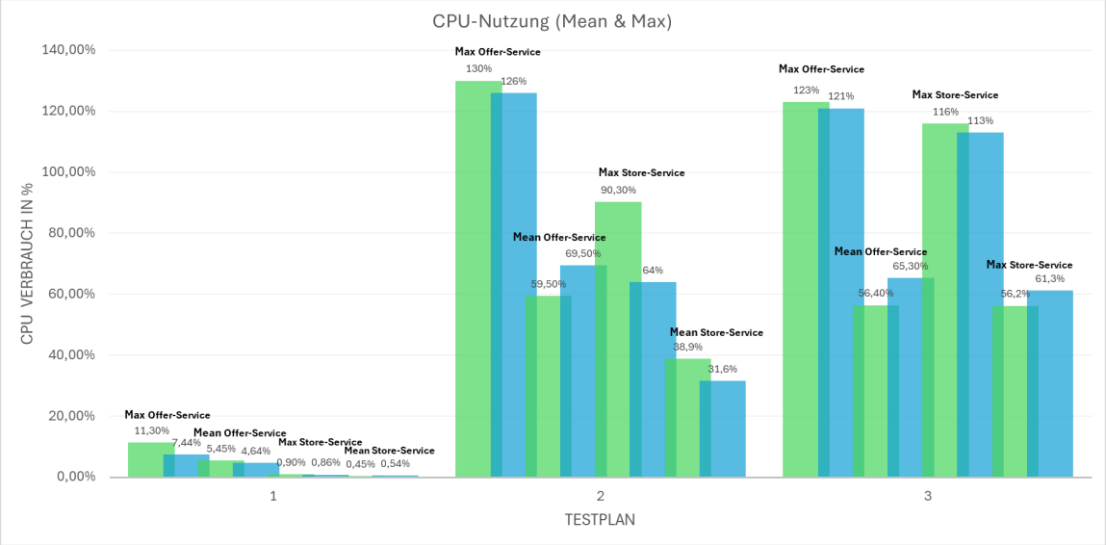


Abbildung 22: CPU-Verbrauchsvergleich

Quarkus zeigt bei leichter Last eine geringere CPU-Bindung, schlägt sich aber auch in anspruchsvolleren Szenarien gut und bleibt nahe am Verhalten von Spring Boot. Insbesondere die etwas niedrigeren Maximalwerte im Offer-Service deuten auf eine stabile Ressourcenverwaltung hin. Insgesamt lassen sich keine dramatischen Vorteile eines Frameworks gegenüber

dem anderen in Bezug auf CPU-Effizienz ausmachen, jedoch demonstriert Quarkus in Testplan 1 eine leicht bessere Sparsamkeit.

5.3.2 Speicherverbrauch

Abbildung X zeigt den RAM-Verbrauch (Mean & Max) der beiden Microservices (Offer-Service und Store-Service) unter den drei definierten Lastszenarien. Die grünen Balken stehen für Spring Boot (Native-Modus), die blauen Balken für Quarkus (Native-Modus).

- **Testplan 1 (leichte Last):**

Im Offer-Service liegt der durchschnittliche RAM-Verbrauch bei etwa 76 MiB (Spring Boot) gegenüber 68 MiB (Quarkus). Die Maximalwerte betragen 110 MiB für Spring Boot und 95 MiB für Quarkus.

Im Store-Service ist der Mittelwert deutlich niedriger: 106 MiB (Spring Boot) versus 56 MiB (Quarkus), mit Spitzen von 109 MiB beziehungsweise 66 MiB.

- **Testplan 2 (mittlere Last):**

Unter moderater Belastung erhöht sich der RAM-Bedarf: Im Offer-Service verbraucht Spring Boot durchschnittlich 488 MiB und erreicht Maxima von 654 MiB, Quarkus liegt mit 394 MiB (Mean) und 488 MiB (Max) etwas darunter.

Im Store-Service steigen die Werte auf 187 MiB (Mean) und 349 MiB (Max) für Spring Boot sowie 158 MiB (Mean) und 301 MiB (Max) für Quarkus.

- **Testplan 3 (hohe Last):**

Bei maximaler Auslastung erreicht der Offer-Service Spitzenverbräuche von 783 MiB (Mean) und 790 MiB (Max) mit Spring Boot. Quarkus weist mit 506 MiB (Mean) und 521 MiB (Max) einen leicht höheren Durchschnittsverbrauch, aber vergleichbare Maximalwerte auf.

Der Store-Service konsolidiert sich auf 476 MiB/689 MiB (Mean/Max) für Spring Boot und bleibt bei 400 MiB für beide Kennzahlen mit Quarkus.

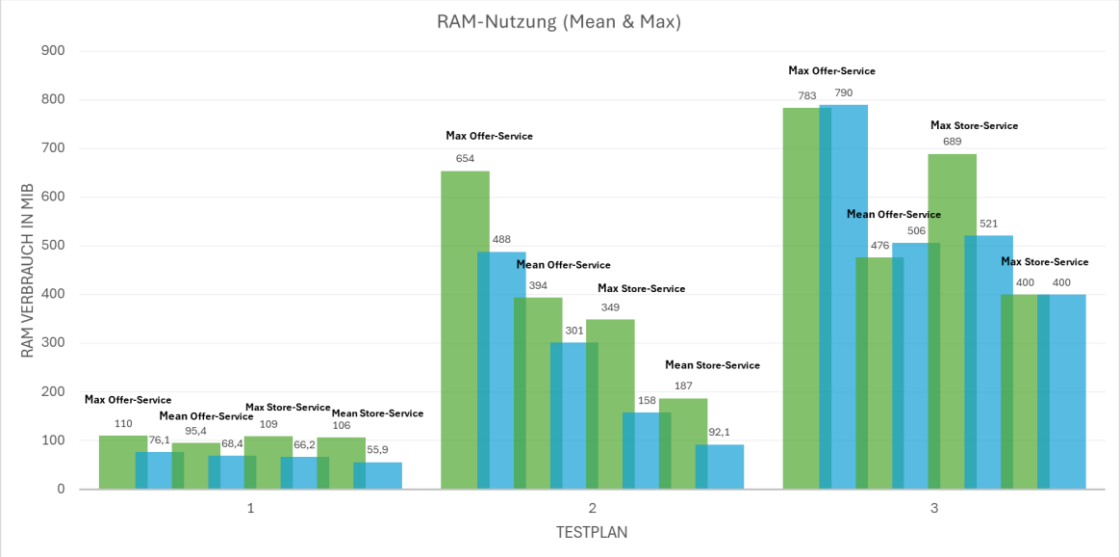


Abbildung 23: RAM-Verbrauch

Der RAM-Verbrauch steigt mit zunehmender Last erwartungsgemäß kontinuierlich an. In allen Szenarien verwendet Quarkus im Mittel und bei Spitzenlasten weniger Arbeitsspeicher als Spring Boot, mit Ausnahme des Offer-Services im Testplan 3, wo Quarkus einen geringfügig höheren mittleren Verbrauch aufweist. Insgesamt demonstriert Quarkus eine tendenziell sparsamere Speicherverwaltung, was insbesondere für Umgebungen mit begrenzten Ressourcen von Vorteil sein kann.

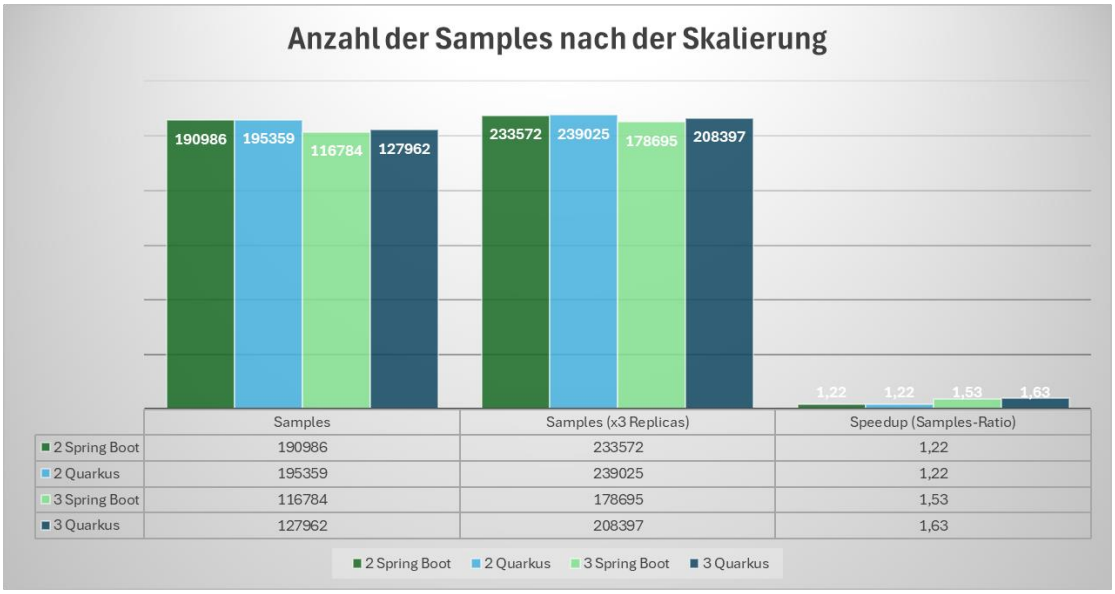
5.4 Skalierung

Zur Untersuchung der Skalierbarkeit wurden die Services im Native-Modus sowohl mit Quarkus als auch mit Spring Boot in einem Docker-Swarm-Cluster betrieben. Dabei wurde Testplan 2 und Testplan 3 jeweils auf drei Replikate des Store-Service und des Offer-Service hochgefahren. Im Folgenden wird die Effizienz dieser Skalierung sowie der resultierende Ressourcenverbrauch beurteilt.

5.4.1 Anzahl der Samples nach der Skalierung

Abbildung 24 vergleicht die Gesamtzahl der verarbeiteten Anfragen („Samples“) in Testplan 2 und Testplan 3, jeweils einmal mit einem einzelnen Service-Replikat und einmal mit drei gleichzeitigen Repliken im Docker-Swarm. Die grünen Balken repräsentieren dabei die Ergebnisse für Spring Boot, die blauen Balken für Quarkus; jeder Testlauf hatte eine feste Dauer von fünf Minuten.

- **Testplan 2:**
  - Mit einem Replikat wurden 190 986 (Spring Boot) respektive 195 359 (Quarkus) Samples verarbeitet.
  - Nach Skalierung auf drei Rep-likate stieg die Anzahl auf 233 572 (Spring Boot) bzw. 239 025 (Quarkus).
  - Die Speedup-Kennzahl (Samples-Ratio) beträgt 1,22× für beide Frameworks und zeigt damit eine nahezu lineare Erhöhung des verarbeiteten Anfragevolumens.
- **Testplan 3:**
  - Einzeln schafften die Dienste 116 784 (Spring Boot) bzw. 127 962 (Quarkus) Samples.
  - Mit drei Repliken erhöhte sich das Volumen auf 178 695 (Spring Boot) und 208 397 (Quarkus).
  - Hier ergeben sich Speedups von 1,53× für Spring Boot und 1,63× für Quarkus, was auf eine etwas bessere Skalierungseffizienz von Quarkus unter extrem hoher Last hinweist.



	Samples	Samples (x3 Replicas)	Speedup (Samples-Ratio)
2 Spring Boot	190986	233572	1,22
2 Quarkus	195359	239025	1,22
3 Spring Boot	116784	178695	1,53
3 Quarkus	127962	208397	1,63

Abbildung 24: Anzahl der Samples nach der Skalierung

Diese Ergebnisse belegen, dass beide Frameworks in der Lage sind, durch horizontale Replikation die verarbeitete Anfragezahl signifikant zu steigern. Quarkus zeigt dabei in Testplan 3 einen leicht höheren Skalierungsgewinn.

5.4.2 Antwortzeit nach der Skalierung

In *Abbildung 25* sind die Speedup-Faktoren dargestellt, die sich durch den Betrieb mit drei Replikaten im Vergleich zu einem einzelnen Service-Pod ergeben. Als Kennzahl wurde jeweils das Verhältnis

$$Speedup = \frac{\text{Antwortzeit bei 1 Replica}}{\text{Antwortzeit bei 3 Replicas}}$$

berechnet.

Ergebnisse im Detail:

- **Durchschnittliche Antwortzeit (Average):**
  - Testplan 2 erreicht Spring Boot einen Speedup von ≈ 3,66×, Quarkus sogar ≈ 6,85×.
  - Testplan 3 liegen die Faktoren bei ≈ 1,62× (Spring Boot) und ≈ 1,80× (Quarkus).

- Dieser Rückgang des Speedups in Testplan 3 illustriert, dass die Skalierung bei sehr hoher Last weniger lineare Gewinne liefert.

➢ **Median der Antwortzeit:**

- Testplan 2: Speedup von 5,36× (Spring Boot) und 2,71× (Quarkus).
- Testplan 3: Speedup von 2,31× (Spring Boot) und 3,04× (Quarkus).
- Quarkus erzielt im Median bei extremer Last (Testplan 3) einen höheren Skalierungsgewinn, während Spring Boot im moderaten Szenario (Testplan 2) stärker profitiert.

➢ **95%-Perzentil:**

- Testplan 2: 2,46× für Spring Boot versus 8,23× für Quarkus – deutliche Verbesserung der Worst-Case-Antwortzeiten bei Quarkus.
- Testplan 3: beide Frameworks liegen knapp über 1×, was zeigt, dass Ausreißerzeiten nur geringfügig durch Replikation reduziert werden.

➢ **Min und Max:**

- Die Speedup-Faktoren für Min und Max sind jeweils 1×, da sich die extremsten einzelnen Antwortzeitwerte durch horizontale Skalierung nicht verändern.

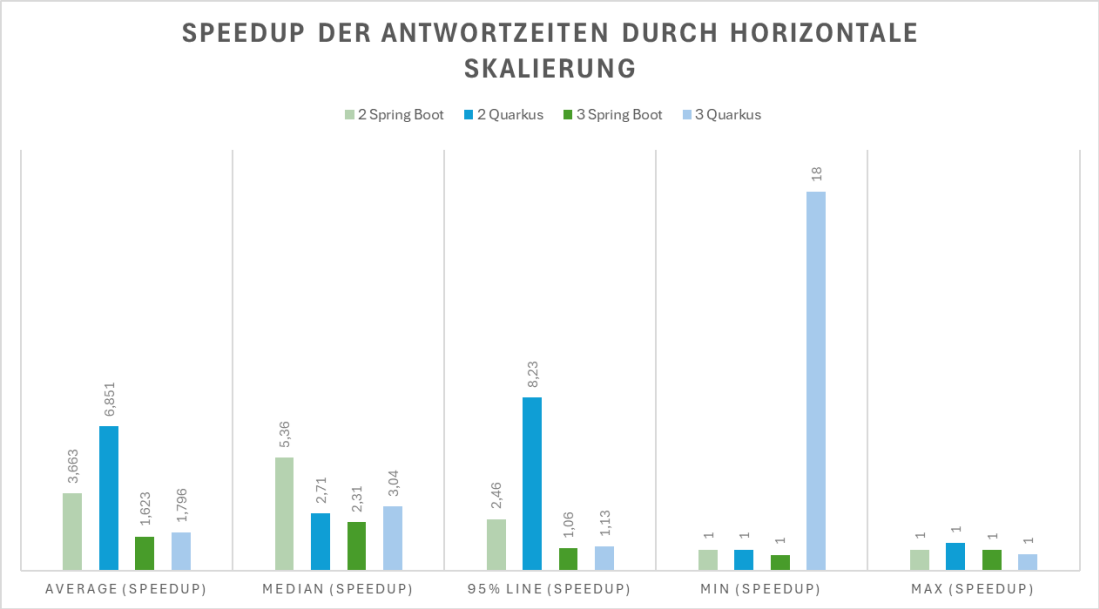


Abbildung 25: Speedup der Antwortzeiten nach der Skalierung

Die Ergebnisse zeigen, dass horizontale Skalierung die mittleren und besonders die perzentilen Antwortzeiten merklich verbessert. Quarkus zeigt in Testplan 2 einen besonders hohen Gewinn im Durchschnitt und im 95%-Perzentil, während Spring Boot im Median des moderaten Szenarios leicht stärker skaliert. In Testplan 3 – der im Vergleich zu Testplan 2 deutlich mehr Schreib-operationen (POST-Anfragen) enthält – fällt der Skalierungs-gewinn beider Frameworks spürbar ab und nähert sich wieder einem Speedup von 1× an. Diese Abschwächung ist vor allem auf die erhöhten Datenbank-Synchronisationskosten und Schreib-Overheads zurückzuführen. Quarkus erreicht jedoch auch in diesem Szenario etwas konsistentere Verbesserungen der Antwortzeiten durch zusätzliche Replikate als Spring Boot.

5.4.3 Durchsatz und Fehlerquote nach der Skalierung

Abbildung X vergleicht für Testplan 2 und 3 jeweils den Durchsatz (req/s), den Speedup sowie die relative Änderung der Fehlerquote, wenn die Anzahl der Service-Replikate von einem auf drei erhöht wird. Die hellen Balken stehen für Testplan 2, die dunklen Balken für Testplan 3; jeweils in Grün für Spring Boot und Blau für Quarkus.

- Durchsatz (req/s):
  - Testplan 2: Spring Boot wächst von 630 req/s auf 773,8 req/s (Speedup 1,23×), Quarkus von 507 req/s auf 792 req/s (ebenfalls 1,23×).
  - Testplan 3: Spring Boot steigert sich von 369,4 req/s auf 580,8 req/s (1,57×), Quarkus von 405,1 req/s auf 674,7 req/s (1,67×).

Dies zeigt, dass beide Frameworks durch Drittelung der Replikate einen deutlich höheren Gesamtdurchsatz erzielen, wobei Quarkus in beiden Szenarien bessere Skalierungsgewinne bietet.

- Speedup (req/s):

Die Speedup-Faktoren liegen zwischen 1,23× und 1,67× und fallen in Testplan 3 höher aus als in Testplan 2, da hier mehr POST-Anfragen (Schreib-operationen) abgewickelt werden und zusätzliche Replikate die Schreib-last effizienter verteilen.

- Fehlerquote (relative Änderung):

Für Spring Boot bleibt die Fehlerquote nach Skalierung in beiden Testplänen unverändert auf 0 (Speedup 1×).

Quarkus weist in Testplan 2 ebenfalls keine Veränderung auf (1×), zeigt jedoch in Testplan 3 eine 8,6-fache Zunahme der Fehlerquote (von 0,21% auf 1,81%), was auf erhöhten Synchronisations- und Schreib-overhead unter maximaler Belastung hindeutet.

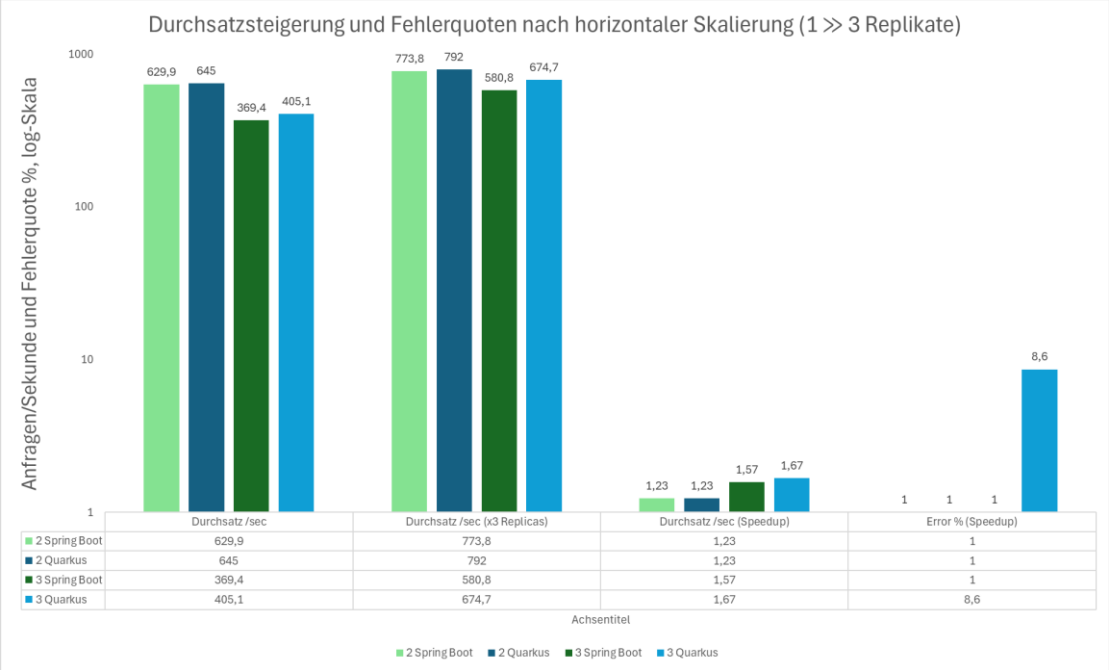


Abbildung 26: Durchsatz- und Fehlerquotevergleich nach der Skalierung

Die horizontale Skalierung auf drei Replikate führt zu einem signifikanten Durchsatzgewinn bei beiden Frameworks, mit einem leichten Vorteil für Quarkus in puncto Skalierungseffizienz. Allerdings zeigt sich bei Quarkus unter extremer Last (Testplan 3) eine deutliche Erhöhung der Fehlerquote, während Spring Boot hier stabiler bleibt. Dies unterstreicht den Trade-off zwischen maximaler Performance-Steigerung und Ausfallsicherheit unter Höchstlast.

5.4.4 Ressourcenverbrauch nach der Skalierung

Um zu beurteilen, ob sich die horizontale Verteilung der Last auch ressourcenseitig lohnt, wurden die mittleren und maximalen CPU- sowie RAM-Verbrauchswerte sämtlicher Replikate pro Service aufsummiert und mit den Ausgangswerten eines Ein-Replica-Betriebs verglichen.

Die absolute Differenz ergibt sich als

$$\Delta_{abs} = V_{1\,Replica} - \sum_{i=1}^{3} V_{Replica_i},$$

die relative Abweichung (in %) als

$$\Delta_{rel} = \frac{\Delta_{abs}}{V_{1\,Replica}}.$$

Die folgende *Abbildung 27* zeigt die relative Veränderung des Ressourcenverbrauchs (CPU- und RAM-Nutzung) nach der Skalierung der Dienste auf drei Replikate. Positive Prozentwerte deuten auf eine Einsparung hin – also eine bessere Ressourcennutzung trotz Mehrinstanzen. Negative Werte hingegen zeigen einen Mehraufwand, bei dem der Gesamtkonsum über den des Einzel-Deployments hinausgeht.

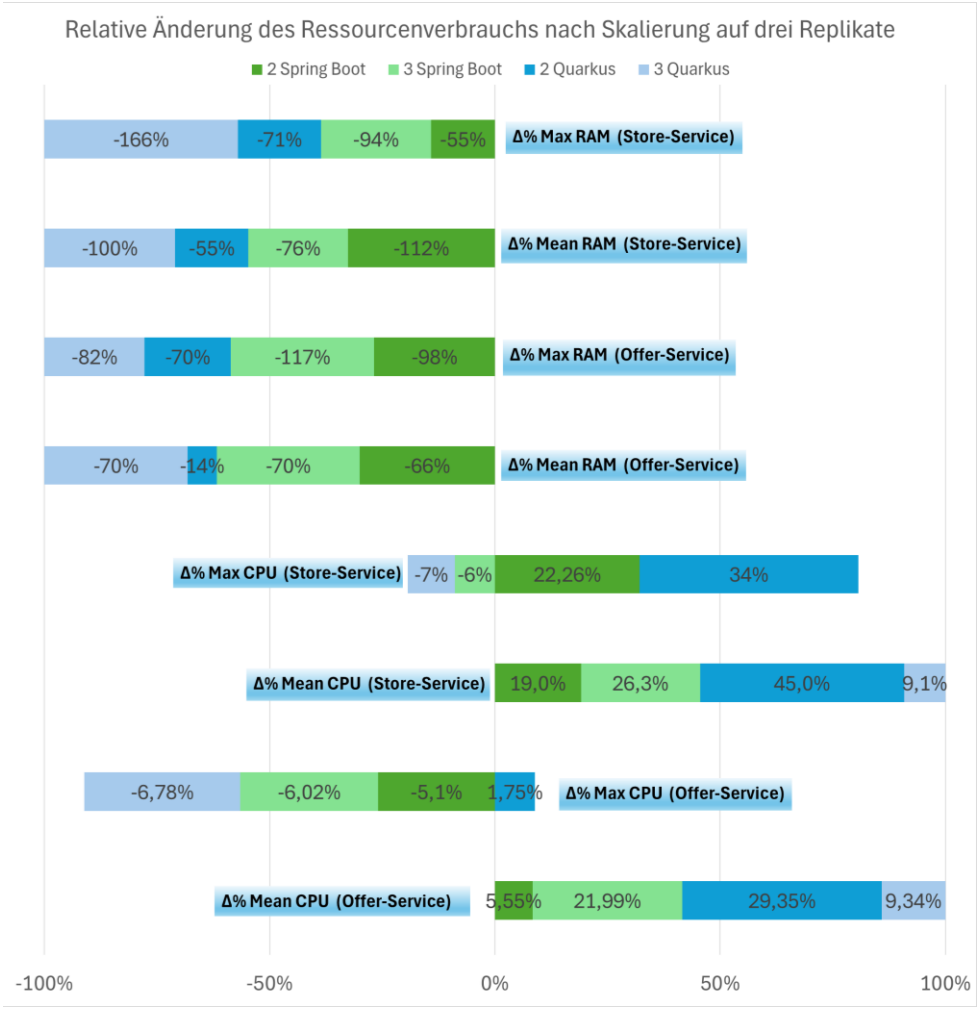


Abbildung 27: Relative Änderung des Ressourcenverbrauchs nach Skalierung

- CPU-Verbrauch
  - Mean CPU % (Offer-Service):

    Quarkus erreicht hier in Testplan 2 mit +29,35 % eine deutlich höhere Einsparung als Spring Boot mit nur +5,55 %.

    Auch in Testplan 3 liegt Quarkus mit +9,34 % leicht vor Spring Boot (+21,99 %), wobei die Einsparung bei beiden sinkt.

- Max CPU % (Offer-Service):

  Bei den Maximalwerten ist der Unterschied weniger stark ausgeprägt. Spring Boot schneidet etwas besser ab (bis −6,78 % Mehraufwand bei Quarkus), aber der Unterschied bleibt gering.

- Mean CPU % (Store-Service):

  In beiden Testplänen zeigt Quarkus eine überlegene Einsparung: +45 % (TP2) und +9,1 % (TP3), während Spring Boot hier nur +26,3 % (TP2) und +19 % (TP3) erreicht.

- Max CPU % (Store-Service):

  Auch hier dominiert Quarkus mit +34 % (TP2) gegenüber +22,26 % bei Spring Boot.

Quarkus nutzt die CPU-Ressourcen nach der Skalierung deutlich effizienter als Spring Boot, insbesondere bei moderater Last (Testplan 2). In hochbelasteten Szenarien (Testplan 3) sinkt der Vorteil, bleibt aber messbar.

➢ RAM-Verbrauch

- Mean & Max RAM (Offer-Service und Store-Service):

  In allen RAM-Metriken ist ein starker Mehraufwand nach der Skalierung zu beobachten – bei beiden Frameworks.

  Besonders auffällig ist jedoch, dass Spring Boot in Testplan 3 im Max RAM Store-Service eine extrem negative Bilanz aufweist (−166 %), während Quarkus mit −71 % weniger RAM verschwendet.

Beide Frameworks leiden unter stark erhöhtem Speicherverbrauch bei der Skalierung. Dennoch ist Quarkus hier minimal effizienter, insbesondere im Vergleich der Maximalwerte. Der Unterschied bleibt aber moderat.

6 Zusammenfassung und Ausblick

Diese Arbeit hatte das Ziel, die beiden Java-Frameworks Quarkus und Spring Boot im Kontext einer cloudnativen Microservice-Anwendung vergleichend zu analysieren. Mithilfe eines praxisorientierten Prototyps – dem Promotion-Management-System – wurden zentrale Leistungsmerkmale wie Startzeit, Buildzeit, Image-Größe, Antwortzeit, Durchsatz, Fehlerquote, Ressourcenverbrauch sowie Skalierbarkeit systematisch untersucht. Die Durchführung unter realitätsnahen Lastbedingungen im Native-Modus, ergänzt durch Skalierungsszenarien mittels Docker Swarm, lieferte dabei aussagekräftige empirische Ergebnisse.

Im Vergleich zeigt sich, dass Quarkus in vielen zentralen Aspekten effizienter arbeitet als Spring Boot. Besonders bei Startzeit, Image-Größe und Speicherverbrauch konnte Quarkus durch seine native Kompilierung mit GraalVM signifikante Vorteile erzielen. Auch bei der horizontalen Skalierung erwies sich Quarkus als tendenziell performanter, wenngleich unter hoher Last synchronisationsbedingte Fehlerzunahmen beobachtet wurden. Spring Boot hingegen bewährte sich als stabil und zuverlässig, insbesondere in stark belasteten Szenarien mit komplexeren Schreiboperationen.

Die Ergebnisse legen nahe, dass Quarkus insbesondere für schlanke, cloudnative Anwendungen mit Fokus auf Performance und Skalierbarkeit geeignet ist, während Spring Boot weiterhin als robuste Allzwecklösung überzeugt, besonders wenn Stabilität und Entwicklerfreundlichkeit im Vordergrund stehen.

Ausblick

Die durchgeführten Analysen liefern eine fundierte Vergleichsbasis, lassen jedoch auch Raum für weiterführende Untersuchungen. In zukünftigen Arbeiten könnten beispielsweise weitere Aspekte wie Entwicklungsaufwand, Fehlertoleranz unter Netzausfallbedingungen, oder die Integration externer Dienste betrachtet werden. Auch eine Langzeitanalyse im produktiven Dauerbetrieb sowie der Einsatz in Serverless-Architekturen oder auf Kubernetes bieten Potenzial für weiterführende Forschung. Zudem wäre ein Vergleich mit anderen modernen Frameworks

wie **Micronaut** oder **Helidon** denkbar, um die Positionierung von Quarkus und Spring Boot im weiteren Java-Ökosystem zu schärfen.

Insgesamt trägt diese Arbeit dazu bei, die Wahl geeigneter Frameworks im Cloud-Zeitalter evidenzbasiert zu unterstützen – ein Aspekt, der angesichts wachsender Systemkomplexität und gestiegener Anforderungen an Effizienz und Skalierbarkeit zunehmend an Bedeutung gewinnt.

Literaturverzeichnis

C, M., 2023. *Spring Boot vs Quarkus: Performance comparison for hello world case.* [Online] Available at: https://medium.com/deno-the-complete-reference/spring-boot-vs-quarkus-performance-comparison-for-hello-world-case-e466d3630329
[Zugriff am 13 Februar 2025].

Geeksforgeeks, 2025. *Spring - Setter Injection vs Constructor Injection.* [Online] Available at: https://www.geeksforgeeks.org/spring-setter-injection-vs-constructor-injection/ [Zugriff am 22 Januar 2025].

Lagnada), K. (., 2024. *Spring vs. Spring Boot: Choosing the Best Java Framework for Your Project.* [Online] Available at: https://www.kapresoft.com/java/2024/03/06/spring-vs-spring-boot.html [Zugriff am 02 Febraur 2025].

Minh, N. H., kein Datum *Spring Boot version history.* [Online] Available at: https://www.codejava.net/frameworks/spring-boot/spring-boot-version-history [Zugriff am 01 Feburar 2025].

Mitropolitsky, M., Simion-Constantinescu, A., Kutsarova, V. & Sellik, H., kein Datum *Spring Boot - production-grade Spring-based Applications that you can "just run".* [Online] Available at: https://se.ewi.tudelft.nl/desosa2019/chapters/spring-boot/ [Zugriff am 02 Februar 2025].

Quarkus, kein Datum *Container First.* [Online] Available at: https://quarkus.io/container-first/ [Zugriff am 04 Febraur 2025].

Quarkus, kein Datum *Quarkus Key Features.* [Online] Available at: https://quarkus.io/ [Zugriff am 13 Febraur 2025].

Quarkus, kein Datum *Quarkus Releases.* [Online] Available at: https://github.com/quarkusio/quarkus/releases [Zugriff am 03 Februar 2025].

Spring, kein Datum *Web MVC framework.* [Online] Available at: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html [Zugriff am 22 Januar 2025].

Spring, kein Datum *Introduction to the Spring Framework.* [Online] Available at: https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/overview.html [Zugriff am 22 Januar 2025].

Spring, kein Datum *Spring Framework 1.0 Final Released.* [Online] Available at: https://spring.io/blog/2004/03/24/spring-framework-1-0-final-released [Zugriff am 20 Januar 2025].

Štefanko, M. & Martiška, J., 2025. *Quarkus in Action.* s.l.:Red Hat Developer.

Ter, M. S., 2024. *Quarkus vs. Micronaut vs. Spring Boot: A Comparative Guide for Java Developers.* [Online] Available at: https://master-spring-ter.medium.com/quarkus-vs-micronaut-vs-spring-boot-a-comparative-guide-for-java-developers-8ece4538b883 [Zugriff am 13 Februar 2025].

Victor, J., 2023. *Spring Framework, History, and Its Structure.* [Online] Available at: https://dev.to/jeanv0/spring-framework-history-and-its-structure-361 [Zugriff am 21 Januar 2025].

Zanetti, E., 2024. *Spring Boot and Quarkus: Comparing Performance and Usage.* [Online] Available at: https://medium.com/%40erickzanetti/spring-boot-and-quarkus-comparing-

performance-and-usage-3610f3dd9719
[Zugriff am 13 Februar 2025].

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original