Performance Optimization for JValue: Implementing and Evaluating Microservice Architectures

MASTER THESIS

Marco Alexander Döll

Eingereicht am 20. März 2024

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open Source Software

Betreuer:
Georg Schwarz
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Technische Fakultät

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 20. März 2024

Lizenz

Diese Arbeit unterliegt der Creative Commons Attribution 4.0 International Lizenz (CC BY 4.0), https://creativecommons.org/licenses/by/4.0/

Erlangen, 20. März 2024

Abstract

The emergence of microservice architectures presents a paradigm shift in application development, offering a modular approach where applications are composed of small, independent services, that focus on one specific task. While this architecture fosters many advantages regarding flexibility, scalability, and maintainability, it introduces challenges, particularly in testing.

Crucially, performance testing serves as a cornerstone in the decision-making process for choosing architecture, technology, frameworks, and infrastructure. Thus, having quantifiable and comparable metrics is essential to build a solid foundation for decision-making. Performance testing becomes paramount to ensure adherence to non-functional requirements like throughput and response time, identifying potential bottlenecks in a distributed environment.

In this thesis, the examination focuses on the broader issue of testability and performance bottlenecks within microservice architectures, through the lens of the JValue application. The JValue Hub serves as a tool for developers and data scientists in the context of open data application development, enabling users to collect open data from APIs, to apply processing and cleaning processes, and to make the transformed data available for further processing.

Given the current architecture of the JValue Hub, there exists a bottleneck. Each generated pipeline run within the JValue architecture spawns an individual subprocess, which demands a substantial allocation of resources. The initialization and management of subprocesses results in significant overhead, including memory usage, process creation and termination, as well as general subprocess management by the main process. This architectural limitation poses a considerable challenge to the overall system performance and scalability of the JValue Hub.

This thesis focuses on comparing three different microservice architectures, analyzing their performance regarding different metrics and aspects, to guide informed architectural decisions. Furthermore, this research aims to provide valuable insights into enhancing the performance of the JValue application.

Zusammenfassung

Die Etablierung von Microservice Architekturen markiert Paradigmenwechsel in der Softwareentwicklung, indem sie einen modularen Ansatz bieten, bei dem Anwendungen aus kleinen, unabhängigen Diensten zusammengesetzt sind, die sich auf eine spezifische Aufgabe konzentrieren. Obwohl diese Architektur viele Vorteile in Bezug auf Flexibilität, Skalierbarkeit und Wartbarkeit bietet, bringt sie Herausforderungen mit sich, insbesondere im Bereich der Testbarkeit.

Performance-Tests stellen ein Eckpfeiler im Entscheidungsprozess für die Wahl von Architektur, Technologie, Frameworks und Infrastruktur dar. Daher ist es von entscheidender Bedeutung, quantifizierbare und vergleichbare Metriken zu haben, um eine solide Grundlage für Entscheidungen zu schaffen. Performance-Tests werden unerlässlich, um die Einhaltung nicht-funktionaler Anforderungen wie Durchsatz und Antwortzeit sicherzustellen und potenzielle Engpässe zu identifizieren.

In dieser Arbeit liegt der Fokus auf dem generellen Problem der Testbarkeit und Bottlenecks innerhalb von Microservice Architekturen am Beispiel der Anwendung JValue. Der JValue Hub ermöglicht es Benutzern, offene Daten von APIs zu sammeln, Verarbeitungs- und Bereinigungsprozesse anzuwenden und die transformierten Daten für weitere Verarbeitungsvorgänge verfügbar zu machen.

In der aktuellen Architektur des JValue Hubs gibt es einen Engpass. Jeder erzeugte Pipeline-Run innerhalb der Ist-Architektur erzeugt einen individuellen Subprozess, der einen hohen Ressourcenaufwand erfordert. Die Initialisierung und Verwaltung von Subprozessen führt zu erheblichem Overhead. Diese Einschränkung stellt eine erhebliche Herausforderung für die Performance und Skalierbarkeit der Anwendung JValue dar.

Diese Arbeit konzentriert sich darauf, drei verschiedene Microservice Architekturen zu vergleichen und ihre Performance hinsichtlich verschiedener Metriken und Aspekte zu analysieren, um informierte Entscheidungen hinsichtlich der Architektur zu treffen. Darüber hinaus zielt diese Forschung darauf ab, wertvolle Einblicke in die Verbesserung der Performance der Anwendung JValue zu bieten.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**
2.1 Kontext: JValue Projekt 3
2.2 Microservices 5
2.3 Performance-Test 8
2.4 Locust 9
2.5 wrk2 10
2.6 RabbitMQ 10
2.7 Docker 11
2.8 Kubernetes 11
2.9 Node.js 12

3 Anforderungen **15**
3.1 Technische Anforderungen 16
3.2 Performance Anforderungen 16
3.3 Abgeleitete Szenarien 18
3.3.1 Szenario Response Time 19
3.3.2 Szenario Integration 21

4 Architekturen **25**
4.1 Ist-Architektur 26
4.2 Threadpool Architektur 27
4.3 Message Queue Architektur 29

5 Implementierung **31**
5.1 Threadpool Architektur 31
5.2 Message Queue Architektur 35
5.3 Evaluator 38
5.3.1 Infrastruktur und Umgebung 38
5.3.2 Performance-Tests 40
5.3.3 Parametrisierung 43

5.3.4 Umsetzungsdetails 44

6 Evaluation **49**
6.1 Ist-Architektur 51
6.2 Threadpool Architektur 53
6.3 Message Queue Architektur 56
6.4 Vergleich 58
6.4.1 Szenario Response Time 58
6.4.2 Szenario Integration 62
6.5 Anforderungen 65
6.5.1 Technische Anforderungen 65
6.5.2 Performance Anforderungen 65

7 Schlussfolgerung **69**
7.1 Limitierungen 69
7.2 Future Work 69
7.3 Empfehlung 70
7.4 Zusammenfassung 71

Appendices **75**
A Kubernetes Cluster Setup 77
B Jayvee Modelle 81
C Implementierung: Performance-Tests 84
D Implementierung: Aufbereitung der Antwortzeiten 87
E Ergebnisse: Vergleich der Architekturen 90
E.1 Testkonfiguration Minimum 90
E.2 Testkonfiguration Medium 91
E.3 Testkonfiguration Maximum 92

Literaturverzeichnis **93**

Abbildungsverzeichnis

2.1 Übersicht Microservice Architektur JValue Hub 5
2.2 Representational State Transfer (REST)-API Microservice Architektur in Anlehnung an Hong et al. (2018) 6
2.3 Messaging-Systeme Architektur in Anlehnung an Hong et al. (2018) 7
2.4 Event Loop Architektur in Node.js in Anlehnung an kinsta (2023) 12

3.1 FunktionsMASTeR Schablone (SOPHISTen, 2022) 15
3.2 Aufbau Testszenario Response Time in Anlehnung an (Czeraszkiweicz, 2015) 21
3.3 Aufbau Testszenario Integration in Anlehnung an (intersog, 2020) 23

4.1 Sequenzdiagramm Ist-Architektur - vereinfachte Darstellung . . . 26
4.2 Sequenzdiagramm Threadpool Architektur - vereinfachte Darstellung 28
4.3 Sequenzdiagramm Message Queue Architektur - vereinfachte Darstellung 30

5.1 Methode startSimpleRun 32
5.2 Block then 32
5.3 Methode runInterpreter 33
5.4 Ausführung eines Pipeline-Runs 34
5.5 Fehlerfall bei Pipeline Ausführung 34
5.6 Erfolgsfall bei Pipeline Ausführung 34
5.7 RunService Konstruktor 35
5.8 Konfiguration der Publisher 36
5.9 Senden einer Nachricht an die Warteschlange 36
5.10 Konfiguration des Consumers 37
5.11 Ausführung des Pipelines-Runs 37
5.12 Konfiguration des Consumers im Pipeline-Service 38
5.13 Sequenzdiagramm des Performance-Tests 41
5.14 Beispiel Antwort des Pipeline-Service nach Start eines Pipeline-Runs 42
5.15 Abspeichern der Antworten auf dem Test-System 43

5.16 Methode on start 44
5.17 Methode create pipeline and start run 45
5.18 Methode on stop 45
5.19 Aufbereitung der Ergebnisse 46
5.20 Berechnung der Antwortzeiten 46
5.21 Abspeichern der Antwortzeiten 47

6.1 Ist-Architektur Testkonfiguration Minimum 53
6.2 Threadpool Architektur Testkonfiguration Minimum 55
6.3 Threadpool Architektur Testkonfiguration Medium 55
6.4 Threadpool Architektur Testkonfiguration Maximum 55
6.5 Message Queue Architektur Testkonfiguration Mininum 57
6.6 Message Queue Architektur Testkonfiguration Medium 57
6.7 Message Queue Architektur Testkonfiguration Maximum 57
6.8 Szenario Integration Testkonfiguration Mininum 64
6.9 Szenario Integration Testkonfiguration Medium 64
6.10 Szenario Integration Testkonfiguration Maximum 64

1 Ergebnisse Testkonfiguration Minimum Datensatz Kategorie „Klein“ (Cars) 90
2 Ergebnisse Testkonfiguration Minimum Datensatz Kategorie „Mittel“ (Bookings) 90
3 Ergebnisse Testkonfiguration Minimum Datensatz Kategorie „Groß“ (Reviews) 90
4 Ergebnisse Testkonfiguration Medium Datensatz Kategorie „Klein“ (Cars) 91
5 Ergebnisse Testkonfiguration Medium Datensatz Kategorie „Mittel“ (Bookings) 91
6 Ergebnisse Testkonfiguration Medium Datensatz Kategorie „Groß“ (Reviews) 91
7 Ergebnisse Testkonfiguration Maximum Datensatz Kategorie „Klein“ (Cars) 92
8 Ergebnisse Testkonfiguration Maximum Datensatz Kategorie „Mittel“ (Bookings) 92
9 Ergebnisse Testkonfiguration Maximum Datensatz Kategorie „Groß“ (Reviews) 92

Tabellenverzeichnis

3.1 Konfiguration der Parameter für Testszenario Response Time . . . 19
3.2 Struktur der Datensätze 20
3.3 Konfiguration der Parameter für Testszenario Integration 22

5.1 Serververteilung der Virtuelle Maschinen (VMs) 39
5.2 Ressourcenverteilung der Testumgebung 44

6.1 Allgemeine Konfiguration der Parameter 50
6.2 Spezifische Konfiguration der Parameter für Testszenario Response Time 50
6.3 Spezifische Konfiguration der Parameter für Testszenario Integration 50
6.4 Aggregierte Ergebnisse Ist-Architektur Datensatz Kategorie „Klein“ (Cars) 52
6.5 Aggregierte Ergebnisse Ist-Architektur Datensatz Kategorie „Mittel“ (Bookings) 52
6.6 Aggregierte Ergebnisse Ist-Architektur Datensatz Kategorie „Groß“ (Reviews) 52
6.7 Aggregierte Ergebnisse Threadpool Architektur Datensatz Kategorie „Klein“ (Cars) 54
6.8 Aggregierte Ergebnisse Threadpool Architektur Datensatz Kategorie „Mittel“ (Bookings) 54
6.9 Aggregierte Ergebnisse Threadpool Architektur Datensatz Kategorie „Groß“ (Reviews) 54
6.10 Aggregierte Ergebnisse Message Queue Architektur Datensatz Kategorie „Klein“ (Cars) 56
6.11 Aggregierte Ergebnisse Message Queue Architektur Datensatz Kategorie „Mittel“ (Bookings) 56
6.12 Aggregierte Ergebnisse Message Queue Architektur Datensatz Kategorie „Groß“ (Reviews) 56
6.13 Aggregierte Ergebnisse Datensatz Kategorie „Klein“ (Cars) 59
6.14 Aggregierte Ergebnisse Datensatz Kategorie „Mittel“ (Bookings) . 59
6.15 Aggregierte Ergebnisse Datensatz Kategorie „Groß“ (Reviews) . . 59

6.16 Aggregierte Ergebnisse Datensatz Kategorie „Klein“ (Cars) 60
6.17 Aggregierte Ergebnisse Datensatz Kategorie „Mittel“ (Bookings) . 60
6.18 Aggregierte Ergebnisse Datensatz Kategorie „Groß“ (Reviews) . . 60
6.19 Aggregierte Ergebnisse Datensatz Kategorie „Klein“ (Cars) 61
6.20 Aggregierte Ergebnisse Datensatz Kategorie „Mittel“ (Bookings) . 61
6.21 Aggregierte Ergebnisse Datensatz Kategorie „Groß“ (Reviews) . . 61
6.22 Aggregierte Ergebnisse Testkonfiguration Minimum 63
6.23 Aggregierte Ergebnisse Testkonfiguration Medium 63
6.24 Aggregierte Ergebnisse Testkonfiguration Maximum 63
6.25 Technische Anforderungen 65
6.26 Performance Anforderungen Antwortzeiten 66
6.27 Performance Anforderungen Robustheit 66
6.28 Performance Anforderungen Verfügbarkeit 66
6.29 Performance Anforderungen Zuverlässigkeit 67
6.30 Performance Anforderungen Durchsatz 67

Akronyme

DSL Domain Specific Language

API Application Programming Interface

AMQP Advanced Message Queuing Protocol

HTTP Hypertext Transfer Protocol

REST Representational State Transfer

FIFO First In First Out

DTO Datenübertragungsobjekt

URL Uniform Resource Locator

VM Virtuelle Maschine

RPS Requests pro Sekunde

UUID Universally Unique Identifier

JSON JavaScript Object Notation

1 Einleitung

Microservice-basierte Systeme verfolgen den Ansatz, leichtgewichtige, wiederverwendbare und kleinteilige Services zu verwenden, die über Netzwerkmechanismen miteinander kommunizieren (Abdelfattah & Cerny, 2022). Diese Services sind loose gekoppelt und funktional separiert. Sie bilden die Grundlage für eine langfristige und skalierbare Entwicklung der Anwendung, da sie autonom entwickelt, betrieben und getestet werden können (Thönes, 2015).

Darüber hinaus erlaubt dieser Architekturstil die Verwendung der am besten geeigneten Programmiersprache und Technologie für die einzelnen Services. Daraus folgen weitere Vorteilen bezüglich der Skalierbarkeit und Fehlertoleranz. Dies hat dazu geführt, dass sich die Microservice Architektur als Standard in der Softwareentwicklung etabliert hat (Nasab et al., 2023).

Neben Vorteilen der vereinfachten Skalierbarkeit, Auswahl von Technologien sowie Entwicklung und Betrieb von Microservice-basierten Anwendungen sind für den Nutzer der Applikationen vor allem performancebezogene Aspekte wie Verfügbarkeit, Zuverlässigkeit, Antwortzeiten und Durchsatz relevant. Studien konnten bereits nachweisen, dass sich eine schlechte Performance negativ auf die Geschäftsergebnisse auswirkt. So legt Clark (2018) dar, dass die BBC für jede zusätzliche Sekunde, die das Laden der Website gedauert hat, 10 % an Nutzern verliert.

Dies verdeutlicht die kritische Rolle, die Performance in der Entwicklung von Microservice-basierten Anwendungen spielt und unterstreicht die Dringlichkeit, Performanceaspekte bei der Gestaltung und Optimierung von Systemen zu berücksichtigen.

Die vorliegende Untersuchung widmet sich der umfassenden Performance-Analyse der Open-Source-Anwendung JValue, mit einem besonderen Fokus auf die Evaluierung und Quantifizierung der Performance verschiedener Microservice Architekturen anhand der Metriken Antwortzeiten und Durchsatz sowie Aspekte der Verfügbarkeit, Zuverlässigkeit und Robustheit.

Diese Arbeit strebt an, eine Orientierungshilfe für Architekturentscheidungen zu

bieten. Der Versuchsaufbau soll darüber hinaus als Leitfaden dienen, um auch anderen Entwicklern bei eigenen Performance-Analysen ihrer Anwendung zu helfen.

In dieser Arbeit werden im ersten Kapitel die grundlegenden Konzepte und Technologien erläutert, auf denen die Architekturen und die Performance-Tests aufbauen. Anschließend werden die Anforderungen definiert, anhand derer die verschiedenen Architekturen verglichen und bewertet werden sollen. Daraufhin werden drei Architekturen vorgestellt, einschließlich der aktuellen Ist-Architektur und zwei alternativer Konzepte. Im nächsten Abschnitt wird dann die Implementierung der beiden alternativen Architekturen sowie der Versuchsaufbau der Performance-Tests beschrieben. Die Evaluation umfasst die Vorstellung und den Vergleich der Ergebnisse, die aus den durchgeführten Tests gewonnen wurden. Zusätzlich werden die Architekturen anhand der definierten Anforderungen gegenübergestellt und beurteilt. Abschließend werden in der Schlussfolgerung die Limitierungen der Studie diskutiert und Empfehlungen für zukünftige Arbeiten abgeleitet.

2 Grundlagen

In den nachfolgenden Abschnitten wird zunächst der Kontext der Untersuchung vorgestellt. Anschließend werden die zentralen Technologien und architektonischen Ansätze eingeführt, die innerhalb dieser Arbeit insbesondere im Versuchsaufbau und Durchführung, sowie der Implementierung der unterschiedlichen Microservice Architekturen und Performance-Tests eine entscheidende Rolle spielen.

Zusätzlich wird aufgezeigt, in welchem Zusammenhang die Technologien und Ansätze im Rahmen der Analyse angewandt werden. Diese Darstellung zielt darauf ab, einen umfassenden Überblick über die eingesetzten Technologien zu bieten und gleichzeitig auf deren spezifischen Beitrag zur Zielsetzung dieser Arbeit hinzuweisen. Darüber hinaus wird eine tiefgehende Verständnisgrundlage geschaffen, die es dem Leser ermöglicht, die implementierten Microservice Architekturen sowie die durchgeführten Performance-Tests und daraus resultierende Ergebnisse in ihrer Gesamtheit besser nachzuvollziehen.

2.1 Kontext: JValue Projekt

JValue Hub[1] ist eine Open-Source-Anwendung, die vom Lehrstuhl für Open-Source-Software an der Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) konzipiert und entwickelt wurde. Die Anwendung zielt darauf ab, das bevorzugte Tool für Entwickler und Data Scientists zu sein, wenn es um die Entwicklung von Anwendungen im Rahmen von offenen Daten geht. Der JValue Hub ermöglicht es seinen Anwendern, offene Daten von Application Programming Interfaces (APIs) zu sammeln, Verarbeitungs- und Bereinigungsprozesse anzuwenden und diese transformierten Daten anschließend für die weitere Nutzung zur Verfügung zu stellen.

Hierbei können mithilfe einer eigens entwickelten, Domain Specific Language (DSL) Jayvee[2] automatisierte Daten-Pipelines erstellt werden. Der Jayvee Interpreter bietet darüber hinaus die Möglichkeit, diese Daten-Pipelines auszuführen.

[1] https://jvalue.com/
[2] https://github.com/jvalue/jayvee

Folglich können Jayvee und der dazugehörige Interpreter dafür genutzt werden, Daten aus APIs abzurufen und diese anschließend für Aktivitäten, wie maschinelles Lernen, zu bereinigen und vorzubereiten.

Der JValue Hub besteht aus fünf Kernkomponenten, welche in eigenständige Microservices aufgeteilt sind. Diese Struktur ermöglicht eine eigenständige Entwicklung und den voneinander unabhängigen Betrieb der einzelnen Komponenten. Die Kommunikation der Dienste untereinander erfolgt über REST-APIs mittels Hypertext Transfer Protocol (HTTP).

Die Abbildung 2.1 zeigt vereinfacht die Microservice Architektur der Anwendung JValue.

Hub-Web

Der Dienst Hub-Web fungiert in der Rolle einer Bedienoberfläche für den Nutzer. Hier hat der Anwender die Möglichkeit, unterschiedliche Datenquellen und Pipelines zu erstellen sowie zu konfigurieren. Zusätzlich gewährt dieser Service eine Übersicht der aktuell verwendeten Datenquellen und deren zugehörigen Pipelines.

File-Service

Der File-Service fungiert als zentraler Dienst für die Verwaltung von Dateien und dient als Schnittstelle für den Zugriff auf gespeicherte Ergebnisse und Informationen innerhalb des Systems. Seine Verantwortlichkeiten umfassen die Speicherung, den Abruf und die Aktualisierung von Dateien, die im Zuge der Verarbeitung von Pipeline-Runs generiert werden.

Hub-Backend

Das Hub-Backend bildet das zentrale Rückgrat des JValue Systems und fungiert als Daten- und Steuerungsdienst. Der Fokus liegt in der Koordination des Datenflusses zwischen den verschiedenen Diensten, um eine effiziente Interaktion zwischen den Komponenten sicherzustellen.

Pipeline-Service

Der Pipeline-Service trägt die Verantwortung für die Erstellung und Verwaltung von Pipeline-Runs, überwacht diese kontinuierlich und leitet sie zur Ausführung an die Runtime weiter. Darüber hinaus besteht eine enge Verbindung zur Datenbank und zum File-Service, um Ergebnisse und Informationen der Pipeline-Runs zu persistieren.

Runtime

Die Runtime stellt das Kernelement dar, welches die Ausführung der Pipeline-Runs im System realisiert. Ihre zentrale Aufgabe besteht darin, die definierten Daten-Pipelines zu verarbeiten und Ergebnisse zurückzuliefern.

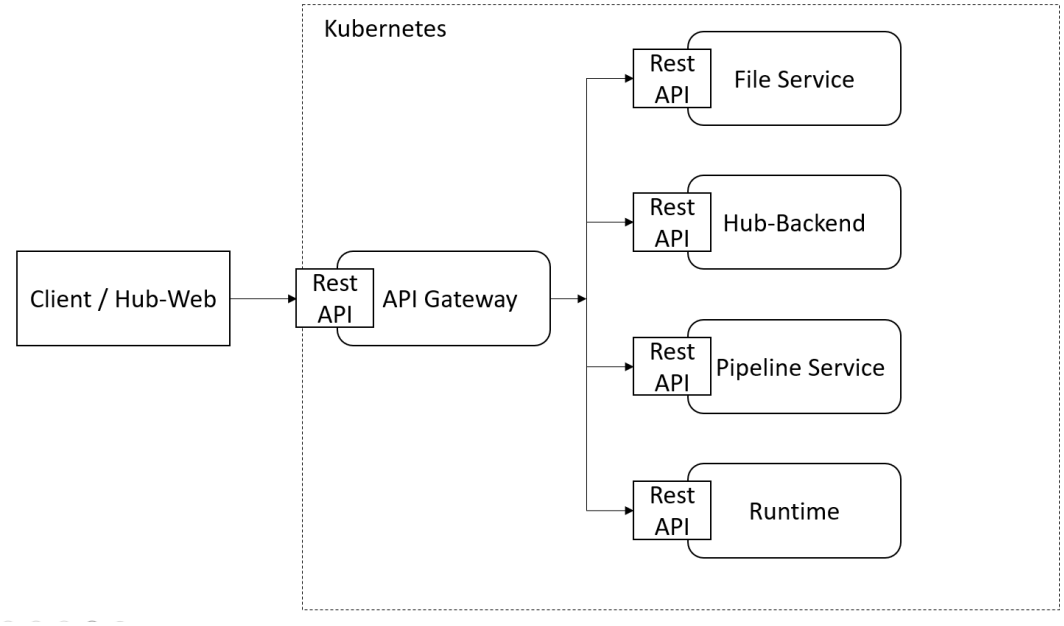

Abbildung 2.1: Übersicht Microservice Architektur JValue Hub

Zudem erfolgt die Verwaltung und Entwicklung der einzelnen Microservices mittels Mono-Repository Ansatz über GitHub[3]. Die Continuous Integration Pipeline der Anwendung wird mithilfe von GitHub Actions[4] orchestriert. Diese Pipeline baut und testet die Microservices und generiert anschließend eigenständige Docker[5] Container. Abschließend werden die Container in die GitHub Container Registry[6] geladen und veröffentlicht.

2.2 Microservices

Microservices stellen einen Ansatz zur Modularisierung von Software dar. Das Hauptziel besteht darin, ein großes, monolithisches System in kleine Bausteine zu unterteilen, um Software einfacher zu erstellen, zu verstehen und weiterzuentwickeln (Wolff, 2018).

[3]https://github.com/
[4]https://github.com/features/actions
[5]https://www.docker.com/
[6]https://github.com/features/packages

Diese Bausteine stellen einzelne eigenständige Services dar, welche über unterschiedliche Netzwerkmechanismen miteinander kommunizieren können. Hierbei kann mithilfe von unterschiedlichen Technologien und Konzepten die Kommunikation innerhalb der Microservices umgesetzt werden (Newman, 2021).

Einerseits besteht die Möglichkeit, dass eine Anwendung über eine Programmierschnittstelle beziehungsweise Application Programming Interface (API) Funktionen und Ressourcen für andere Dienste bereitstellt. In Bezug auf Microservices handelt es sich oft um REST-APIs, auch bezeichnet als RESTful-APIs. Dabei steht REST für Representational State Transfer und beschreibt ein Programmierparadigma zur Kommunikation in verteilten Systemen. Außerdem erfolgt der Austausch von Informationen im Kontext von REST über das HTTP-Protokoll, wobei REST auf dem Prinzip der Zustandslosigkeit basiert. Dies hat zur Folge, dass Interaktionen zwischen Sender und Empfänger isoliert voneinander ablaufen und keine Sitzungsinformationen zwischen den Anfragen gespeichert werden (Richardson & Ruby, 2008).

In Abbildung 2.2 ist eine beispielhafte Architektur der Kommunikation zwischen Microservices über REST abgebildet.

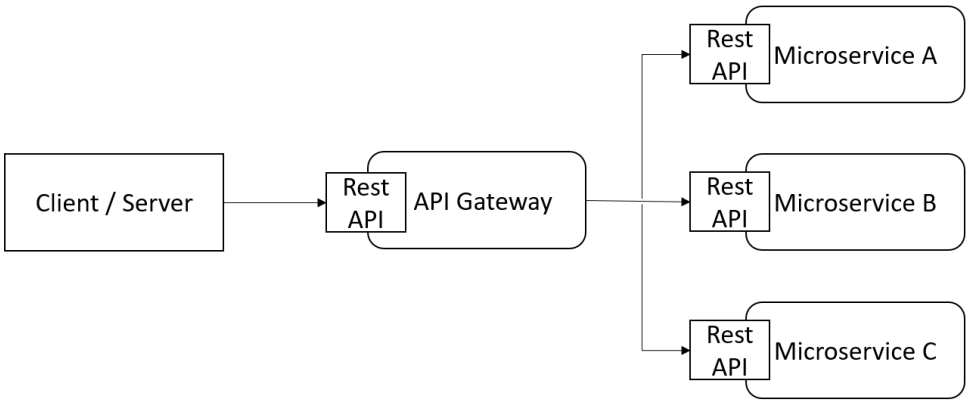

Abbildung 2.2: REST-API Microservice Architektur in Anlehnung an Hong et al. (2018)

Andererseits können Messaging-Systeme, welche meist das Publish-Subscribe-Prinzip umsetzen, zur Kommunikation innerhalb von Microservices eingesetzt werden. Durch die vollständige Entkopplung von Produzent und Konsument der Nachricht erfolgt die Interaktion zwischen den beiden Seiten durch einen Mechanismus zur Veröffentlichung und zur Konsumierung von Nachrichten. Das in der Praxis am häufigsten verwendete Protokoll zur Implementierung des Publish-Subscribe-Prinzips ist das Advanced Message Queuing Protocol (AMQP) (Xiong & Fu, 2011).

Grafik 2.3 veranschaulicht die Kommunikation innerhalb von Microservices mithilfe von Messaging-Systemen. Nachrichten werden vom Producer über eine Exchange an die entsprechende Warteschlange weitergeleitet. Anschließend können die Consumer die Nachricht abrufen und verarbeiten.

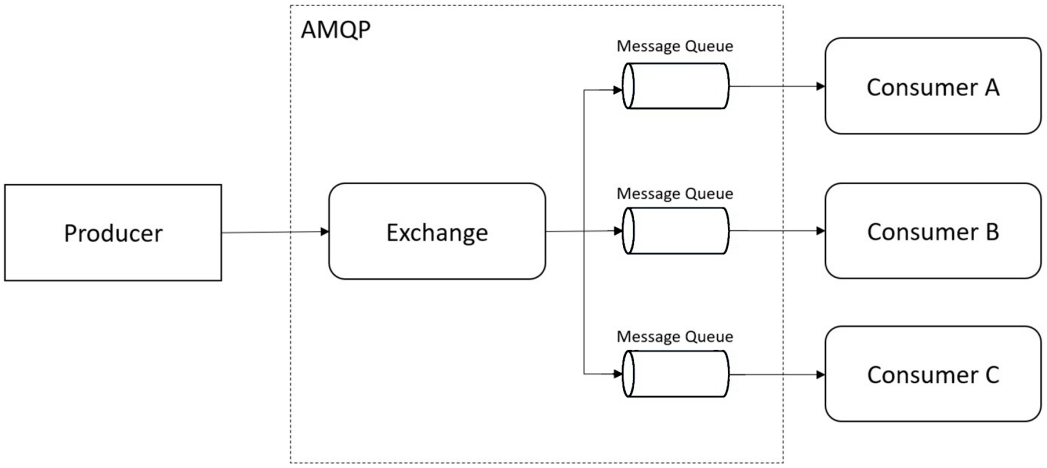

Abbildung 2.3: Messaging-Systeme Architektur in Anlehnung an Hong et al. (2018)

2.3 Performance-Test

Performance-Tests umfassen eine Vielzahl von Techniken, welche das Antwortverhalten, die Skalierbarkeit, Stabilität sowie Ressourcennutzung einer Anwendung untersuchen, aufzeigen und dabei helfen können, diese zu bewerten. Im Gegensatz zu funktionalem Testen, das sich auf die Korrektheit des Softwareverhaltens konzentriert, beschäftigt sich das Performance-Testing damit, wie effektiv eine Anwendung mit realen Nutzungsszenarien umgehen kann. Darüber hinaus können mithilfe von kontrollierten Lasten und Stressniveaus Engpässe identifiziert, potenzielle Systemgrenzen aufgedeckt und ein klares Verständnis darüber ermittelt werden, wie die Software unter Bedingungen agiert, die denen einer produktiven Umgebung ähneln (Pargaonkar, 2023).

Jiang und Hassan (2015) legen dar, dass viele unterschiedliche Definitionen und Interpretationen des Begriffes Performance-Test sowohl im Kontext der wissenschaftlichen Forschung als auch in der Industrie zu finden sind. Zusätzlich führen sie aus, dass der Begriff häufig als Synonym für zwei andere Begriffe verwendet wird: Lasttest und Stresstest.

Im Rahmen dieser Arbeit wird der Begriff „Performance-Test“ als systematischer Prozess verstanden, der darauf abzielt, Metriken und Aspekte im Zusammenhang mit der Performance zu identifizieren und zu evaluieren. Insbesondere werden im Rahmen dieser Untersuchung zwei Arten von Performance-Tests durchgeführt.

Dabei erfolgt die Messung und Auswertung der Antwortzeiten in einem Szenario, während in einem anderen Szenario die Verfügbarkeit, Zuverlässigkeit sowie der Durchsatz analysiert wird.

Gorton (2006) beschreibt die Response Time beziehungsweise Antwortzeit als Maßeinheit für die Latenz, die eine Anwendung für die Verarbeitung einer Geschäftstransaktion benötigt. Außerdem ist die Response Time in der Regel mit der Zeit verbunden, die eine Anwendung benötigt, um auf eine bestimmte Eingabe zu reagieren. Eine schnellere Response Time ermöglicht es den Benutzern, effektiver zu arbeiten und trägt somit zur Steigerung der Kundenzufriedenheit bei.

Throughput beziehungsweise Durchsatz ist ein Maß dafür, wie viel Arbeit eine Anwendung in einer bestimmten Zeiteinheit verrichtet. In der Praxis wird als Maßeinheit beispielsweise Transaktionen pro Sekunde, Nachrichten pro Sekunde oder Requests pro Sekunde verwendet (Gorton, 2006).

Im Rahmen von Performance-Tests werden die Begriffe Verfügbarkeit und Zuverlässigkeit oft in ähnlichen Kontexten genutzt, obgleich sie spezifische Bedeutungen aufweisen und daher nicht zwangsläufig als vollständige Synonyme betrachtet werden können. Es existieren klare Unterschiede zwischen den beiden Begriffen.

Verfügbarkeit wird als die Wahrscheinlichkeit gemessen, dass ein Software-Service oder System verfügbar ist, wenn es benötigt wird (Immonen & Niemelä, 2008). Zuverlässigkeit hingegen beschreibt die Wahrscheinlichkeit, dass ein Softwaresystem über einen bestimmten Zeitraum ohne Softwarefehler funktioniert (Shooman, 1984).

Während Verfügbarkeit und Zuverlässigkeit eng miteinander verbunden sind, kann ein System theoretisch hochverfügbar sein, indem es immer erreichbar ist, aber möglicherweise nicht zuverlässig, wenn es häufig Fehlfunktionen oder inkonsistente Ergebnisse liefert.

In der Performance-Analyse ist es wichtig, beide Aspekte zu berücksichtigen, um sicherzustellen, dass das System nicht nur gut erreichbar ist, sondern auch die erwartete Leistung mit einer hohen Zuverlässigkeit bereitstellt. Dies soll im Rahmen eines zweiten Testszenarios eingehend untersucht werden.

2.4 Locust

Locust[7] ist ein auf Python[8] basiertes Werkzeug zur Durchführung und Evaluierung von Performance-Tests. Das Werkzeug erzeugt mithilfe von vielen gleichzeitigen Benutzern Netzwerkverkehr und soll somit die Performance der Anwendung hinsichtlich unterschiedlicher Aspekte prüfen. Darüber hinaus bietet es die Möglichkeit, Metriken wie Response Time, Throughput und Fehler über eine Weboberfläche zu analysieren. Das Projekt ist Open-Source und wird unter der MIT-Lizenz vertrieben (Pradeep & Sharma, 2019).

Im Zuge dieser Arbeit wird Locust als Tool für die Durchführung der Performance-Tests eingesetzt und soll insbesondere Erkenntnisse darüber gewinnen, wie sich das zu untersuchende System unter der Last durch eine Vielzahl gleichzeitiger Benutzer verhält. Darüber hinaus ermöglicht Locust im Rahmen des zweiten Testszenarios die Bewertung der Effektivität der Anwendung bei der Bewältigung des Throughputs, der Verfügbarkeit und der Zuverlässigkeit.

Diese gezielte Anwendung von Locust trägt dazu bei, nicht nur die Performance im Hinblick auf Durchsatz zu analysieren, sondern auch die Fähigkeit der Anwendung, ihre Dienste mit hoher Verfügbarkeit und Zuverlässigkeit bereitzustellen. Diese Erweiterung des Testfokus trägt dazu bei, eine ganzheitliche Bewertung der Leistungsfähigkeit und Robustheit der Anwendung zu gewährleisten und liefert wertvolle Informationen für die umfassende Bewertung der Architekturen.

[7] https://locust.io/
[8] https://www.python.org/

2.5 wrk2

wrk2[9] stellt eine weitere Option zum Testen und Auswerten der Leistung einer Anwendung dar. Im Gegensatz zu Locust basiert es jedoch nicht auf der Anzahl der Benutzer, sondern generiert stattdessen einen konstanten Durchsatz, welche in Form von Requests pro Sekunde angegeben werden. Mithilfe eines Lua[10] Skripts können komplexe Sachverhalte der Tests formuliert werden, wodurch das Testverhalten exakter auf die Anforderungen des zu prüfenden Systems zugeschnitten werden kann.

Im Kontext dieser Untersuchung wird das Tool wrk2 eingesetzt, um die Antwortzeiten unter einer konstanten Last zu identifizieren und zu bewerten. Insbesondere wird wrk2 in einem Szenario der Performance-Tests verwendet, um die Antwortzeiten der Anwendung unter variablen Belastungen zu analysieren. Diese Verwendung des Tools ermöglicht eine präzise Beurteilung der Performance der Architekturen im Hinblick auf die Metrik Antwortzeit und trägt zur umfassenden Bewertung der Anwendungsperformance bei.

2.6 RabbitMQ

RabbitMQ[11] ist ein Open-Source-Messaging-System, das auf dem im Kapitel 2.2 eingeführten Protokoll zur Kommunikation zwischen Microservices, AMQP, basiert (Dossot, 2014).

Im Gegensatz zur Kommunikation mittels REST und HTTP bietet die Nutzung von RabbitMQ als Messaging-System verschiedene Vorteile. Zunächst ermöglicht es eine lockerere Kopplung der einzelnen Microservices. Wenn beispielsweise Service A Nachrichten in eine Queue schreibt und diese von Service B gelesen werden, benötigt Service A keine Kenntnisse über Service B. Beim Hinzufügen eines neuen Services C, der ebenfalls die Nachrichten in der Queue erhalten möchte, erfordert dies keinerlei Änderungen an Service A. Diese Flexibilität ist bei der Kommunikation über REST und HTTP nicht gegeben.

Darüber hinaus kann es vorkommen, dass synchrone REST Aufrufe die Dienste blockieren und somit die Gesamtperformance der Anwendung beeinträchtigt wird. RabbitMQ ermöglicht eine asynchrone Kommunikation der Microservices, bei der die einzelnen Dienste nicht auf die unmittelbare Antwort des anderen Dienstes warten müssen, was zu einer verbesserten Performance beitragen kann (Schabowsky, 2017).

[9] https://github.com/giltene/wrk2
[10] https://www.lua.org/
[11] https://rabbitmq.com/

Im weiteren Verlauf dieser Arbeit wird RabbitMQ als integraler Bestandteil einer Architektur verwendet und soll einen alternativen Ansatz der Kommunikation zwischen den Microservices repräsentieren.

2.7 Docker

Docker[12] ist eine Softwareplattform, die es ermöglicht, Anwendungen effizient zu erstellen, zu testen und bereitzustellen. Die Funktionsweise von Docker beruht darauf, Software in standardisierte Einheiten, sogenannte Container, zu verpacken. Diese Container enthalten sämtliche Komponenten, die zur Ausführung der Software benötigt werden, darunter Bibliotheken, Systemtools, Code und Laufzeitumgebungen. Durch die Verwendung von Docker können Anwendungen in jeder Umgebung schnell bereitgestellt und skaliert werden. Dies gewährleistet, dass der Code unabhängig von der Zielumgebung zuverlässig ausgeführt wird (aws.amazon.com, 2024).

So werden mithilfe von Docker die einzelnen Microservices der zu analysierenden Anwendung JValue containerisiert. Anschließend erfolgt die Veröffentlichung der Container in der GitHub Container Registry[13], wodurch ein Zugriff auf diese ermöglicht wird.

2.8 Kubernetes

Kubernetes[14] fungiert als Open-Source-System zur automatischen Bereitstellung, Skalierung sowie Management von containerbasierten Anwendungen (Burns et al., 2016).

Innerhalb des Versuchsaufbaus und der Durchführung der Performance-Tests wird Kubernetes als zentrales Orchestrierungswerkzeug eingesetzt, um die Bereitstellung und Verwaltung der Anwendungen sowie weiteren Werkzeugen wie Datenbanken und Messaging-Systemen zu automatisieren.

Mittels dem speziell für Kubernetes entwickeltem Paketmanager Helm[15] werden hierbei Deployment und Verwaltung der Anwendungen auf dem verwendeten Kubernetes Cluster erleichtert. Es ermöglicht die einfache Definition, Installation und Aktualisierung von Kubernetes Anwendungen und deren Konfigurationen durch die Verwendung von sogenannten Helm-Charts (Shah & Dubaria, 2019).

[12] https://www.docker.com/
[13] https://github.com/features/packages
[14] https://kubernetes.io/
[15] https://helm.sh/

2.9 Node.js

Node.js[16] ist eine single threaded, Open-Source und plattformunabhängige Laufzeitumgebung zum Erstellen von schnellen, skalierbaren serverseitigen Anwendungen. Basierend auf der V8 JavaScript Laufzeitumgebung, verwendet es eine asynchrone, ereignisgesteuerte und nicht blockierende Architektur. Dieser Ansatz ermöglicht eine effiziente und effektive Verarbeitung von Anforderungen (Node.js, 2024).

Node.js setzt auf die Architektur des sogenannten „Single Threaded Event Loop“, um simultan mehrere Anfragen nicht blockierend zu bewältigen. Ein begrenzter Threadpool wird von Node.js gepflegt, um Anfragen effizient zu bedienen, wobei die Verarbeitung durch den sogenannten „Event Loop“ erfolgt. Wenn eine Anfrage eingeht, wird sie zunächst in die Warteschlange „Event Queue“ gestellt. Der Event Loop, als zentraler Mechanismus, wartet kontinuierlich auf ankommende Anfragen. Sobald eine Anfrage eintrifft, entnimmt der Event Loop sie aus der Warteschlange und überprüft, ob sie blockierende I/O-Operationen oder ressourcenintensive CPU-Aufgaben erfordert. Bei blockierenden Operationen weist der Event Loop die Bearbeitung der Anfrage einem Thread aus dem internen Threadpool zu (Tilkov & Vinoski, 2010).

Der Event Loop überwacht blockierende Anfragen und stellt sicher, dass diese nach Abschluss der blockierenden Aufgaben wieder in die Warteschlange eingereiht werden. Diese spezielle Struktur ermöglicht es Node.js, effizient auf Anfragen zu reagieren, selbst wenn sie intensive oder blockierende Operationen beinhalten. Diese Mechanismen sind von grundlegender Bedeutung für die nicht blockierende Natur von Node.js und tragen dazu bei, die Leistung und Reaktionsfähigkeit des Systems zu gewährleisten (Chitra & Satapathy, 2017).

In folgender Abbildung 2.4 ist dieser Mechanismus vereinfacht dargestellt.

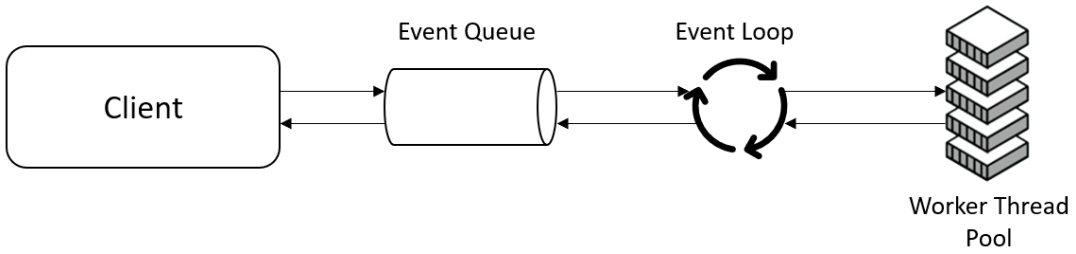

Abbildung 2.4: Event Loop Architektur in Node.js in Anlehnung an kinsta (2023)

[16] https://nodejs.org/en

Die Services der zu untersuchenden Anwendung JValue basieren auf Node.js. Die Entwicklung der Architekturen, welche im Verlauf dieser Arbeit miteinander verglichen und analysiert werden, erfolgt ausschließlich im Kontext von Node.js.

3 Anforderungen

Im folgenden Abschnitt werden die Anforderungen an die unterschiedlichen Architekturen definiert. Die Strukturierung orientiert sich hierbei an der FunktionsMASTeRs Schablone (SOPHISTen, 2022), wie sie in Abbildung 3.1 veranschaulicht ist. Eine präzise Festlegung dieser Anforderungen ist von entscheidender Bedeutung für das Verständnis, die Implementierung sowie die Evaluierung der Architekturen im Kontext dieser Masterarbeit.

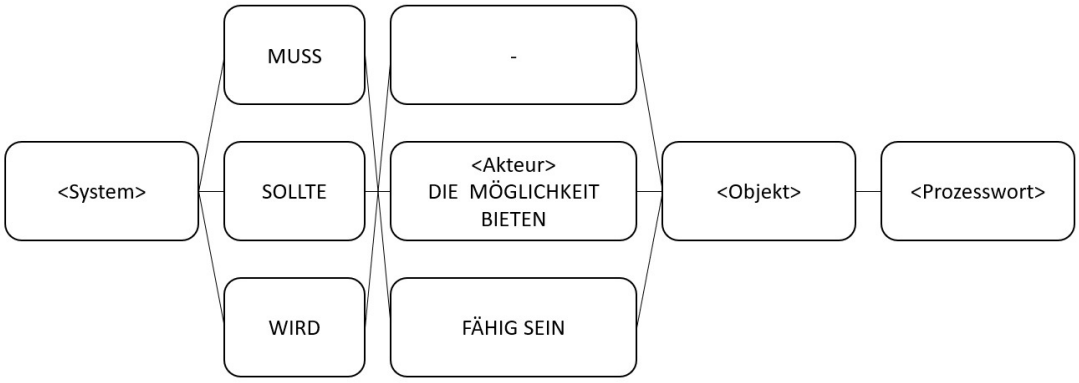

Abbildung 3.1: FunktionsMASTeR Schablone (SOPHISTen, 2022)

Im Rahmen dieser Untersuchung bezieht sich der Begriff <System> auf die jeweilige Architektur. Die Schlüsselwörter „MUSS“, „SOLLTE“ und „WIRD“ repräsentieren die Wichtigkeit der festgelegten Anforderungen. Eine „MUSS“ Anforderung legt dabei fest, dass die Architektur diese zwingend einhalten muss und keinerlei Spielraum für Abweichungen bietet. Hingegen beschreibt eine „SOLLTE“ Anforderung einen Wunsch an die Architektur, der nicht zwangsläufig erfüllt sein muss, jedoch angestrebt wird. Die Verwendung des Schlüsselworts „WIRD“ ermöglicht Aussagen über die Zukunft und erleichtert somit Entwicklungsanstrengungen in Hinblick auf zukünftige Anforderungen.

3.1 Technische Anforderungen

RQ-1 Die Architektur muss fähig sein, ausschließlich Open-Source-Technologien für die Umsetzung sowie die Durchführung der Performance-Tests zu verwenden.

RQ-2 Die Architektur muss fähig sein, den aktuellen Industriestandards zu entsprechen und breite Unterstützung in der Entwicklergemeinschaft zu genießen.

RQ-3 Die Architektur muss fähig sein, sich nahtlos in das vorhandene Ökosystem zu integrieren und eine reibungslose Interoperabilität zu ermöglichen.

RQ-4 Die Architektur muss fähig sein, durch die Auswahl sicherer Technologien, die Sicherheit der Anwendung zu gewährleisten, einschließlich der Berücksichtigung von Schwachstellen und regelmäßigen Aktualisierungen.

RQ-5 Die Architektur muss fähig sein, eine umfassende Dokumentation und klare Entwicklungsrichtlinien hinsichtlich der verwendeten Technologien bereitzustellen, um einen reibungslosen Entwicklungsprozess zu ermöglichen.

3.2 Performance Anforderungen

Antwortzeiten

PRQ-1 Die Architektur muss fähig sein, eine **geringe** Anzahl von Pipeline-Runs der Kategorie **„Klein“** mit einer akzeptablen Fehlertoleranz auszuführen.

PRQ-2 Die Architektur muss fähig sein, eine **geringe** Anzahl von Pipeline-Runs der Kategorie **„Klein“** mit einer akzeptablen Bearbeitungszeit auszuführen.

PRQ-3 Die Architektur muss fähig sein, eine **mittlere** Anzahl von Pipeline-Runs der Kategorie **„Mittel“** mit einer akzeptablen Fehlertoleranz auszuführen.

PRQ-4 Die Architektur muss fähig sein, eine **mittlere** Anzahl von Pipeline-Runs der Kategorie **„Mittel“** mit einer akzeptablen Bearbeitungszeit auszu-

führen.

PRQ-5 Die Architektur sollte fähig sein, eine **hohe** Anzahl von Pipeline-Runs der Kategorie **„Groß“** mit einer akzeptablen Fehlertoleranz auszuführen.

PRQ-6 Die Architektur sollte fähig sein, eine **hohe** Anzahl von Pipeline-Runs der Kategorie **„Groß“** mit einer akzeptablen Bearbeitungszeit auszuführen.

Robustheit

PRQ-7 Die Architektur muss fähig sein, Fehler zu tolerieren und nicht bei einzelnen Komponentenausfällen komplett auszufallen.

PRQ-8 Die Architektur sollte fähig sein, dass bei einem partiellen Systemausfall kritische Funktionen weiterhin verfügbar bleiben.

Verfügbarkeit

PRQ-9 Die Architektur muss fähig sein, jederzeit neue Aufträge anzunehmen.

PRQ-10 Die Architektur muss fähig sein, unter **geringer** Last durchgehend während des festgelegten Testzeitraums erreichbar zu sein.

PRQ-11 Die Architektur muss fähig sein, unter **mittlerer** Last durchgehend während des festgelegten Testzeitraums erreichbar zu sein.

PRQ-12 Die Architektur sollte fähig sein, unter **hoher** Last durchgehend während des festgelegten Testzeitraums erreichbar zu sein.

Zuverlässigkeit

PRQ-13 Die Architektur muss fähig sein, unter **geringer** Last während des festgelegten Testzeitraums eine akzeptable Zuverlässigkeitsrate aufzuweisen.

PRQ-14 Die Architektur muss fähig sein, unter **mittlerer** Last während des festgelegten Testzeitraums eine akzeptable Zuverlässigkeitsrate aufzuweisen.

PRQ-15 Die Architektur sollte fähig sein, unter **hoher** Last während des festgelegten Testzeitraums eine akzeptable Zuverlässigkeitsrate aufzuweisen.

Durchsatz

PRQ-16 Die Architektur muss fähig sein, unter **minimaler** Last während des festgelegten Testzeitraums eine konstante Rate von Requests pro Sekunde (RPS) aufzuweisen.

PRQ-17 Die Architektur muss fähig sein, unter **mittlerer** Last während des festgelegten Testzeitraums eine konstante Rate von RPS aufzuweisen. Die RPS sollte dabei gegen Ende des Testzeitraums abnehmen, um die Auswirkungen einer steigenden Last und Ressourcennutzung auf das System zu berücksichtigen.

PRQ-18 Die Architektur sollte fähig sein, unter **hoher** Last während des festgelegten Testzeitraums eine konstante Rate von RPS aufzuweisen. Die RPS sollte dabei gegen Ende des Testzeitraums abnehmen, um die Auswirkungen einer steigenden Last und Ressourcennutzung auf das System zu berücksichtigen.

3.3 Abgeleitete Szenarien

Im Kontext dieser Arbeit werden zwei verschiedene Ansätze zur Untersuchung der Performance verfolgt. Szenario Response Time fokussiert sich auf die Analyse der Antwortzeiten unter einer konstanten Anzahl an Anfragen pro Sekunde.

Szenario Integration zielt darauf ab, das Zusammenspiel der Komponenten Pipeline-Service und Runtime unter maximalen Bedingungen zu evaluieren. In diesem Zusammenhang werden insbesondere Aspekte der Verfügbarkeit, Zuverlässigkeit und der Durchsatz analysiert.

Darüber hinaus werden beide Szenarien dazu beitragen, Aussagen über die Robustheit der Anwendung zu treffen und die Architekturen hinsichtlich der Anforderungen bezüglich der Robustheit zu bewerten.

In den folgenden Abschnitten werden diese Szenarien detailliert vorgestellt und beschrieben.

3.3.1 Szenario Response Time

Die Festlegung des ersten Testszenarios, Response Time resultiert aus den definierten Anforderungen, die spezifische Bedingungen an die Antwortzeiten der Anwendung stellen. Diese Anforderungen sind von entscheidender Bedeutung, um die Leistungsfähigkeit der verschiedenen Architekturen zu evaluieren und sicherzustellen, dass sie den gestellten Anforderungen gerecht wird. Die Ableitung dieses Testszenarios aus den Antwortzeitanforderungen ermöglicht eine präzise Beurteilung und Vergleichbarkeit der Leistungsfähigkeit der verschiedenen Microservice Architekturen.

Das Szenario Response Time richtet sich dadurch gezielt auf die Analyse der unterschiedlichen Architekturen in Hinblick auf die Bearbeitungszeit der Pipeline-Runs aus. Besonderes Augenmerk wird dabei auf die Komponente Runtime gelegt, da in dieser die Verarbeitung der Pipeline-Runs erfolgt.

Hierbei fungiert das Tool wrk2 als zentrales Instrument zur Performance-Analyse der Anwendung. Im Gegensatz zu Locust, das auf eine adaptive Anpassung des Workload setzt, zielt wrk2 darauf ab, konstante Belastungsszenarien zu schaffen. Hierbei liegt der Fokus auf der Aufrechterhaltung einer stabilen Anzahl an Requests pro Sekunde über einen festgelegten Zeitraum.

Beim Starten des Tests in wrk2 können die Anzahl der Clients, die Anzahl der Threads, die Testdauer sowie die Anzahl der Requests pro Sekunde konfiguriert werden. Diese präzise Konfiguration der Testausführung ermöglicht eine gezielte Erzeugung eines konstanten Workloads, um die Antwortzeiten der Anwendung unter stabilen Bedingungen zu untersuchen.

Die Konfiguration der Parameter im Kontext von Testszenario Response Time werden in drei Kategorien, „Minimal“, „Medium“ sowie „Maximum“ unterteilt und sind in Tabelle 3.1 abgebildet.

	Minimum	Medium	Maximum
Anzahl Clients	1	5	10
Anzahl Threads	1	3	5
Requests pro Sekunde	3	10	20
Testdauer in Sekunden	60	120	120
Anzahl Request insgesamt	180	1200	2400

Tabelle 3.1: Konfiguration der Parameter für Testszenario Response Time

Zum anderen wird der Workload mithilfe von drei unterschiedlichen Datensätzen variiert, die aus öffentlich verfügbaren Datenquellen von Kaggle[1] stammen. Diese sind im entsprechenden Jayvee Modell hinterlegt. Die Größe der Datensätze beeinflusst die Ausführungszeit und Ressourcennutzung eines Pipeline-Runs und wird daher in Rahmen der Untersuchung angepasst. Infolgedessen werden für jede der drei Testkonfigurationen alle drei Datensätze als Workload für die Performance-Tests verwendet, um eine realistische Simulation des Nutzerverhaltens sicherzustellen. Der Aufbau der drei Datensätze ist in folgender Tabelle 3.2 dargestellt.

Datensatz	Kategorie	Spalten	Zeilen	Datenpunkte gesamt
Cars	Klein	12	33	396
Bookings	Mittel	6	526	3156
Reviews	Groß	10	4151	41510

Tabelle 3.2: Struktur der Datensätze

Die Architektur des Testszenarios Response Time ist in Abbildung 3.2 vereinfacht dargestellt. Hierbei wird das Tool wrk2 über einen Docker Container auf dem Test-System ausgeführt. Innerhalb dieses Containers werden Threads sowie dazugehörige Clients erzeugt, welche die definierten Anfragen an den Pipelineservice ausführen.

[1] https://www.kaggle.com/datasets

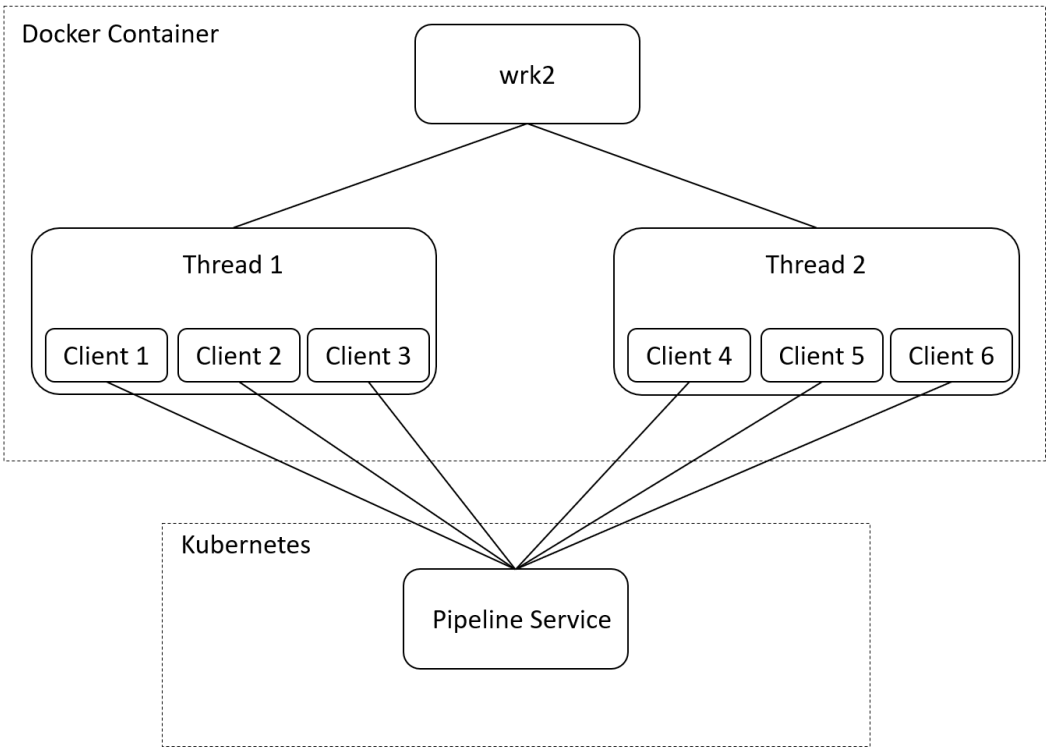

Abbildung 3.2: Aufbau Testszenario Response Time in Anlehnung an (Czeraszkiweicz, 2015)

3.3.2 Szenario Integration

Das zweite Szenario, Integration ergibt sich aus den Anforderungen bezüglich der Verfügbarkeit, Zuverlässigkeit und dem Durchsatz.

In diesem Testszenario wird Locust als zentrales Instrument verwendet, um die Performance der Microservices zu analysieren. Besonderes Augenmerk liegt im Kontext dieses Szenarios auf der detaillierten Analyse des Durchsatzes und der damit verbundenen Verfügbarkeit und Zuverlässigkeit der Anwendung. Der Testverlauf, beziehungsweise das Verhalten der von Locust erzeugten virtuellen Benutzer wird über ein sogenanntes Locust File definiert.

Darüber hinaus können beim Initiieren eines Tests in Locust die maximale Anzahl an gleichzeitigen Benutzern, die Anzahl der Arbeiterprozesse sowie die Spawn Rate konfiguriert werden. Die Spawn Rate gibt an, wie viele neue Benutzer pro Sekunde gestartet werden sollen. Bei einer Konfiguration von beispielsweise 100 für die maximale Anzahl an Benutzern und fünf für die Spawn Rate würde Locust jede Sekunde versuchen, fünf neue Benutzer zu erzeugen, bis die vordefinierte maximale Anzahl erreicht ist. Diese Konfigurationsoptionen bieten eine flexible Kontrolle über die Testumgebung und ermöglichen es, gezielt verschiedene Belas-

tungsszenarien zu simulieren.

Für die im Rahmen dieser Arbeit durchgeführten Performance-Tests wurden folgende, in Tabelle 3.3 dargestellt, Konfigurationsoptionen der Parameter in Testszenario Integration gewählt.

	Minimum	Medium	Maximum
Anzahl Benutzer	1	50	100
Anzahl Arbeiterprozesse	1	5	10
Spawn Rate	1	5	10
Testdauer in Sekunden	180	180	180

Tabelle 3.3: Konfiguration der Parameter für Testszenario Integration

Locust arbeitet nach dem Master-Slave-Prinzip, wobei der sogenannte Master-Node die Weboberfläche sowie Statistiken anzeigt und der Slave-Node dafür verantwortlich ist, die Lastgenerierung sowie die Ausführung der simulierten Benutzeraktivitäten zu steuern (Locust, 2024). Durch dieses Prinzip wird eine verteilte Lastgenerierung erreicht, welche die Simulation realistischer Benutzerverhaltensmuster ermöglicht und somit die Voraussetzung für eine präzise Performance Evaluation der Anwendung schafft.

Folgende Abbildung 3.3 zeigt den Aufbau der Architektur des Szenarios Integration. Die in der Parameterkonfiguration 3.3 festgelegte Anzahl der Arbeiterprozesse spiegeln die Slave-Nodes wider. Die Anzahl der Benutzer werden durch die virtuellen User repräsentiert.

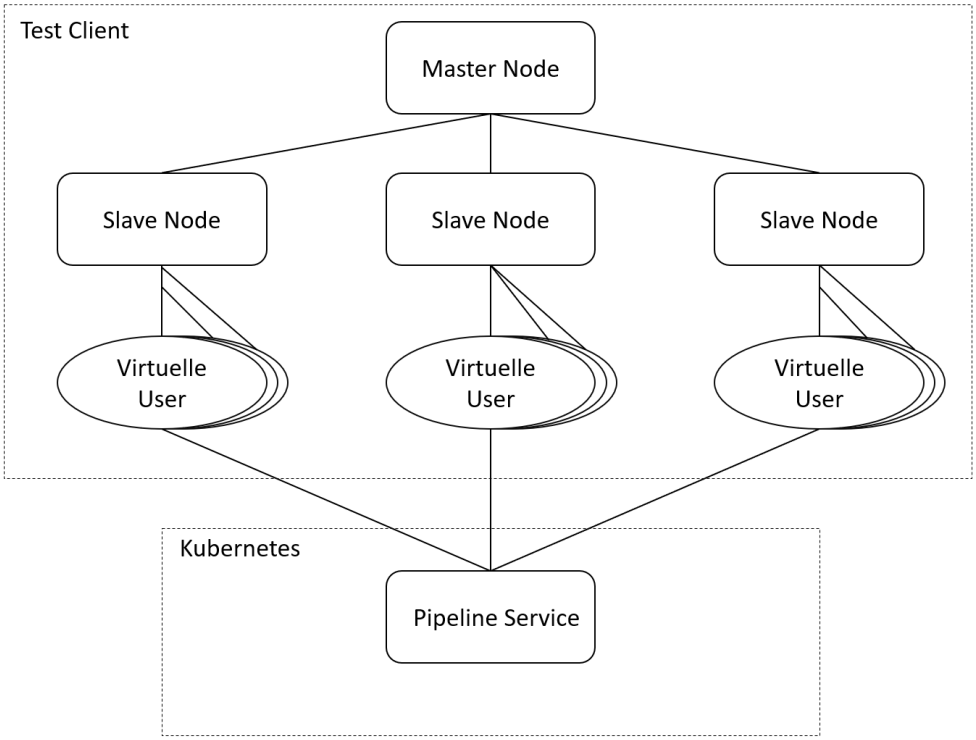

Abbildung 3.3: Aufbau Testszenario Integration in Anlehnung an (intersog, 2020)

Locust verfolgt einen adaptiven Ansatz zur Regulierung der Anzahl von RPS indem es diese dynamisch an das aktuelle Verhalten der Benutzer und Antwortzeiten des Services anpasst (Locust, 2024). Aufgrund dieser Konzeption von Locust ist eine konstante RPS an das System nicht möglich. Folglich zielt Testszenario Integration darauf ab, den Durchsatz, die Verfügbarkeit sowie die Zuverlässigkeit der Anwendung unter verschiedenem Workload eingehend zu evaluieren.

4 Architekturen

Im Verlauf dieser Untersuchung werden drei verschiedene Architekturen der Anwendung JValue analysiert und miteinander verglichen. Hierfür wird zum einen die Architektur in ihrem gegenwärtigen Ist-Zustand betrachtet. Zum anderen werden zwei alternative Architekturen konzipiert und entwickelt. Dieses Kapitel bietet eine Vorstellung dieser Architekturen, welche mithilfe der in 3.3 beschriebenen Performance-Test-Szenarien miteinander verglichen werden.

Zunächst erfolgt eine eingehende Vorstellung und Analyse des aktuellen Ist-Zustands der Anwendung JValue. Die präsentierten Erkenntnisse dienen dazu, relevante Implikationen für die weiterführende Untersuchung und Entwicklung der alternativen Architekturen aufzuzeigen.

Im Anschluss daran wird der konzeptionelle Aufbau zwei alternativer Architekturen vorgestellt. Hierbei verfolgt die Threadpool Architektur den Ansatz, die Kommunikation zwischen den Microservices über asynchrone REST Aufrufe durchzuführen. Darüber hinaus erfolgt die Verarbeitung der Pipeline-Runs im Vergleich zur Architektur des Ist-Zustands nicht mehr in eigenständigen Subprozessen, sondern wird vom internen Node.js Threadpool der libuv[1] abgewickelt. Konträr dazu zeichnet sich die Architektur Message Queue durch einen abweichenden Ansatz in der Kommunikation der Dienste aus, der auf dem Messaging-System RabbitMQ basiert.

[1] https://libuv.org

4.1 Ist-Architektur

Die Grundfunktionalität der Anwendung JValue besteht aus der Erzeugung sowie Ausführung von Pipeline-Runs. Im Ist-Zustand des JValue Dienstes erfolgt die Initiierung eines Pipeline-Runs durch ein HTTP-Request vom Anwender beziehungsweise der Frontend-Applikation Hub-Web zum Hub-Backend. Daraufhin wird die Anfrage zunächst an den Pipeline-Service und anschließend an die Runtime weitergeleitet. Für die Interpretation und Ausführung des Pipeline-Runs generiert die Runtime einen dedizierten Subprozess. Diese Vorgehensweise ist vereinfacht in Abbildung 4.1 dargestellt.

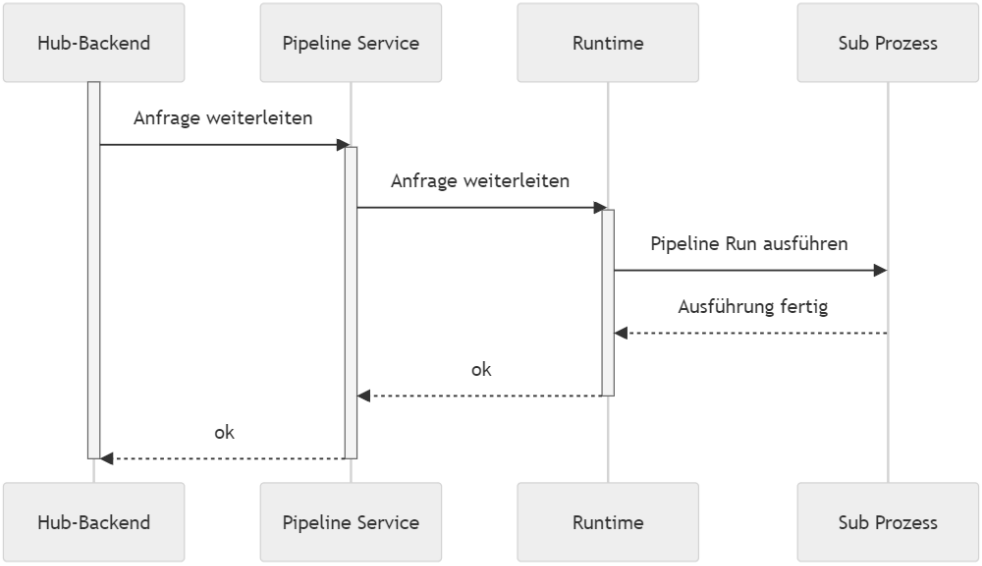

Abbildung 4.1: Sequenzdiagramm Ist-Architektur - vereinfachte Darstellung

Jedoch muss aufgrund dieses Ansatzes pro erzeugtem Pipeline-Run eine hohe Menge an Ressourcen aufgebracht werden. Beim Erzeugen von Subprozessen führen Initialisierung und Verwaltung jedes einzelnen Subprozesses zu einem gewissen Overhead. Dies umfasst den Speicherbedarf, die Prozesserstellung und -beendigung sowie die allgemeine Verwaltung der Subprozesse durch den Hauptprozess. Der hieraus resultierende Overhead kann eine erhebliche Belastung für das Gesamtsystem darstellen.

Durch diese Vorgehensweise kann ein Engpass bei der Verarbeitung der Pipeline-Runs entstehen, da es zum einen an einem Mechanismus zur Regulierung der Subprozesse fehlt. Das bedeutet, dass unabhängig von der Anzahl der eingehenden

Anfragen genau so viele Subprozesse erzeugt werden, was bei steigender Last dazu führen kann, dass die Anwendung abstürzt. Zum anderen existiert kein Mechanismus zur Wiederherstellung oder zum Neustarten der Pipeline-Runs. Falls also ein Replikat der Runtime während der Verarbeitung ausfällt, gehen alle Pipeline-Runs, die von diesem Replikat verarbeitet werden, unwiederbringlich verloren und werden nicht neu aufgenommen. Diese unzureichende Regulierung und fehlende Möglichkeit zur Wiederherstellung stellen kritische Schwachstellen dar, die die Stabilität und Zuverlässigkeit des Systems erheblich beeinträchtigen.

Aufgrund der identifizierten Probleme und Herausforderungen des Ist-Zustandes der Anwendung JValue, konzentrieren sich die entwickelten Architekturen sowie die durchgeführten Performance-Tests auf die Komponenten Pipeline-Service, Runtime sowie die Kommunikation zwischen den beiden Services.

4.2 Threadpool Architektur

Die Threadpool Architektur weicht stark vom aktuellen Zustand der Anwendung JValue ab. Das Konzept besteht darin, dass keine Subprozesse für die Verarbeitung eines einzelnen Pipeline-Run erzeugt werden. Hierfür werden die Funktionen des Jayvee Interpreters, welcher für die Interpretation und Verarbeitung der Pipeline-Runs verantwortlich ist, in eine eigenständige Bibliothek ausgelagert. Hierdurch wird ein direkter Aufruf dieser Verarbeitungs-Funktionalitäten des Interpreters ermöglicht.

Die Kommunikation der beiden Services erfolgt unverändert mittels REST und HTTP. Die Anfrage wird vom Pipeline-Service an die Runtime weitergeleitet, dort erfolgt nun die Ausführung des Runs mittels Aufruf der Interpreter Bibliothek. Nach Abschluss wird ein Callback an den Pipeline-Service durchgeführt, welcher das Ergebnis des Pipeline-Runs erhält. Diese Vorgehensweise ist stark vereinfacht im Sequenzdiagramm 4.2 dargestellt.

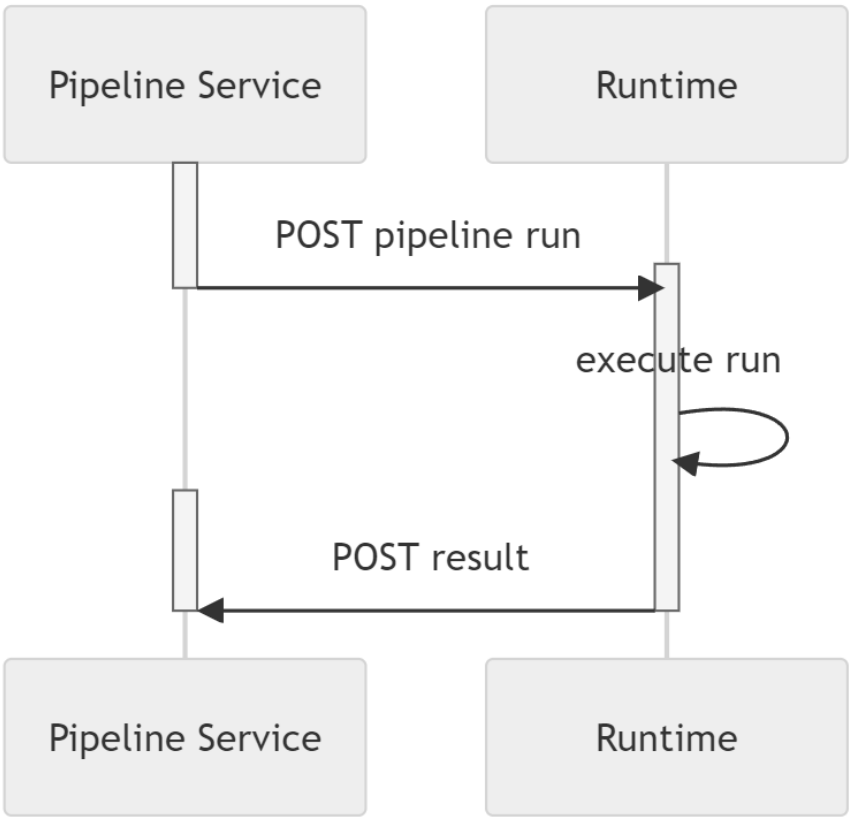

Abbildung 4.2: Sequenzdiagramm Threadpool Architektur - vereinfachte Darstellung

Diese Änderung der Architektur verfolgt primär das Ziel, den entstehenden Overhead bei der Erzeugung und Verwaltung von Subprozessen und die damit entstehenden Probleme zu eliminieren. Zusätzlich wird angestrebt, die internen Mechanismen von Node.js zur Verarbeitung von rechenintensiven Prozessen zu nutzen, um die Gesamtperformance der Anwendung zu verbessern. Es sei hervorzuheben, dass das manuelle Erzeugen von Subprozessen in Node.js nicht erforderlich ist. Node.js verfügt bereits über integrierte Mechanismen, insbesondere des in Kapitel 2 beschriebenen Threadpools, welcher auf der in der Programmiersprache C entwickelten Bibliothek libuv basiert. Diese Mechanismen ermöglichen es, CPU- und I/O intensive Aufgaben, wie die Ausführung von Pipeline-Runs, automatisch in den Threadpool auszulagern.

Dieser Ansatz dient dazu, die Anwendung nicht zu blockieren und eine Ausführung paralleler Anfragen zu gewährleisten. Diese effiziente Nutzung der gegebenen Mechanismen von Node.js soll dazu beitragen, die Performance der Anwendung zu verbessern.

4.3 Message Queue Architektur

Die Message Queue Architektur verfolgt einen alternativen Ansatz, indem die Kommunikation zwischen den beteiligten Services über das Messaging-System RabbitMQ realisiert wird. Hierbei werden zwei Queues beziehungsweise Warteschlangen, „hub-work“ und „hub-answer“ eingesetzt, um den Austausch von Nachrichten zwischen dem Pipeline-Service und der Runtime zu ermöglichen.

In der Queue „hub-work“, platziert der Pipeline-Service die Aufforderungen sowie alle erforderlichen Informationen zum Ausführen eines Pipeline-Runs. Diese Nachrichten werden nach dem First In First Out (FIFO) Prinzip von der Runtime gelesen und verarbeitet. Hierbei erfolgt die Verarbeitung, ähnlich wie in der Threadpool Architektur, direkt in der Runtime über die Funktionsaufrufe der Interpreter Bibliothek. Nach erfolgreicher Bearbeitung des Pipeline-Runs sendet die Runtime eine Nachricht in die „hub-answer“ Queue und übermittelt somit die Ergebnisse der Verarbeitung an den Pipeline-Service.

Eine vereinfachte Darstellung dieses Ansatzes ist in Abbildung 4.3 zu sehen.

Der Hauptfokus dieses Ansatzes besteht darin, die Kommunikation und damit die gesamte Performance der Anwendung zu verbessern. Zum einen kann im Gegensatz zur Kommunikation über REST und HTTP, wie in der Threadpool Architektur umgesetzt, ein Nachrichtenaustausch auch stattfinden, wenn einer der beiden Services gerade nicht erreichbar ist. Die Nachrichten werden in diesem Fall in der Warteschlange abgelegt und deren Verarbeitung kann wieder aufgenommen werden, sobald der ausgefallene Service zur Verfügung steht. Zum anderen sollen mithilfe dieser Änderung, Überlastungen am System vermieden und dadurch eine Reduzierung der Stillstandszeit des Systems erreicht werden.

Darüber hinaus soll mittels dieser Architektur eine robuste Verarbeitung der Pipeline-Runs unterstützt werden. Bei einem Absturz von Replikaten der Runtime während der Bearbeitung von Pipeline-Runs wird die Nachricht erneut in die Warteschlange zurückgeschrieben. Dadurch geht der Pipeline-Run nicht verloren und kann von einem anderen Replikat erneut aufgenommen und verarbeitet werden.

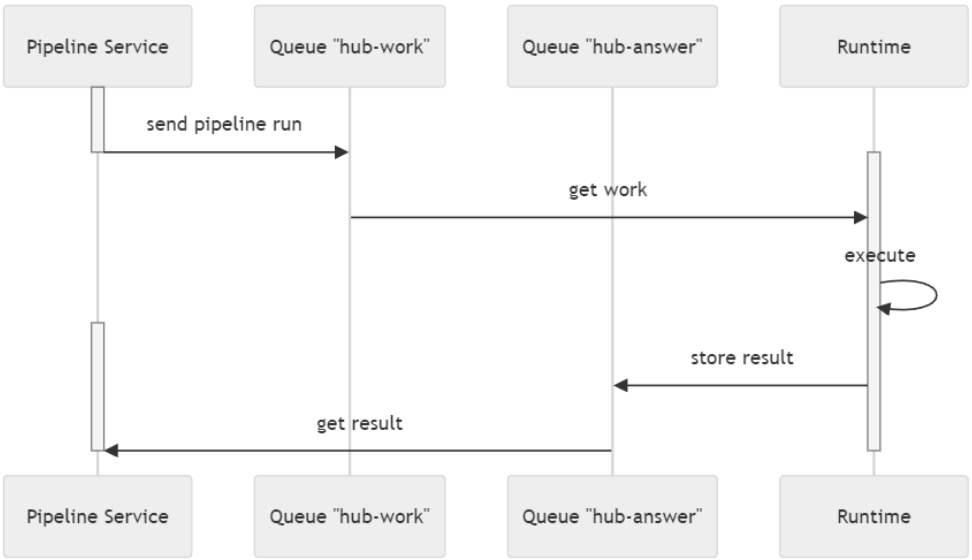

Abbildung 4.3: Sequenzdiagramm Message Queue Architektur - vereinfachte Darstellung

5 Implementierung

Das nachfolgende Kapitel bietet einen umfassenden Einblick in die Implementierung der beiden Architekturen Threadpool und Message Queue. Darüber hinaus wird der Versuchsaufbau der durchgeführten Performance-Tests vorgestellt.

Im Fokus stehen dabei die wesentlichen Komponenten der Infrastruktur und Umgebung, auf der die Anwendung JValue deployt ist. Hierbei werden insbesondere die Konfiguration der VMs sowie die verwendeten Kubernetes Operatoren beleuchtet. Zusätzlich wird eine Beschreibung der Konfiguration des Test-Systems vorgelegt, von dem aus die Performance-Tests durchgeführt werden.

Darüber hinaus wird der allgemeine Aufbau und die Parametrisierung der durchgeführten Performance-Tests ausführlich erläutert. Abgerundet wird das Kapitel durch eine eingehende Beschreibung der Implementierung der Performance-Tests, welche die Grundlage für die nachfolgenden Ergebnisse und Schlussfolgerungen bildet.

In den folgenden Abschnitten werden die präsentierten Code-Ausschnitte zur besseren Lesbarkeit leicht angepasst dargestellt.

5.1 Threadpool Architektur

Zum besseren Verständnis soll zunächst die bereits bestehende Logik zum asynchronen Empfangen von Nachrichten sowie Senden der Ergebnisse nach Ausführung eines Pipeline-Runs beschrieben werden. Dieser Ansatz bildet die Grundlage der Kommunikation zwischen dem Pipeline-Service und der Runtime im Ist-Zustand sowie der Architektur Threadpool.

Die im folgenden Code-Ausschnitt 5.1 präsentierte Methode „startSimpleRun“ stellt die Schnittstelle zum Empfangen von neuen Pipeline-Runs der Runtime dar. Beim Aufruf wird als Parameter eine Run Konfiguration übergeben, welche wichtige Informationen zur Verarbeitung des Pipeline-Runs, wie beispielsweise das auszuführende Jayvee Modell, Datenbankzugriffsinformationen sowie Details zum Zurücksenden der Ergebnisse beinhaltet.

Die Verarbeitung wird in Zeile 9-15 an die Klasse „RunsService“ weitergeleitet, indem die asynchrone Methode „runInterpreter“ aufgerufen wird. Der Aufruf von „runInterpreter“ gibt ein Promise zurück. Das bedeutet, dass der Dienst nach dem Aufruf der asynchronen Methode nicht auf das Ergebnis wartet und dieser somit nicht während der gesamten Bearbeitungszeit des Pipeline-Runs blockiert ist. Dieser Ansatz soll sicherstellen, dass auch während der Verarbeitung weitere Anfragen angenommen und gestartet werden können.

```
1  @Controller('runs')
2  export class RunsController {
3    logger = new Logger('RunsController.ts');
4    constructor(private readonly runsService: RunsService) {}
5    @Post()
6    @HttpCode(HttpStatus.OK)
7    startSimpleRun(@Body() config: RunConfigDto): void {
8      const callbackUrl = config.callbackUrl;
9      void this.runsService
10       .runInterpreter(
11         config.model,
12         config.databaseConnectionDetails,
13         `runresult_${config.runId.replace(/-/g, '')}`,
14         config.uploadedFileFolder,
15       )
```

Abbildung 5.1: Methode startSimpleRun

Der Block von Zeile 16 bis 20, welcher durch die Methode „then“ gekennzeichnet ist, wird ausgeführt, sobald das Promise erfüllt ist und ein Ergebnis zur Verfügung steht. In diesem Abschnitt wird das Resultat der Ausführung „executionResult“ in Zeile 17–18 verarbeitet und in ein Datenübertragungsobjekt (DTO) umgewandelt. Anschließend wird dieses Objekt an die definierte „callbackUrl“, welche auf den Pipeline-Service verweist, gesendet.

```
16       .then((executionResult) => {
17         const executionResultDto = RunsController.buildDto(
18           executionResult);
19         this.runsService.performCallback(callbackUrl,
20           executionResultDto);
```

Abbildung 5.2: Block then

Die vorliegende Methode „runInterpreter“ initiiert die Ausführung des Pipeline-Runs unter Verwendung des mitgelieferten Jayvee Modells und den Ausführungsoptionen „runOptions“. Diese werden zunächst in Zeile 32–48 vorbereitet, wobei insbesondere Datenbankinformationen zum Abspeichern der Ergebnisse in der Map „env“ gesetzt werden. Zusätzlich werden Debugging Optionen für eine bessere Analyse des Dienstes aktiviert.

```
25 async runInterpreter(
26     modelFile: string,
27     databaseConnectionDetails: DatabaseConnectionDetails,
28     table: string,
29     cwd: string,
30     uploadedFileFolder: string,
31   ): Promise<Result<RunSuccessData, RunErrorData>> {
32     const runOptions = {
33       env: new Map<string, string>(),
34       debug: true,
35       debugGranularity: 'exhaustive',
36       debugTarget: undefined,
37     };
38
39     runOptions.env.set('HUB_DB_HOST',
40     databaseConnectionDetails.database);
41     runOptions.env.set('HUB_DB_PORT',
42     databaseConnectionDetails.port.toString());
43     runOptions.env.set('HUB_DB_PASSWORD',
44     databaseConnectionDetails.password);
45     runOptions.env.set('HUB_DB_USERNAME',
46     databaseConnectionDetails.username);
47     runOptions.env.set('HUB_DB_DATABASE',
48     databaseConnectionDetails.username);
```

Abbildung 5.3: Methode runInterpreter

Daraufhin erfolgt die Ausführung eines Pipeline-Runs durch den Aufruf der asynchronen Funktion „interpretString“ in Zeile 57. Zuvor werden die Registrys „metaInformationenRegistry“, „blockExecutorRegistry“ sowie „constraintExecutorRegistry“ zurückgesetzt, um einen sauberen Start des Pipeline-Runs zu gewährleisten.

```
50     let result = null;
51     try {
52       // Clear registries
53       metaInformationRegistry.clear();
```

```
54          blockExecutorRegistry.clear();
55          constraintExecutorRegistry.clear();
56
57          result = await interpretString(modelFile, runOptions);
58        } catch (error) {
59          error_message = (error as Error).message;
60          this.logger.log('error in interpretstring: ', error);
61          result = ExitCode.FAILURE;
62        }
```

Abbildung 5.4: Ausführung eines Pipeline-Runs

Falls während der Ausführung ein Fehler auftritt, wird dieser abgefangen, eine entsprechende Fehlermeldung mit Informationen zur internen Ursache erstellt und anschließend an den Aufrufer zurückgegeben.

```
64        if (result === ExitCode.FAILURE) {
65          const failureData: RunErrorData = {
66            message: error_message,
67            cause: RunErrorCause.INTERNAL,
68          };
69          return Err(failureData);
70        }
```

Abbildung 5.5: Fehlerfall bei Pipeline Ausführung

Bei erfolgreichem Abschluss der Ausführung werden die Informationen zum Ablageort der Ergebnisse in der Datenbank an den Aufrufer Pipeline Service übergeben.

```
72        const successData: RunSuccessData = {
73          tableNames: [table],
74        };
75        return Ok(successData);
```

Abbildung 5.6: Erfolgsfall bei Pipeline Ausführung

5.2 Message Queue Architektur

Die Implementierung der Message Queue Architektur stellt einen alternativen Ansatz in der Kommunikation zwischen Pipeline-Service und Runtime dar. Dieses Konzept soll einen effizienten und sicheren Austausch von Nachrichten innerhalb der Microservices ermöglichen. Die Voraussetzung für die erfolgreiche Implementierung dieses Entwurfs ist die vorherige Installation und Konfiguration von RabbitMQ auf der zugrunde liegenden Infrastruktur auf Kubernetes. Eine Anleitung zur Installation ist in Anhang A zu finden.

Zunächst muss dazu in beiden Services eine Verbindung zum RabbitMQ Dienst hergestellt werden. Hierfür wird beim Starten der Services die Uniform Resource Locator (URL) der RabbitMQ Instanz aus dem Parameter „RABBIT URL“ aus der Helm Konfiguration ausgelesen und eine Verbindung über den Aufruf in Zeile 6 hergestellt. Darüber hinaus werden zwei Ereignis Handler „error“ und „connection“ hinzugefügt, welche bei einem Fehler im Verbindungsaufbau sowie einer erfolgreichen Verbindung mit dem RabbitMQ Dienst eine entsprechende Meldung an die Konsole ausgeben.

```
1  private rabbit: Connection;
2
3  constructor(private readonly runsService: RunsService,
4    private readonly configService: ConfigService) {
5    const url: string = configService.getOrThrow('RABBIT_URL');
6    this.rabbit = new Connection(
7      url,
8    );
9    this.rabbit.on('error', (err) => {
10     console.log('RabbitMQ connection error', err);
11   });
12   this.rabbit.on('connection', () => {
13     console.log('Connection successfully (re)established');
```

Abbildung 5.7: RunService Konstruktor

Außerdem werden sowohl im Pipeline-Service als auch in der Runtime ein Publisher für den RabbitMQ Dienst konfiguriert, um Nachrichten an eine entsprechende Warteschlange senden zu können. Dieser wird im folgenden Code-Ausschnitt durch den Aufruf der Methode „createPublisher“ erstellt. Dabei werden verschiedene Konfigurationsoptionen festgelegt. „confirm: true“ aktiviert die Bestätigung für gesendete Nachrichten, um sicherzustellen, dass diese erfolgreich an die Warteschlange übermittelt wurden. „maxAttempts: 2“ erlaubt bis zu zwei Wiederholungsversuche für den Fall, dass das Senden einer Nachricht fehlschlägt.

Diese Konfiguration der Parameter soll dazu beitragen, eine zuverlässige und widerstandsfähige Architektur zu gewährleisten. „exchanges“ definiert die Konfiguration der Warteschlange, „hub-work“ für den Pipeline-Service und „hub-answer“ für die Runtime.

```
17    private pub: Publisher;
18    // Publisher Pipeline Service
19    this.pub = this.rabbit.createPublisher({
20      confirm: true,
21      // Enable retries
22      maxAttempts: 2,
23      exchanges: [{ exchange: 'hub-work', type: 'direct' }],
24    });
25
26    // Publisher Runtime
27    this.pub = this.rabbit.createPublisher({
28      confirm: true,
29      // Enable retries
30      maxAttempts: 2,
31      exchanges: [{ exchange: 'hub-answer', type: 'direct' }],
32    });
```

Abbildung 5.8: Konfiguration der Publisher

Erreicht den Pipeline-Service eine Anfrage zur Bearbeitung eines Pipeline-Runs, leitet er diese über die Methode „send“ des Publishers an die Warteschlange „hub-work“ weiter.

```
50      await this.pub.send('hub-work', {
51        id: updateKey,
52        runid: runId,
53        config: runConfig,
54      });
```

Abbildung 5.9: Senden einer Nachricht an die Warteschlange

Im Konstruktor der Runtime wird ein Consumer zum Lesen der Nachrichten in der Warteschlange „hub-work“ definiert. Hier wird zusätzlich die Option „durable:true“ aktiviert, sodass Nachrichten auch bei einem Neustart von RabbitMQ persistiert sind und nicht verloren gehen. Diese Maßnahme soll zur Robustheit der Architektur beitragen, indem sie eine zuverlässige und beständige Nachrichtenverarbeitung ermöglicht.

```
58    const sub = this.rabbit.createConsumer(
59      {
60        queue: 'hub-work',
61        queueOptions: { durable: true },
62        exchanges: [{ exchange: 'hub-work', type: 'direct' }],
63        queueBindings: [{ exchange: 'hub-work'}],
64      },
```

Abbildung 5.10: Konfiguration des Consumers

Wenn die Runtime eine Nachricht konsumiert, wird folgender Codeblock 5.11 ausgeführt. Der Inhalt der Nachricht wird zunächst in Zeile 66-68 ausgelesen. Anschließend wird die Ausführung des Pipeline-Runs über den Funktionsaufruf „this.runService.interpretModel“ durchgeführt. Im Code-Ausschnitt von Zeile 75 bis 78 wird das Ergebnis der Ausführung gelesen und analog zu den beiden anderen Architekturen ein DTO erzeugt. Dieses wird über die Methode „this.pub.send“ als Nachricht in die Warteschlange „hub-answer“ geschrieben.

```
65      async (msg) => {
66        const id = msg.body.id;
67        const runid = msg.body.runid;
68        const config: RunConfigDto = msg.body.config;
69
70        void this.runsService.interpretModel(
71          config.model,
72          config.databaseConnectionDetails,
73          `runresult_${config.runId.replace(/-/g, '')}`,
74          config.uploadedFileFolder)
75        .then((executionResult) => {
76          // Benchmark: Stelle an der der Run fertig ist
77          const executionResultDto =
78          RunsController.buildDto(executionResult);
79
80        // Send Back Results
81        return this.pub.send('hub-answer',
82        {
83          id: id, runid: config.runId,
84          result: executionResultDto
85        })
86      });
87    );
```

Abbildung 5.11: Ausführung des Pipelines-Runs

Zum Abschluss erfolgt die Verarbeitung dieser Nachrichten durch den Consumer

des Pipeline-Services. In diesem Kontext erfolgt die Aktualisierung des Status und die Persistierung in einer Datenbank.

```
90    const sub = this.rabbit.createConsumer(
91      {
92        queue: 'hub-answer',
93        queueOptions: { durable: true },
94        exchanges: [{ exchange: 'hub-answer', type: 'direct' }],
95        queueBindings: [{ exchange: 'hub-answer'}],
96      },
97      async (msg) => {
98        let updateKey = msg.body.id;
99        let runid = msg.body.runid;
100       let result = msg.body.result;
101       let updateResult = null;
102
103       updateResult = await this.runsService.update({
104         runId: runid,
105         status: result.status,
106         data: result.payload,
107         updateKeyOptions: {
108           keyToCheck: updateKey,
109           newKey: null,
110         }
111       });
112     }
113   )
```

Abbildung 5.12: Konfiguration des Consumers im Pipeline-Service

5.3 Evaluator

5.3.1 Infrastruktur und Umgebung

Das verwendete Server-Cluster basiert auf der Open-Source Virtualisierungsplattform ProxMox 7.3-3[1] und besteht aus insgesamt sechs Servern, die physisch im gleichen Rack untergebracht sind. Die Verbindung zwischen den Servern erfolgt über ein virtuelles Netzwerk, das vom regionalen Rechenzentrum Erlangen[2] gehostet wird. Drei dieser Server wurden für die Ausführung von VMs konfiguriert.

[1] https://www.proxmox.com/de/
[2] https://www.rrze.fau.de

Diese VMs bilden die Grundlage der Testumgebung, unterliegen jedoch der Limitation, dass sie teilweise parallel zu anderen Namespaces und Stages der Anwendung, wie beispielsweise der produktiven Anwendung, betrieben werden. Dadurch kann es zu Einschränkungen in der Ressourcennutzung kommen, was bei der Gestaltung des Versuchsaufbaus berücksichtigt werden muss. Die Server und die darauf laufenden VMs sind, wie in Tabelle 5.1 dargestellt, benannt.

Server 1 ossammy	Server 2 ossleela	Server 3 osszoidberg
jvdoell-master01-amy	jvdoell-master03-leela	jvdoell-master02-zoidberg
jvdoell-worker05-amy	jvdoell-worker04-leely	jvdoell-worker01-zoidberg
jvdoell-worker06-amy	jvdoell-worker03-leela	jvdoell-worker02-zoidberg
jvdoell-worker07-amy		jvdoell-worker08-zoidberg

Tabelle 5.1: Serververteilung der VMs

Die VMs teilen sich eine standardisierte Konfiguration:

- Speicher: 16 GB
- Prozessoren: 4 (2 Sockets, 2 Kerne) mit Qemu[3] Maximalkonfiguration
- Speicherplatz: 53.9 GB

Die Netzwerkkonfiguration erfolgt über CloudInit[4], wobei die VMs durch die ProxMox Firewall geschützt und nur ausgewählte Ports geöffnet sind.
Das Kubernetes Cluster wurde auf den VMs installiert und verwendet die folgenden Versionen:

- k3s Version v1.22.6+k3s1
- etcd Version 3.5.9 auf den Master-Knoten
- ingress-nginx Version 4.6.1
- Locust 2.15.1
- PostgreSQL 14.3.0
- RabbitMQ 12.15.0
- RabbitMQ-cluster-operator 3.20.0

Die Durchführung der Performance-Tests erfolgt unabhängig vom Kubernetes Cluster über ein externes Test-System. Diese Vorgehensweise soll sicherstellen, dass sich Client und Server nicht im selben Netzwerk befinden und somit Netzwerkfaktoren, die im realen Betrieb auftreten, in der Betrachtung dieser Analyse berücksichtigt werden. Das verwendete Test-System weist folgende Spezifikationen auf:

[3]https://www.qemu.org
[4]https://cloud-init.io

- 6 CPU Kerne á 3.6GHz
- 16 GB Arbeitsspeicher

Diese Struktur legt den Grundstein für die Implementierung der Architekturen sowie der Durchführung und Evaluation der Performance-Tests.
Eine detaillierte Anleitung zum Installieren der Namespace, Operatoren und Services sowie der Performance-Tests ist im Anhang A zu finden.

5.3.2 Performance-Tests

Das folgende Sequenzdiagramm 5.13 illustriert die Funktionsweise des Performance-Tests, wobei lediglich das Verhalten eines einzelnen Test-Clients dargestellt wird. Es sei darauf hingewiesen, dass in der Durchführung der Performance-Tests mehrere Test-Clients gleichzeitig aktiv sind, hier jedoch aus Gründen der Vereinfachung nur das Verhalten eines einzelnen Clients abgebildet wird. Zusätzlich wird das Zusammenspiel des Pipeline-Service und Runtime dargestellt.

Darüber hinaus liefert das im Test bereitgestellte Jayvee Modell, welches detailliert im Anhang B zur Verfügung steht, keine Informationen bezüglich der Datenpersistenz. Dieser Ansatz soll sicherstellen, dass Datenbankzugriffe und damit verbundene Engpässe die Testergebnisse nicht beeinträchtigen. Folglich trägt dies zu einer präziseren Leistungsbewertung der Anwendung bei.

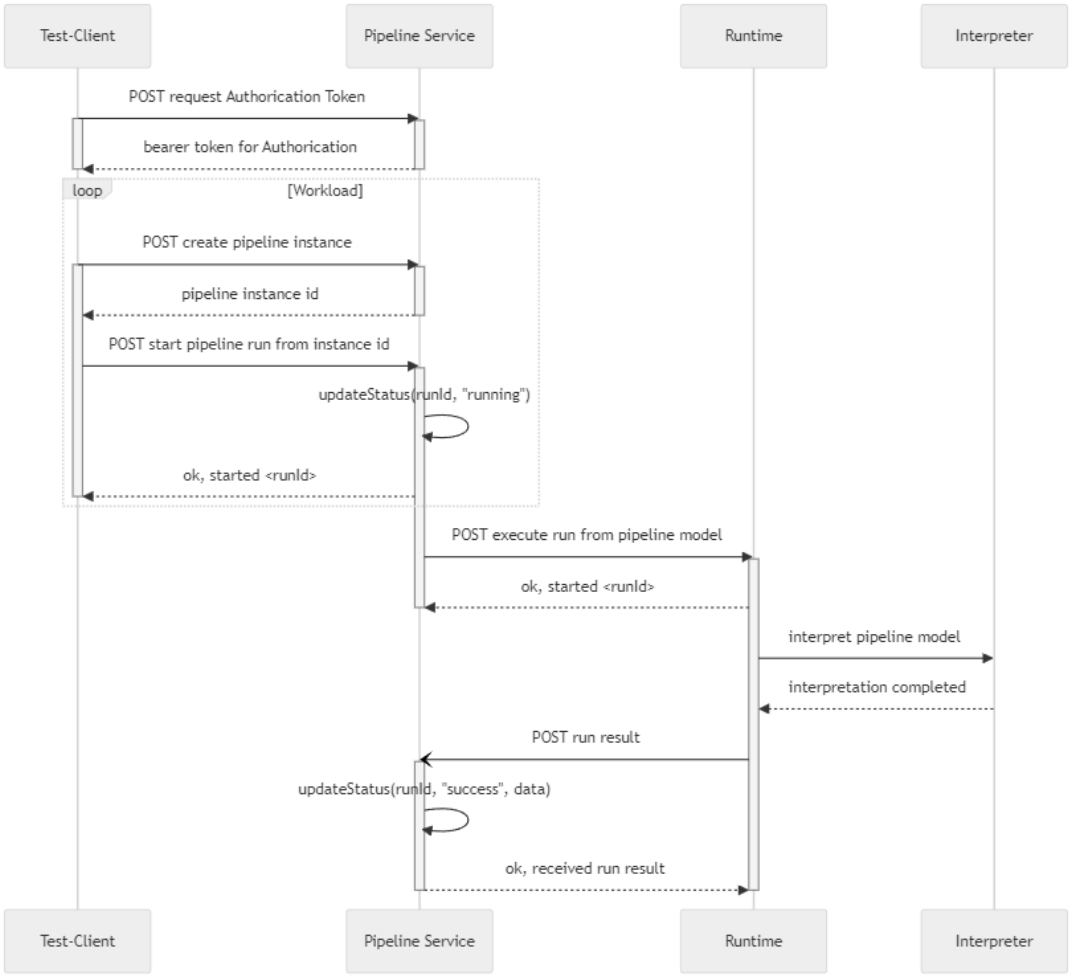

Abbildung 5.13: Sequenzdiagramm des Performance-Tests

Zunächst initiiert der Test-Client einen POST-Request, um ein Autorisierungstoken beim Pipeline-Service anzufordern. Dieses wird benötigt, um weitere Anfragen an den Service zu senden. Daraufhin startet die Generierung von Workload durch wiederholtes Anlegen von Pipeline-Instanzen in Verbindung mit der Initiierung eines Pipeline-Runs. Diesen Prozess setzt der Test-Client so lange fort, bis der gesamte Testzyklus vollständig abgeschlossen ist. Der Pipeline-Service leitet diese Anfragen an die Runtime weiter, in welcher die tatsächliche Ausführung der definierten Aktionen erfolgt. Dabei wird das mitgelieferte Jayvee Modell von einem Interpreter in der Komponente Runtime ausgeführt. Zusätzlich dazu übernimmt der Pipeline-Service die Koordination der laufenden Anfragen, indem der Dienst die Antworten und Ergebnisse der Runtime verwaltet sowie relevante Informationen wie Statusänderungen und Datenbankzugriffsinformationen in einer Datenbank persistiert.

Im Verlauf eines Performance-Tests erfolgt regelmäßig die Übermittlung von

Anfragen an den Pipeline-Service, um Pipeline-Runs auszuführen. Dies dient dazu, auf der Anwendung Last zu erzeugen und damit die Performance zu analysieren. Hierbei gilt es hervorzuheben, dass die Antwortzeit dieses Aufrufs nicht der Antwortzeit der Bearbeitung eines Pipeline-Runs entspricht. Die Antwort des Pipeline-Service symbolisiert lediglich, dass die Anfrage angenommen wurde. Zusätzlich wird mit der Antwort eine vom Pipeline Service generierte Universally Unique Identifier (UUID) des gestarteten Pipeline-Runs mitgeliefert sowie der Status des Pipeline-Runs auf „Running“ gesetzt. Eine beispielhafte Antwort des Pipeline-Service ist in folgender Abbildung 5.14 zu sehen.

```
{
    "runId": "2c1aaf72-3f6d-4f53-8f29-9a5728378780",
    "instanceId": "3e3a9ac0-606d-4b0b-af63-b3b810596603",
    "status": "Running",
    "createdAt": "2024-01-29T14:34:19.187Z"
}
```

Abbildung 5.14: Beispiel Antwort des Pipeline-Service nach Start eines Pipeline-Runs

Die Verarbeitung des Pipeline-Runs wird an die Runtime Komponente weitergeleitet. Der Status sowie die Ergebnisse eines abgeschlossenen Runs können daraufhin über einen separaten Endpoint am Pipeline-Service abgefragt werden.

Aufgrund dieser Vorgehensweise werden für die Berechnung der Antwortzeit im Rahmen von Szenario Response Time zusätzliche Schritte benötigt. Dies ist von essenzieller Bedeutung, um eine exakte Evaluierung der Performance, insbesondere der Antwortzeiten sicherzustellen. Während der kontinuierlichen Anfragen an den Pipeline-Service werden die Antworten zwischengespeichert. Nachdem der eigentliche Testzyklus abgeschlossen ist, werden die zurückgelieferten IDs gesammelt und anschließend in einer Textdatei auf dem Test-System abgelegt. Dieses Vorgehen ist stark vereinfacht in Abbildung 5.15 dargestellt.

In der Folge wird ein Python-Skript ausgeführt, um die Berechnung der Antwortzeiten anhand der generierten Datei durchzuführen. Eine detaillierte Beschreibung dieses Skripts erfolgt in Abschnitt 5.3.4. Zusätzlich ist das Skript im Anhang D beigefügt, um eine umfassende Nachvollziehbarkeit und Reproduzierbarkeit zu ermöglichen.

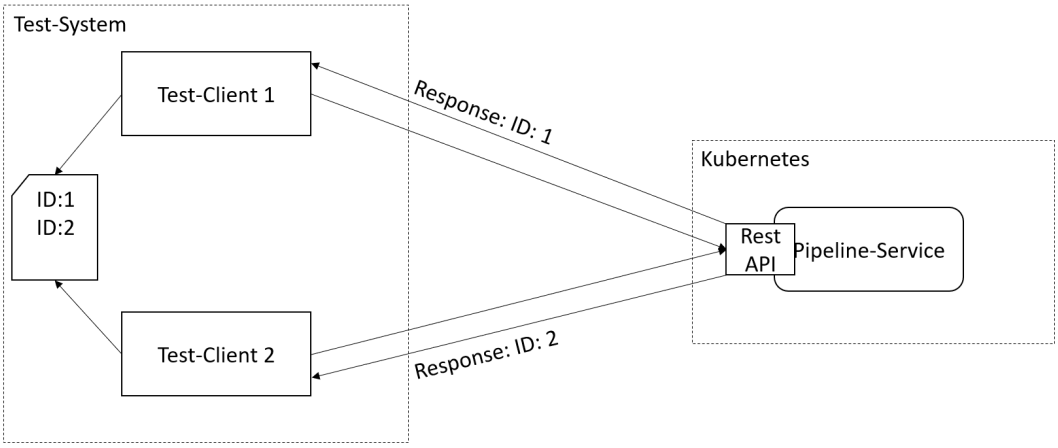

Abbildung 5.15: Abspeichern der Antworten auf dem Test-System

5.3.3 Parametrisierung

Im Verlauf der Untersuchung wird der im vorangegangen Abschnitt beschriebene Performance-Test mehrfach durchgeführt, wobei verschiedene Testkonstellationen und Parametervariationen angewendet werden. Hierbei werden die unterschiedlichen Parameterzusammenstellungen in drei Kategorien, „Minimal“, „Medium“ sowie „Maximum“ eingeteilt.

Diese Kategorisierung dient dazu, die Anwendung unter realistischen Belastungen zu analysieren. Dabei zielt die Testkonfiguration „Minimal“ darauf ab, die Grundvoraussetzungen der Anwendung unter minimaler Belastung zu evaluieren. Die Testkonfiguration „Medium“ spiegelt eine standardmäßige Nutzung der Anwendung wider, während die Testkonfiguration „Maximum“ die Anwendung unter maximaler Auslastung untersucht.

Diese Vorgehensweise ermöglicht eine umfassende Bewertung der Anwendungsleistung im Hinblick auf verschiedene Belastungsszenarien und trägt zur Realitätsnähe der durchgeführten Tests bei. Darüber hinaus wird der Performance-Test mit der Parameterkonstellation der jeweiligen Testkategorie dreimal durchgeführt und ein Durchschnittswert der drei Ergebnisse gebildet. Dies stellt sicher, dass die Performance der Anwendung eingehend evaluiert und nicht durch Ausreißer oder ähnliche Einflüsse verfälscht wird.

Im Rahmen der Testkonfigurationen erfolgt eine Modifikation der Ressourcen der Infrastruktur, wobei Anpassungen an der Anzahl der CPU-Kerne, der Instanzen pro Service sowie dem verfügbaren Arbeitsspeicher vorgenommen werden. Die Spezifikation der drei unterschiedlichen Setups ist in Tabelle 5.2 dargestellt.

Testkonfiguration	Minimum	Medium	Maximum
Replika Pipeline-Service	1	3	5
Anzahl CPU Pipeline-Service	1	1	1
Arbeitsspeicher Pipeline-Service	1 GB	2 GB	2 GB
Replika Runtime	1	3	7
Anzahl CPU Runtime	1	1	2
Arbeitsspeicher Runtime	2 GB	2 GB	8 GB

Tabelle 5.2: Ressourcenverteilung der Testumgebung

5.3.4 Umsetzungsdetails

Im folgenden Abschnitt wird die Implementierung der Performance-Tests beschrieben. An dieser Stelle sei erneut auf das Sequenzdiagramm 5.13 hingewiesen, welches die Funktionsweise des Performance-Tests detailliert darstellt.

Die vorgestellten Codeauszüge zeigen ausschließlich das Python-basierte Locust Skript, welches für die Durchführung von Testszenario Integration verwendet wird. Zur besseren Übersichtlichkeit wird das Lua-Skript, welches das Testszenario Response Time steuert, nicht dargestellt. Es ist anzumerken, dass die Logik der beiden Skripte nahezu identisch ist, lediglich die Programmiersprachen unterscheiden sich. Darüber hinaus wurden die Code-Ausschnitte leicht modifiziert, um eine bessere Lesbarkeit zu gewährleisten. Die originale Version der Skripte sowie eine Anleitung, wie diese auszuführen sind, befindet sich im Anhang C.

Zunächst wird über das Event „on start“ vor dem Start des Performance-Tests ein Autorisierungtoken beim Pipeline-Service mittels POST Aufruf in Zeile 11 angefordert. Die Antwort wird daraufhin in das Datenformat JavaScript Object Notation (JSON) konvertiert und das Token ausgelesen. Anschließend wird über die Zuweisung in Zeile 15 sichergestellt, dass das Token in allen zukünftigen Anfragen im Header der HTTP Requests mitgegeben wird.

```
10      def on_start(self):
11          response = self.client.post("/pipeline-service/auth/\
12          login?username=user&password=password")
13          json_var = response.json()
14          token = json_var['access_token']
15          self.client.headers = {'Authorization': 'Bearer ' + token}
```

Abbildung 5.16: Methode on start

Im Anschluss daran wird das Testverhalten des Clients in folgender Methode „create pipeline and start run“ spezifiziert. Die Annotation „@task“ teilt Locust

mit, dass diese „Aufgabe“ von einem Test-Client auszuführen ist. Beim Erstellen der Instanz in Zeile 19-25 wird das in Anhang B dargestellte Jayvee Modell sowie die Runtime „simple“ als Argumente an den Pipeline-Service übergeben. Daraufhin wird die Antwort des Dienstes verarbeitet und die entsprechende Instanz ID zum Initiieren eines Runs in der zweiten Anfrage in Zeile 29 und 30 übergeben. Abschließend wird in Zeile 33 die Kombination aus Instanz und Run ID in dem Array „run ids“ zwischengespeichert.

```
17      @task
18      def create_pipeline_and_start_run(self):
19          response = self.client.post("/pipeline-service/instances",
20          json=
21          {
22              "pipelineModel": self.pipelineModel,
23              "targetRuntime": "simple"
24          }
25          )
26          json_response_dict = response.json()
27          instance_id = json_response_dict['id']
28
29          response = self.client.post("/pipeline-service/instances/"
30          + instance_id + "/runs")
31          json_var = response.json()
32          run_id = json_var['runId']
33          self.run_ids.append(instance_id + "/runs/" + run_id)
```

Abbildung 5.17: Methode create pipeline and start run

Nach Abschluss eines Testlaufs wird das Event „on stop“ in Zeile 37 ausgelöst und alle abgespeicherten Kombinationen aus Instanz und Run ID werden, wie in Abbildung 5.15 illustriert, in einer Datei auf dem Test-System abgelegt.

```
37      def on_stop(self):
38          f = open(completeName, "x")
39          f.write(' '.join(self.run_ids))
40          f.close()
```

Abbildung 5.18: Methode on stop

Nachdem der gesamte Testlauf abgeschlossen ist, müssen die Antwortzeiten zur korrekten Evaluierung der Ergebnisse im Rahmen von Testszenario Response Time aufbereitet werden. Hierfür wird ein zusätzliches, in Python entwickeltes Skript bereitgestellt. Das vollständige Skript ist im Anhang D aufzufinden.

Zunächst wird die vom Performance-Test erzeugte Datei in den Zeilen 43 bis 45 geöffnet und die Kombinationen aus Instanz und Run ID in die Liste „requests as list“ geschrieben. Für jeden Eintrag in der Liste wird nun in Zeile 47-48 ein Request an den Pipeline-Service erzeugt. Der Status eines Runs kann beim Pipeline-Service über den Endpunkt
„instances/Instanz ID/runs/Run ID“ abgefragt werden. Die in der Datei abgelegten Kombinationen unterliegen bereits diesem Schema.

```
43      with open(os.path.join(save_path, user_file)) as f:
44          lines = f.readlines()
45          requests_as_list = lines
46          for request in requests_as_list:
47              response = requests.get(url_get_run + request,
48              headers={'Authorization' :'Bearer ' +token})
```

Abbildung 5.19: Aufbereitung der Ergebnisse

Daraufhin werden die Parameter „status“, „updatedAt“ und „createdAt“ aus der Antwort gelesen und zwischengespeichert. Zunächst wird mittels der Überprüfung in Zeile 52 sichergestellt, dass der Run bereits abgeschlossen und nicht mehr der Status „running“ gesetzt ist. Die beiden Parameter „createdAt“ und „updatedAt“ repräsentieren hierbei einen Zeitstempel, welche angeben, wann der entsprechende Run erzeugt und abgeschlossen wurde. Diese Zeitstempel werden jeweils in den Zeilen 58-61 in ein Zeitformat umgewandelt, um daraufhin in Zeile 62 die Differenz der beiden Zeitstempel in Sekunden zu berechnen. Die Differenz eines Runs wird abschließend in Zeile 63 in die Liste „time of requests“ abgespeichert.

```
49      json_resp = response.json()
50      status = json_resp['status']
51
52      if status == 'running':
53          pending_requests = True
54          continue
55
56          updatedAt = json_resp['updatedAt']
57          createdAt = json_resp['createdAt']
58          dt_finished = datetime.strptime(updatedAt,
59          '%Y-%m-%dT%H:%M:%S.%fZ')
60          dt_created = datetime.strptime(createdAt,
61          '%Y-%m-%dT%H:%M:%S.%fZ')
62          diff = (dt_finished-dt_created).total_seconds()
63          time_of_requests.append([diff, dt_created])
```

Abbildung 5.20: Berechnung der Antwortzeiten

Anschließend wird ein Verzeichnis zur Speicherung der Ergebnissen erstellt. Die Funktion „os.makedirs“ wird in Zeile 72 verwendet, um diese Verzeichnisstruktur anzulegen. Daraufhin werden die Ergebnisse in ein DataFrame-Objekt mit der Pandas-Bibliothek organisiert. Dieses DataFrame, das als „df“ bezeichnet wird, ist mit Spalten für „Antwortzeit in Sekunden“ und „Startzeit“ strukturiert. Danach wird das DataFrame als CSV-Datei innerhalb des zuvor erstellten Verzeichnisses gespeichert. Die Methode „to csv“ des DataFrame-Objekts ermöglicht diese Operation. Die Parameter „sep “ und „index“ stellen sicher, dass die Daten entsprechend formatiert sind, um sie später einfach weiterverarbeiten zu können.

```
70 # Create Dir for Results
71 results_path = os.path.join(savepath, str(uuid.uuid4()))
72 os.makedirs(results_path)
73
74 #Safe Raw Data to CSV
75 df = pd.DataFrame(time_of_requests,
76 columns=['Response Time in Seconds', 'Start Time'])
```

Abbildung 5.21: Abspeichern der Antwortzeiten

6 Evaluation

Im folgenden Kapitel werden die drei Architekturen der Anwendung JValue mithilfe der beiden in Kapitel 3.3 vorgestellten Szenarien analysiert. Für jedes dieser Szenarien dienen die drei in Kapitel 5.3.3 präsentierten Testkonfigurationen als Grundlage der Performance-Tests.

Darüber hinaus wurden im Rahmen von Szenario Response Time die durchgeführten Anfragen in zeitliche Blöcke von fünf Sekunden unterteilt. Anschließend wurden die Anfragen, die innerhalb jedes dieser Zeitfenster erfolgt, zu sogenannten Bins zusammengefasst. Für jeden dieser Bins wurde ein Durchschnittswert der Antwortzeiten berechnet. Ferner wurde der erste und letzte Bin der Untersuchung verworfen, um zu garantieren, dass die Start- und Abbauzeit der Testumgebung nicht in die Ergebnisse der Performance-Tests einfließen.
Diese Maßnahmen wurden ergriffen, um die Genauigkeit der Resultate zu gewährleisten und sicherzustellen, dass die Ergebnisse und deren Schlussfolgerungen ausschließlich auf den relevanten Daten basieren.

Zur verbesserten Übersicht und Verständlichkeit der nachfolgenden Ergebnisse werden in den folgenden Tabellen die Parameterkonfigurationen zusammengefasst abgebildet. In Tabelle 6.1 werden die allgemeinen Parameter der Testkonfigurationen dargestellt. Tabellen 6.2 und 6.3 beschreiben die spezifischen Parameter der beiden Testszenarien.

	Minimum	Medium	Maximum
Replika Pipeline-Service	1	3	5
Anzahl CPU Pipeline-Service	1	1	1
Arbeitsspeicher Pipeline-Service	1 GB	2 GB	2 GB
Replika Runtime	1	3	7
Anzahl CPU Runtime	1	1	2
Arbeitsspeicher Runtime	2 GB	2 GB	8 GB

Tabelle 6.1: Allgemeine Konfiguration der Parameter

	Minimum	Medium	Maximum
Anzahl Clients	1	5	10
Anzahl Threads	1	3	5
Requests pro Sekunde	3	10	20
Testdauer in Sekunden	60	120	120
Anfragen pro Testdurchlauf	180	1200	2400

Tabelle 6.2: Spezifische Konfiguration der Parameter für Testszenario Response Time

	Minimum	Medium	Maximum
Anzahl Benutzer	1	50	100
Anzahl Arbeiterprozesse	1	5	10
Spawn Rate	1	5	10
Testdauer in Sekunden	180	180	180

Tabelle 6.3: Spezifische Konfiguration der Parameter für Testszenario Integration

Die Darstellung der Ergebnisse für alle drei betrachteten Architekturen folgt einem einheitlichen Schema. Zu Beginn wird eine kurze Übersicht über die gewonnenen Erkenntnisse geboten, welche im späteren Kapitel 6.4 ausführlicher miteinander verglichen werden.

Im Anschluss wird eine konsolidierte Darstellung der Resultate präsentiert. Hierbei werden die aggregierten Ergebnisse der durchgeführten Performance-Tests im Testszenario Response Time für jede untersuchte Architektur in drei Tabellen dargestellt. Die Gruppierung erfolgt dabei nach den verschiedenen Datensatzkategorien „Klein“ (Cars), „Mittel“ (Bookings) und „Groß“ (Reviews).

Im weiteren Verlauf wird eine visuelle Darstellung der Testergebnisse präsentiert, in der die Antwortzeiten der Anwendung im zeitlichen Verlauf der Performance-Tests ersichtlich werden. Dieses strukturierte Vorgehen ermöglicht eine umfassende und systematische Analyse der Performance der betrachteten Architekturen unter verschiedenen Testbedingungen.

In den kommenden Kapiteln wird ausschließlich auf die Ergebnisse des Szenarios Response Time eingegangen. Die Resultate sowie Erkenntnisse aus dem Szenario Integration werden hingegen im späteren Kapitel 6.4 ausführlich dargestellt.

Diese strukturierte Herangehensweise ermöglicht eine klare Fokussierung auf die spezifischen Aspekte des Szenarios Response Time, während im Vergleichskapitel eine umfassende Analyse und Gegenüberstellung der gewonnenen Erkenntnisse erfolgt.

6.1 Ist-Architektur

Die Darstellung der Ergebnisse der Ist-Architektur Time beschränkt sich auf die Testkonfiguration „Minimum“ in Verbindung mit dem Datensatz der Kategorie „Klein“ (Cars). Dies ergibt sich daraus, dass sämtliche andere Testkonstellationen zu einer 100-prozentigen Fehlerrate der Anfragen geführt haben und daher nicht dargestellt werden können.

Diese signifikanten Fehler lassen sich auf zwei Hauptursachen zurückführen. Zum einen wird für jeden Pipeline-Run ein eigener Subprozess erzeugt. Dabei existieren keine Mechanismen oder Regelungen hinsichtlich der Anzahl gleichzeitig laufender Subprozesse. In der Konsequenz entstehen im Verlauf der meisten Performance-Tests zunehmend Subprozesse, was letztendlich zu einem Ressourcenmangel und zu einem Absturz der Runtime Instanzen führt.

Zum anderen fehlt es an einem Mechanismus, um Pipeline-Runs, die während der Verarbeitung abstürzen, wiederherzustellen. Das bedeutet, dass der Pipeline-Run dauerhaft den Status „Running“ beibehält und nicht ausgeführt wird. Diese Umstände sind problematisch, da einerseits keine Verarbeitung möglich ist und andererseits der Benutzer nicht über den erfolglosen Verarbeitungsversuch informiert wird.

Infolgedessen wurden bei 8 von 9 Performance-Tests im Szenario Response Time 100 Prozent der gestarteten Pipeline-Runs nicht verarbeitet und gingen somit verloren. Diese Erkenntnisse ermöglichen einen kritischen Blick auf die Stabilität und Fehlerbehandlungsmöglichkeiten der Ist-Architektur in Bezug auf die betrachteten Testkonfigurationen und Datensätze.

	Minimum	Medium	Maximum
Anzahl Requests pro Durchlauf	180	1200	2400
Ø Anzahl fehlerhafter Requests	12	1200	2400
Ø Anzahl erfolgreicher Requests	168	0	0
Ø Fehlerrate in Prozent	7,14	100	100
Ø Antwortzeit in Sekunden	474,44	-	-

Tabelle 6.4: Aggregierte Ergebnisse Ist-Architektur Datensatz Kategorie „Klein“ (Cars)

	Minimum	Medium	Maximum
Anzahl Requests pro Durchlauf	180	1200	2400
Ø Anzahl fehlerhafter Requests	180	1200	2400
Ø Anzahl erfolgreicher Requests	0	0	0
Ø Fehlerrate in Prozent	100	100	100
Ø Antwortzeit in Sekunden	-	-	-

Tabelle 6.5: Aggregierte Ergebnisse Ist-Architektur Datensatz Kategorie „Mittel“ (Bookings)

	Minimum	Medium	Maximum
Anzahl Requests pro Durchlauf	180	1200	2400
Ø Anzahl fehlerhafter Requests	180	1200	2400
Ø Anzahl erfolgreicher Requests	0	0	0
Ø Fehlerrate in Prozent	100	100	100
Ø Antwortzeit in Sekunden	-	-	-

Tabelle 6.6: Aggregierte Ergebnisse Ist-Architektur Datensatz Kategorie „Groß“ (Reviews)

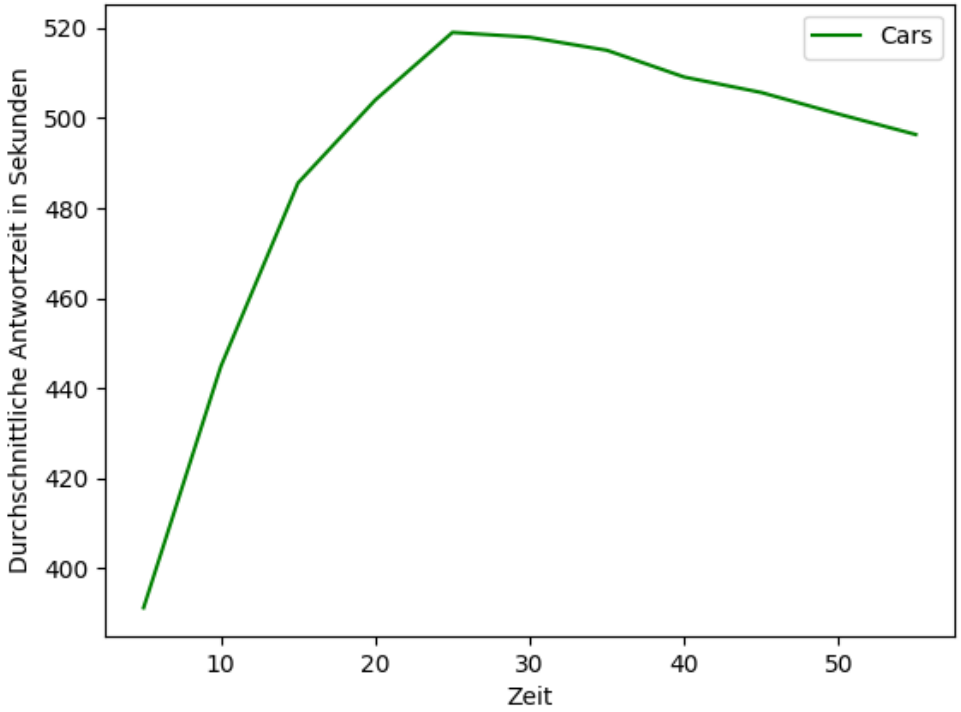

Abbildung 6.1: Ist-Architektur Testkonfiguration Minimum

6.2 Threadpool Architektur

Die Ergebnisse der Performance-Tests offenbaren analog zur Ist-Architektur vergleichbare Herausforderungen. Die Architektur ermöglicht zwar die erfolgreiche Durchführung von 8 von 9 Performance-Tests, jedoch weist auch diese Defizite auf, insbesondere hinsichtlich der Regulierung gestarteter Pipeline-Runs sowie der Wiederaufnahme fehlgeschlagener Pipeline-Runs.

Es ist festzustellen, dass das zugrundeliegende Problem nicht mehr in der unkontrollierten Erzeugung von Subprozessen besteht, sondern lediglich in die internen Threadpool Mechanismus von Node.js verlagert wird. Durch diese Struktur können die Pipelines-Runs zwar effizienter verarbeitet werden, was zu einer verbesserten Nutzung der Ressourcen führt. Trotzdem fehlt es weiterhin an einem Mechanismus, der die Anzahl der gestarteten Pipeline-Runs reguliert und die Wiederaufnahme von fehlgeschlagenen Pipeline-Runs ermöglicht.

Auffällig ist zudem der deutliche Einfluss der Datensatzgröße auf die Antwortzeiten, insbesondere in der Kategorie „Groß“ (Reviews). Hier ist ein signifikanter

Anstieg der Antwortzeiten zu verzeichnen, was auf die komplexere Verarbeitung größerer Datensätze zurückzuführen ist.

	Minimum	Medium	Maximum
Anzahl Requests pro Durchlauf	180	1200	2400
Ø Anzahl fehlerhafter Requests	0	0	0
Ø Anzahl erfolgreicher Requests	180	1200	2400
Ø Fehlerrate in Prozent	0	0	0
Ø Antwortzeit in Sekunden	0,32	1,13	0,32

Tabelle 6.7: Aggregierte Ergebnisse Threadpool Architektur Datensatz Kategorie „Klein“ (Cars)

	Minimum	Medium	Maximum
Anzahl Requests pro Durchlauf	180	1200	2400
Ø Anzahl fehlerhafter Requests	0	3	6
Ø Anzahl erfolgreicher Requests	180	1197	2394
Ø Fehlerrate in Prozent	0	0,25	0,25
Ø Antwortzeit in Sekunden	2,32	51,66	5,69

Tabelle 6.8: Aggregierte Ergebnisse Threadpool Architektur Datensatz Kategorie „Mittel“ (Bookings)

	Minimum	Medium	Maximum
Anzahl Requests pro Durchlauf	180	1200	2400
Ø Anzahl fehlerhafter Requests	8	1200	6
Ø Anzahl erfolgreicher Requests	172	0	2394
Ø Fehlerrate in Prozent	4,6	100	0,25
Ø Antwortzeit in Sekunden	308,30	-	447,35

Tabelle 6.9: Aggregierte Ergebnisse Threadpool Architektur Datensatz Kategorie „Groß“ (Reviews)

Insbesondere in der Architektur Threadpool fällt auf, dass die Antwortzeiten, aufgrund der parallelen Ausführung unabhängig vom Datensatz, relativ gleichmäßig sind. Dies verdeutlicht die Effizienz dieses Architekturansatzes, da die zur Verfügung stehenden Ressourcen besser genutzt werden und unter bestimmten Voraussetzungen eine konstante Performance gewährleistet ist.

Es ist wichtig zu betonen, dass der Datensatz der Kategorie „Groß“ (Reviews) in der Medium-Testkonfiguration zu einer Fehlerquote von 100% geführt hat und daher nicht in der Grafik enthalten ist. Dieser Umstand verdeutlicht die Herausforderungen und Schwächen der betrachteten Architekturen.

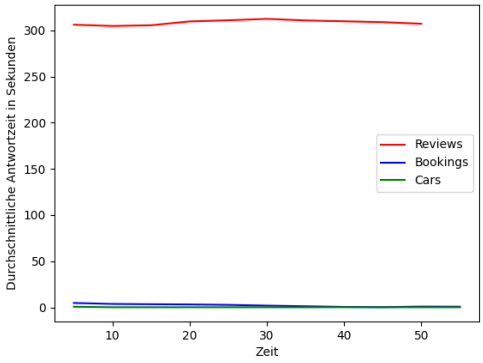

Abbildung 6.2: Threadpool Architektur Testkonfiguration Minimum

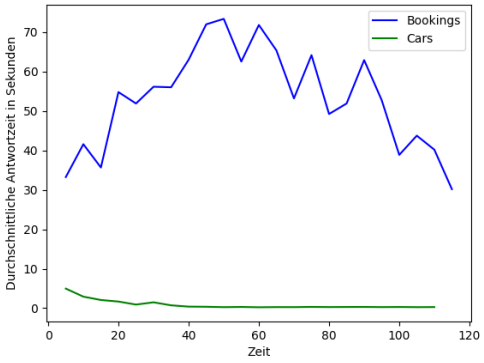

Abbildung 6.3: Threadpool Architektur Testkonfiguration Medium

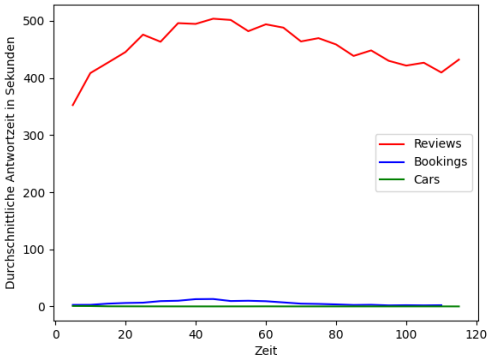

Abbildung 6.4: Threadpool Architektur Testkonfiguration Maximum

6.3 Message Queue Architektur

Die Message Queue Architektur präsentiert sich als bedeutender Fortschritt im Vergleich zu den vorherigen Architekturen, indem alle 9 Performance-Tests im Szenario Response Time erfolgreich durchgeführt werden konnten. Diese Verbesserung resultiert in erster Linie aus der Umstellung der Kommunikationsstruktur zwischen dem Pipeline-Service und der Runtime von REST auf RabbitMQ.

Dadurch wird zum einen eine präzisere Kontrolle und Regulierung der Pipeline-Runs ermöglicht. Dies erfolgt durch die Möglichkeit, die Anzahl der parallel ausgeführten Pipeline-Runs mittels anpassbarer Parameter zu regulieren. Diese Flexibilität ermöglicht eine optimale Ressourcennutzung und eine effiziente Verarbeitung von Anfragen. Zum anderen ermöglicht die Message Queue das erneute Aufnehmen abgebrochener Pipeline-Runs, was die Architektur robuster gegenüber Ausfällen der Replikate macht.

	Minimum	Medium	Maximum
Anzahl Requests	180	1200	2400
Ø Anzahl fehlerhafter Requests	0	0	0
Ø Anzahl erfolgreicher Requests	180	1200	2400
Ø Fehlerrate in Prozent	0	0	0
Ø Antwortzeit in Sekunden	0,15	0,86	0,15

Tabelle 6.10: Aggregierte Ergebnisse Message Queue Architektur Datensatz Kategorie „Klein“ (Cars)

	Minimum	Medium	Maximum
Anzahl Requests	180	1200	2400
Ø Anzahl fehlerhafter Requests	0	0	0
Ø Anzahl erfolgreicher Requests	180	1200	2400
Ø Fehlerrate in Prozent	0	0	0
Ø Antwortzeit in Sekunden	0,74	27,89	0,32

Tabelle 6.11: Aggregierte Ergebnisse Message Queue Architektur Datensatz Kategorie „Mittel“ (Bookings)

	Minimum	Medium	Maximum
Anzahl Requests	180	1200	2400
Ø Anzahl fehlerhafter Requests	0	0	0
Ø Anzahl erfolgreicher Requests	180	1200	2400
Ø Fehlerrate in Prozent	0	0	0
Ø Antwortzeit in Sekunden	83,14	272,61	361,77

Tabelle 6.12: Aggregierte Ergebnisse Message Queue Architektur Datensatz Kategorie „Groß“ (Reviews)

Die grafischen Ergebnisse der Architektur Message Queue offenbaren interessante Einblicke in das Verhalten der Anwendung während der durchgeführten Performance-Tests.
Zunächst fällt auf, dass die Antwortzeiten bei den Datensätzen der Kategorien „Klein“ (Cars) und „Mittel“ (Bookings) über den gesamten Testverlauf konstant bleiben. Hervorzuheben ist jedoch die lineare Entwicklung der Antwortzeiten bei der Kategorie „Groß“ (Reviews).

Dieses Muster ist in erster Linie auf den Warteschlangencharakter der Architektur zurückzuführen. Wenn das System mit zu hoher Last konfrontiert wird, können nicht alle Nachrichten gleichmäßig verarbeitet werden. Stattdessen kommt es zu einem Anstau von Aufgaben in der Warteschlange, die nach und nach abgearbeitet werden. Dieser Prozess ermöglicht zwar weiterhin die Verarbeitung, jedoch mit einer gewissen Verzögerung und einem linearen Anstieg der Antwortzeiten.

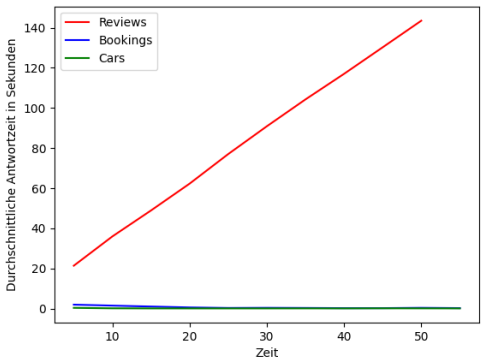

Abbildung 6.5: Message Queue Architektur Testkonfiguration Mininum

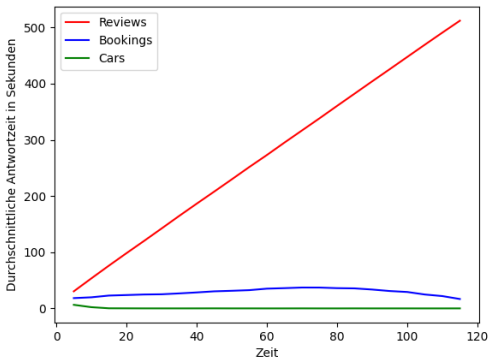

Abbildung 6.6: Message Queue Architektur Testkonfiguration Medium

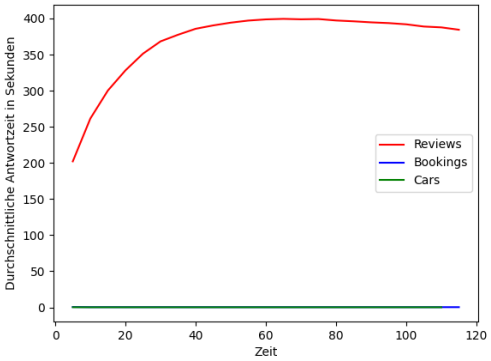

Abbildung 6.7: Message Queue Architektur Testkonfiguration Maximum

6.4 Vergleich

6.4.1 Szenario Response Time

Die Ist-Architektur offenbart in Bezug auf die verarbeiteten Pipeline-Runs signifikante Schwächen, insbesondere in der unkontrollierten Generierung von Subprozessen. Dies führt zu einem ineffizienten Ressourceneinsatz, mangelnder Kontrolle über die Anzahl der gleichzeitig laufenden Prozesse und dem Fehlen von Mechanismen zur Wiederherstellung fehlgeschlagener Pipeline-Runs. Diese Defizite resultieren in einem 100-prozentigen Fehleranteil bei 8 von 9 durchgeführten Performance-Tests.

Die Threadpool Architektur zeigt hinsichtlich der Antwortzeiten eine starke Verbesserung, indem die Pipeline-Runs effizienter abgearbeitet werden und dadurch eine deutlich bessere Antwortzeit aufzuweisen ist. Dennoch bleibt das Fehlen eines regulierenden Mechanismus für gestartete Pipeline-Runs sowie die Möglichkeit zur Wiederaufnahme fehlgeschlagener Runs bestehen. Dies führt zu vergleichbaren Herausforderungen wie bei der Ist-Architektur, wenn auch auf einer besseren Basis.

Die Message Queue Architektur präsentiert sich als deutlicher Fortschritt. Alle 9 Performance-Tests konnten erfolgreich durchgeführt werden, was auf eine verbesserte Stabilität und Robustheit der Anwendung hinweist. Die Einführung einer Message Queue ermöglicht eine präzise Kontrolle über die Anzahl der parallel laufenden Pipeline-Runs und erlaubt die Wiederaufnahme abgebrochener Verarbeitungen. Dies führt zu einer optimierten Ressourcennutzung und einer erhöhten Widerstandsfähigkeit gegenüber Verarbeitungsfehlern sowie Ausfällen der Services.

Die Beobachtungen verdeutlichen die Robustheit der Message Queue Architektur gegenüber hoher Last, zeigen aber auch die Herausforderungen bei umfangreichen Verarbeitungsaufgaben. Die lineare Verlaufsentwicklung unterstreicht die Funktionsweise des Warteschlangenprinzips und die Fähigkeit der Architektur, auch unter erhöhter Last eine kontinuierliche Verarbeitung zu gewährleisten.

Nicht nur in Bezug auf die Kontrollierbarkeit und Fehlerbehandlungsmöglichkeiten, sondern auch hinsichtlich der Antwortzeiten erweist sich die Message Queue Architektur als überlegen. Bei allen Testkonstellationen erzielt sie bessere Werte als die beiden anderen Architekturen.

In den nachfolgenden Tabellen erfolgt eine Zusammenfassung der Ergebnisse, die einen gezielten Vergleich der betrachteten Architekturen ermöglicht. Hierbei werden die aggregierten Resultate der durchgeführten Performance-Tests detailliert

dargestellt. Für jede Testkonfiguration werden die Ergebnisse der Datensatzkategorien in einzelnen Tabellen aufgeführt. Diese strukturierte Darstellung erleichtert eine präzise Analyse und Vergleich der Performance der Architekturen.
Eine grafische Repräsentation der Ergebnisse ist im Anhang E zu finden.

Ergebnisse Testkonfiguration Minimum

	Ist-Architektur	Threadpool	Message Queue
Ø Anzahl fehlerhafter Requests	12	0	0
Ø Anzahl erfolgreicher Requests	168	180	180
Ø Fehlerrate in Prozent	7,14	0	0
Ø Antwortzeit in Sekunden	474,44	0,32	0,15

Tabelle 6.13: Aggregierte Ergebnisse Datensatz Kategorie „Klein“ (Cars)

	Ist-Architektur	Threadpool	Message Queue
Ø Anzahl fehlerhafter Requests	180	0	0
Ø Anzahl erfolgreicher Requests	0	180	180
Ø Fehlerrate in Prozent	100	0	0
Ø Antwortzeit in Sekunden	-	2,32	0,74

Tabelle 6.14: Aggregierte Ergebnisse Datensatz Kategorie „Mittel“ (Bookings)

	Ist-Architektur	Threadpool	Message Queue
Ø Anzahl fehlerhafter Requests	180	8	0
Ø Anzahl erfolgreicher Requests	0	172	180
Ø Fehlerrate in Prozent	100	4,6	0
Ø Antwortzeit in Sekunden	-	308,30	83,14

Tabelle 6.15: Aggregierte Ergebnisse Datensatz Kategorie „Groß“ (Reviews)

Ergebnisse Testkonfiguration Medium

	Ist-Architektur	Threadpool	Message Queue
Ø Anzahl fehlerhafter Requests	1200	3	0
Ø Anzahl erfolgreicher Requests	0	1197	1200
Ø Fehlerrate in Prozent	100	0,25	0
Ø Antwortzeit in Sekunden	-	1.14	0,86

Tabelle 6.16: Aggregierte Ergebnisse Datensatz Kategorie „Klein“ (Cars)

	Ist-Architektur	Threadpool	Message Queue
Ø Anzahl fehlerhafter Requests	1200	3	0
Ø Anzahl erfolgreicher Requests	0	1197	1200
Ø Fehlerrate in Prozent	100	0,25	0
Ø Antwortzeit in Sekunden	-	51,66	27,89

Tabelle 6.17: Aggregierte Ergebnisse Datensatz Kategorie „Mittel“ (Bookings)

	Ist-Architektur	Threadpool	Message Queue
Ø Anzahl fehlerhafter Requests	1200	1200	0
Ø Anzahl erfolgreicher Requests	0	0	1200
Ø Fehlerrate in Prozent	100	100	0
Ø Antwortzeit in Sekunden	-	-	272.61

Tabelle 6.18: Aggregierte Ergebnisse Datensatz Kategorie „Groß“ (Reviews)

Ergebnisse Testkonfiguration Maximum

	Ist-Architektur	Threadpool	Message Queue
Ø Anzahl fehlerhafter Requests	2400	0	0
Ø Anzahl erfolgreicher Requests	0	2400	2400
Ø Fehlerrate in Prozent	100	0	0
Ø Antwortzeit in Sekunden	-	0,32	0,15

Tabelle 6.19: Aggregierte Ergebnisse Datensatz Kategorie „Klein“ (Cars)

	Ist-Architektur	Threadpool	Message Queue
Ø Anzahl fehlerhafter Requests	2400	6	0
Ø Anzahl erfolgreicher Requests	0	2394	2400
Ø Fehlerrate in Prozent	100	0,25	0
Ø Antwortzeit in Sekunden	-	5,69	0,32

Tabelle 6.20: Aggregierte Ergebnisse Datensatz Kategorie „Mittel“ (Bookings)

	Ist-Architektur	Threadpool	Message Queue
Ø Anzahl fehlerhafter Requests	2400	6	0
Ø Anzahl erfolgreicher Requests	0	2400	2400
Ø Fehlerrate in Prozent	100	0,25	0
Ø Antwortzeit in Sekunden	-	447,35	361,77

Tabelle 6.21: Aggregierte Ergebnisse Datensatz Kategorie „Groß“ (Reviews)

6.4.2 Szenario Integration

Die Ergebnisse des Testszenarios Integration bestätigen und vertiefen die Erkenntnisse aus dem Szenario Response Time. Aufgrund der erhöhten Last in diesem Testszenario sind die Replikate der Ist-Architektur und der Threadpool Architektur bei allen durchgeführten Tests abgestürzt, was die Limitationen dieser Architekturen unter höherer Belastung verifiziert.

Die Zuverlässigkeitsrate, in dieser Arbeit definiert als der Prozentsatz der Anfragen, die erfolgreich verarbeitet oder aktiv vom System abgelehnt wurden, bietet Einblicke in die Robustheit der Architekturen. Dieser Aspekt ist von zentraler Bedeutung, um die Fähigkeit der Architekturen zur korrekten Verarbeitung von Anfragen unter unterschiedlichen Bedingungen zu beurteilen. In dieser Bewertung werden nicht nur die korrekte Bearbeitung der Pipeline-Runs als positiv angesehen, sondern auch die Fähigkeit der Anwendung, unter zu hoher Last zu erkennen, dass die Ressourcen erschöpft sind, und daraufhin temporär weitere Anfragen ablehnt. Diese zusätzliche Dimension der Bewertung repräsentiert die Fähigkeit der Architekturen, proaktiv auf Überlastsituationen zu reagieren und somit eine bessere Kontrolle über ihre Leistungsfähigkeit und Verfügbarkeit zu demonstrieren.

Die Verfügbarkeit im Kontext dieser Untersuchung wird als der Prozentsatz der Anfragen definiert, die angenommen oder aktiv vom System abgelehnt wurden. Die Verfügbarkeit gibt Aufschluss darüber, wie viele Anfragen erfolgreich angenommen oder bewusst abgelehnt wurden. Dieser Parameter ist entscheidend, um die Fähigkeit der Architekturen zur Bewältigung von Anfragen und zur Vermeidung von Überlastsituationen zu beurteilen. Darüber hinaus ist es von zentraler Bedeutung, die Verfügbarkeit zu betrachten, um zu bewerten, wie die Anwendung unter extremen Bedingungen, einschließlich hoher Last und sogar Replikatausfällen, reagiert und weiterhin erreichbar bleibt. Diese Aspekte tragen maßgeblich zur umfassenden Beurteilung der Robustheit und Widerstandsfähigkeit der Architekturen bei.

Die nachfolgenden Tabellen 6.22, 6.23 und 6.24 bieten einen detaillierten Vergleich der drei betrachteten Architekturen hinsichtlich ihrer Zuverlässigkeitsrate, Verfügbarkeit und der durchschnittlichen Anzahl der Requests pro Sekunde.

	Ist-Architektur	Threadpool	Message Queue
Zuverlässigkeitsrate	0	0	100
Verfügbarkeit in %	99	100	100
Ø Requests pro Sekunde	4,63	11,46	15,96

Tabelle 6.22: Aggregierte Ergebnisse Testkonfiguration Minimum

	Ist-Architektur	Threadpool	Message Queue
Zuverlässigkeitsrate	0	0	100
Verfügbarkeit in %	79	100	100
Ø Requests pro Sekunde	9,37	104,89	116,1

Tabelle 6.23: Aggregierte Ergebnisse Testkonfiguration Medium

	Ist-Architektur	Threadpool	Message Queue
Zuverlässigkeitsrate	0	0	100
Verfügbarkeit in %	70	100	100
Ø Requests pro Sekunde	14,94	187,51	195,59

Tabelle 6.24: Aggregierte Ergebnisse Testkonfiguration Maximum

Die nachfolgenden Grafiken bieten eine visuelle Darstellung des Verlaufs der drei Architekturen bezüglich der Metrik RPS. Eine auffällige Ähnlichkeit zeigt sich zwischen dem Verlauf der Threadpool und Message Queue Architektur, wobei beide Architekturen eine vergleichsweise stabile Anzahl von RPS mit geringen Schwankungen aufweisen. Besonders hervorzuheben sind dabei die besseren Werte bei der Message Queue Architektur, was auf eine effizientere Verarbeitung von Anfragen und eine stabilere Performance im Vergleich zum Threadpool sowie der Ist-Architektur hinweist.

Die Ist-Architektur präsentiert im Allgemeinen eine konstante Performance, unterbrochen durch markante Abfälle, die auf den Absturz von Runtime-Replikaten zurückzuführen sind. Diese Abfälle führen zu einem deutlichen Einbruch der RPS, da keine Anfragen vom Pipeline-Service an die Runtime weitergeleitet werden können. Zusätzlich zeigt sich dadurch, dass die Verfügbarkeit der Ist-Architektur während Phasen mit hoher RPS kurzzeitig beeinträchtigt ist, was die Herausforderungen bezüglich der Lastbewältigung und der Systemstabilität bestätigt.

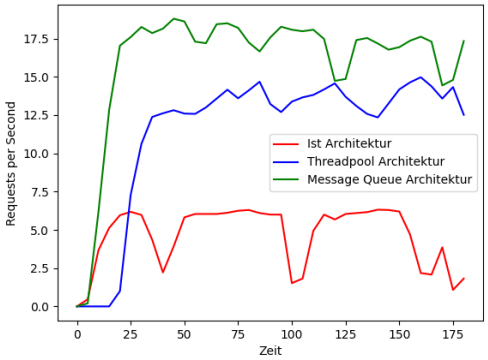

Abbildung 6.8: Szenario Integration Testkonfiguration Mininum

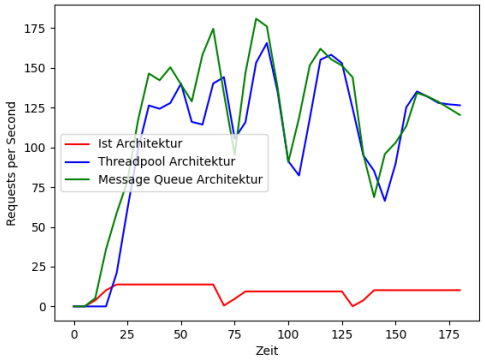

Abbildung 6.9: Szenario Integration Testkonfiguration Medium

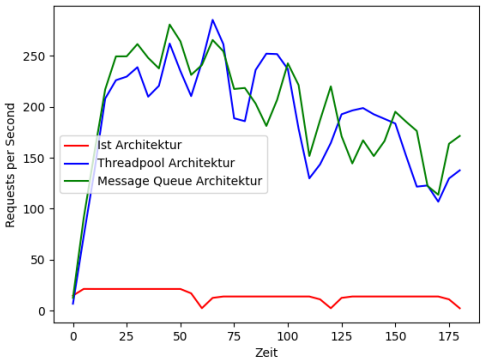

Abbildung 6.10: Szenario Integration Testkonfiguration Maximum

6.5 Anforderungen

In den nachfolgenden Abschnitten werden die in Kapitel 3 definierten Anforderungen mit den drei Architekturen verglichen. Dabei steht „X“ in den folgenden Tabellen dafür, dass die Anforderung erfüllt wurde, „(X)“ für teilweise erfüllt und „-“ für gar nicht erfüllt.

6.5.1 Technische Anforderungen

Die technischen Anforderungen wurden von allen drei Architekturen erfüllt. Die verwendeten Technologien und Tools für die Durchführung der Performance-Tests sowie Implementierung der alternativen Architekturen wurden in Kapitel 4 beschrieben.

Anforderung	Ist-Architektur	Threadpool	Message Queue	Begründung
RQ1	X	X	X	siehe 4
RQ2	X	X	X	siehe 4
RQ3	X	X	X	siehe 4
RQ4	X	X	X	siehe 4
RQ5	X	X	X	siehe 4

Tabelle 6.25: Technische Anforderungen

6.5.2 Performance Anforderungen

Antwortzeiten

Die Message Queue Architektur hat die Anforderungen bezüglich der Antwortzeiten vollständig erfüllt, während die Ist-Architektur lediglich die Anforderungen in Bezug auf die Datensatzkategorie „Klein“ (Cars) PRQ1 und PRQ2 abgdeckt. Die Threadpool Architektur hingegen hält die Anforderungen PRQ5 und PRQ6 nur teilweise ein. Dies liegt daran, dass die Performance-Tests mit der Datensatzkategorie „Groß“ (Reviews) unter der Testkonfiguration Medium nicht durchgeführt werden konnten. Eine Übersicht der Anforderungen hinsichtlich der Antwortzeiten ist in Tabelle 6.26 zu sehen.

Robustheit

Sowohl die Ist-Architektur als auch die Threadpool Architektur erfüllen die Anforderungen der Robustheit nur teilweise. Die Performance-Tests haben gezeigt, dass bei einem Ausfall der Runtime Komponente das Starten von Pipeline-Runs zwar angefragt werden kann und mit einer ID bestätigt wird, jedoch die Verarbeitung nicht durchgeführt wird. Zudem wird der Aufrufer über dieses Fehlverhalten

Anforderung	Ist-Architektur	Threadpool	Message Queue	Begründung
PRQ1	X	X	X	siehe 6.13
PRQ2	X	X	X	siehe 6.13
PRQ3	-	X	X	siehe 6.17
PRQ4	-	X	X	siehe 6.17
PRQ5	-	(X)	X	siehe 6.21
PRQ6	-	(X)	X	siehe 6.21

Tabelle 6.26: Performance Anforderungen Antwortzeiten

nicht informiert und der Pipeline-Run steckt endlos im Status „Running“ fest. Die Message Queue Architektur wirkt diesem Problem durch den Einsatz von Warteschlangen entgegen. Solange der Runtime-Service nicht erreichbar ist, werden die Anfragen gesammelt und können wieder verarbeitet werden, sobald der Service erneut verfügbar ist. In Tabelle 6.27 ist eine Übersicht der Anforderungen bezüglich der Robustheit zu sehen.

Anforderung	Ist-Architektur	Threadpool	Message Queue	Begründung
PRQ7	(X)	(X)	X	siehe 6.4.2
PRQ8	(X)	(X)	X	siehe 6.4.2

Tabelle 6.27: Performance Anforderungen Robustheit

Verfügbarkeit

Abschnitt 6.4.2 legt dar, dass die Anforderungen hinsichtlich der Verfügbarkeit bei Threadpool und Message Queue Architekturen erfüllt sind. Es ist jedoch anzumerken, dass die Verfügbarkeit keine Aussage darüber trifft, ob die Pipeline-Runs erfolgreich verarbeitet werden. Die Ist-Architektur erfüllt die Anforderungen teilweise, da ihre Verfügbarkeit bei der Durchführung der Performance Tests zum Teil eingeschränkt war. Tabelle 6.28 zeigt eine Zusammenfassung der Anforderungen zur Verfügbarkeit.

Anforderung	Ist-Architektur	Threadpool	Message Queue	Begründung
PRQ9	(X)	X	X	siehe 6.4.2
PRQ10	(X)	X	X	siehe 6.22
PRQ11	(X)	X	X	siehe 6.23
PRQ12	(X)	X	X	siehe 6.24

Tabelle 6.28: Performance Anforderungen Verfügbarkeit

Zuverlässigkeit

Im Gegensatz zur Verfügbarkeit kann weder die Ist-Architektur noch die Threadpool Architektur die Anforderungen der Zuverlässigkeit umsetzen. Der Vergleich in Kapitel 6.4.2 legt dar, dass die Anfragen unter der Last der Performance-Tests nicht erfüllt werden konnten. Die Message Queue Architektur hingegen erfüllt sämtliche Anforderungen. In Tabelle 6.29 ist ein Überblick dieser zu sehen.

Anforderung	Ist-Architektur	Threadpool	Message Queue	Begründung
PRQ13	-	-	X	siehe 6.22
PRQ14	-	-	X	siehe 6.23
PRQ15	-	-	X	siehe 6.24

Tabelle 6.29: Performance Anforderungen Zuverlässigkeit

Durchsatz

Die Ist-Architektur kann die Anforderungen bezogen auf den Durchsatz nicht erfüllen. Sie kann zwar über den gesamten Testverlauf eine konstante Anzahl an RPS aufweisen, jedoch gibt es deutliche Einbrüche, die auf den Ausfall der Runtime zurückzuführen sind. Die alternativen Architekturen erfüllen die Anforderung PRQ16. Allerdings werden PRQ17 und PRQ18 nur teilweise erfüllt, da kein Mechanismus existiert, der die ankommenden Anfragen bei steigender Last und Ressourcennutzung limitiert. Die Anforderungen des Durchsatzes sind in Tabelle 6.30 dargestellt.

Anforderung	Ist-Architektur	Threadpool	Message Queue	Begründung
PRQ16	-	X	X	siehe 6.8
PRQ17	-	(X)	(X)	siehe 6.9
PRQ18	-	X	X	siehe 6.10

Tabelle 6.30: Performance Anforderungen Durchsatz

7 Schlussfolgerung

7.1 Limitierungen

Die vorliegende Arbeit weist bestimmte Limitationen auf, die bei der Interpretation der Ergebnisse und der Generalisierung der Erkenntnisse berücksichtigt werden müssen.

Eine wesentliche Limitation besteht darin, dass in den durchgeführten Performance-Tests keine Autoskalierung verwendet wurde. Diese Entscheidung wurde getroffen, um sicherzustellen, dass effizientere Architekturen nicht „bestraft“ werden, indem sie aufgrund ihrer Effizienz weniger Ressourcen benötigen. Damit soll gewährleistet werden, dass die unterschiedlichen Architekturen unter denselben Rahmenbedingungen getestet werden. Zusätzlich würde die Implementierung von Autoskalierung das Problem der zu vielen parallel laufenden Pipeline-Runs lediglich nach „hinten“ verlagern und könnte somit die Vergleichbarkeit der Architekturen beeinträchtigen.

Die begrenzten Ressourcen in der Infrastruktur und im Testsystem stellen eine weitere Limitation dar. Die Tests wurden unter den vorhandenen Ressourcenbedingungen durchgeführt. Darüber hinaus basieren die Ergebnisse auf bestimmten Annahmen über die Testkonfiguration und Parametrisierung. Diese Annahmen decken nicht alle realen Szenarien und Konfigurationen ab, was die Generalisierbarkeit der Ergebnisse beeinflussen könnte.

Ein weiterer Aspekt ist das „Aufwärmen“ des Systems. Da Replika ohne Arbeit initial keine Memory-Allokation durchführen, kann dies zu einer zeitlichen Verzerrung der Ergebnisse führen. Dieser Aspekt sollte bei der Interpretation der Antwortzeiten berücksichtigt werden.

7.2 Future Work

Die vorliegende Arbeit legt den Grundstein für weitere Untersuchungen und bietet zahlreiche Ansatzpunkte für zukünftige Arbeiten im Bereich der Performance

Evaluierung von Microservice Architekturen. Die betrachteten Architekturansätze bilden nur einen begrenzten Teil des breiten Spektrums möglicher Implementierungen ab.

Ein vielversprechendes Konzept für zukünftige Forschung könnte die Exploration von alternativen Architekturansätzen, wie die Kommunikation mittels gRPC[1], sein. Die Integration von gRPC könnte nicht nur neue Erkenntnisse in Bezug auf die Interaktion und Effizienz von Microservices bieten, sondern auch eine umfassendere Grundlage für den Vergleich alternativer Architekturen schaffen.

Des Weiteren bietet sich die Möglichkeit einer feingranularen Optimierung der Architektur und der eingesetzten Tools. Beispielsweise können Parameter und Einstellungen von Messaging-Systemen wie RabbitMQ weiter angepasst werden, um die Performance der Warteschlange zu optimieren. Solche detaillierten Anpassungen könnten zu signifikanten Verbesserungen der Gesamtleistung führen.

Ein aussichtsreicher Pfad für künftige Untersuchungen liegt auch in der Erweiterung der Metrikpalette. Neben den in dieser Arbeit behandelten Metriken könnten zusätzliche Leistungsindikatoren in die Analyse einbezogen werden, um ein umfassenderes Bild der Anwendungsleistung zu erhalten. Komplexere Szenarien, wie absichtliche Verbindungsabbrüche oder verschiedene Methodiken zur automatischen Skalierung der Anwendung, könnten ebenfalls in die Analyse integriert werden.

Kommende Untersuchungen können somit dazu beitragen, die Ergebnisse dieser Arbeit zu vertiefen und neue Erkenntnisse in Bezug auf die Performance von Microservice Architekturen zu gewinnen. Durch die Integration weiterer Architekturansätze, Optimierungsmöglichkeiten und erweiterter Metriken kann ein fortlaufender Beitrag zur Weiterentwicklung und Verfeinerung von Best Practices im Bereich der Softwarearchitektur für Microservices und insbesondere deren Kommunikation geleistet werden.

7.3 Empfehlung

Die gewonnenen Ergebnisse verdeutlichen Probleme und Herausforderungen in der Ist-Architektur, die eine umfassende Überarbeitung erforderlich machen. Insbesondere die unkontrollierte Erzeugung von Subprozessen führt zu erheblichem Overhead und erhöhter Fehleranfälligkeit. Wie im Fall der Ist-Architektur sichtbar wird, führt dieses Vorgehen zu einem unkontrollierten Anstieg an

[1] https://grpc.io/

Prozessen, was letztendlich zum Absturz der Services und zum Fehlschlagen sämtlicher Anfragen führt.

Die Performance der Threadpool Architektur zeigt sich hingegen deutlich verbessert durch die effiziente Nutzung der internen Mechanismen von Node.js. Dennoch stößt auch diese Architektur bei steigender Last an ihre Grenzen und weist dabei ähnliche Herausforderungen wie die Ist-Architektur auf.

Eine kritische Herausforderung besteht darin, einen Mechanismus zu implementieren, der den Ressourcenverbrauch überwacht und bei zu hoher Last Anfragen ablehnt. Der zusätzliche Entwicklungsaufwand birgt jedoch die Gefahr, die Komplexität zu erhöhen und erfordert daher eine sorgfältige und gründliche Herangehensweise, um eine reibungslose Implementierung sicherzustellen.

Vor diesem Hintergrund ergibt sich eine klare Empfehlung für die Message Queue Architektur. Sie zeigt in allen durchgeführten Tests eine verbesserte Performance, vor allem hinsichtlich der Antwortzeiten. Diese kann durch Konfiguration noch weiter optimiert werden. Beispielsweise kann die Deaktivierung von Publisher Acks sowie eine erhöhte parallele Verarbeitung der Nachrichten in der Warteschlange dazu führen, die Response Time des Systems zu verbessern. Es ist jedoch anzumerken, dass diese Maßnahmen mit Einbußen in der Robustheit einhergehen.

Zusätzlich bietet die Message Queue Architektur Out-of-the-box-Lösungen für Zuverlässigkeit, Robustheit und Verfügbarkeit. So können ohne erhöhten Entwicklungsaufwand Mechanismen zur Regulierung der Pipeline-Runs und Wiederaufnahme von fehlgeschlagenen Pipeline-Runs in das System integriert werden. Um dem linearen Verlauf der Antwortzeiten bei extrem hoher Last entgegenzuwirken, kann zudem ein Timeout für Nachrichten in der Warteschlange aktiviert werden. Infolge der steigenden Belastung können Pipeline-Runs, die aufgrund langer Wartezeiten auf eine Verarbeitung warten müssen, vom System aktiv abgelehnt und als fehlgeschlagen markiert werden, um eine akzeptable Antwortzeit zu gewährleisten.

Insgesamt bietet die Message Queue Architektur eine solide Grundlage mit integrierten Features und Konfigurationsmöglichkeiten für eine optimale Performance, Zuverlässigkeit, Robustheit und Verfügbarkeit einer Anwendung.

7.4 Zusammenfassung

Die vorliegende Arbeit hat eingehende Performance-Analysen verschiedener Microservice Architekturen für die Open-Source-Anwendung JValue vorgenommen.

Die erzielten Ergebnisse bieten wertvolle Einblicke und Orientierungspunkte für Entwickler bei der Auswahl geeigneter Architekturen.

Die Resultate der Untersuchung verdeutlichen gravierende Probleme in der Ist-Architektur, insbesondere durch die unkontrollierte Erzeugung von Subprozessen. Die Threadpool Architektur zeigt zwar eine klar verbesserte Performance, stößt jedoch auch bei hoher Last an ihre Grenzen. Insgesamt bietet die Message Queue Architektur eine solide Grundlage mit integrierten Features und Konfigurationsmöglichkeiten bezüglich der Zuverlässigkeit, Robustheit und Verfügbarkeit. Im Gegensatz dazu erfordern ähnliche Verbesserungen in den anderen Architekturen einen erheblichen Entwicklungsaufwand, welcher mit zusätzlichen Risiken verbunden ist. Sowohl die Ist-Architektur als auch die Threadpool Architektur weisen erhebliche Mängel auf, insbesondere in der parallelen Verarbeitung von Pipeline-Runs sowie der Wiederherstellung fehlgeschlagener Pipeline-Runs. Diese Defizite führen zu einem hohen Fehleranteil bei den durchgeführten Performance-Tests.

Es ist jedoch wichtig zu betonen, dass Architekturentscheidungen nicht ausschließlich auf Basis der Performance getroffen werden sollten. Sie sind ein komplexer Prozess, welche die Berücksichtigung einer Vielzahl von Faktoren erfordert. Neben der Performance spielen auch Aspekte wie Skalierbarkeit, Wartbarkeit, Limitierungen durch bestehende Infrastruktur sowie Erfahrungen und Fähigkeiten der beteiligten Entwickler eine entscheidende Rolle. Konsistenzkriterien und spezifische Anwendungsanforderungen sind ebenfalls von großer Bedeutung und müssen in die Entscheidung einfließen.

Die Auswahl der Architektur sollte als ganzheitlicher Ansatz betrachtet werden, der die individuellen Anforderungen und Rahmenbedingungen einer Anwendung umfassend berücksichtigt. Die erzielten Ergebnisse dienen dabei als wichtige, jedoch nicht alleinige Grundlage für fundierte Architekturentscheidungen.

Entwickler sollten sich bewusst sein, dass jede Architektur ihre eigenen Stärken und Schwächen hat und dass die Wahl einer geeigneten Architektur eine Abwägung verschiedener Kriterien erfordert. Die vorliegenden Ergebnisse können dabei als Leitfaden dienen, jedoch ist eine sorgfältige Abstimmung mit den individuellen Anforderungen und Zielen der Anwendung unerlässlich.

Insgesamt soll diese Schlussfolgerung dazu ermutigen, Architekturentscheidungen in einem breiteren Kontext zu betrachten und verschiedene Aspekte gleichzeitig zu evaluieren, um eine umfassende und nachhaltige Grundlage für die Entwicklung von leistungsfähigen und zukunftsfähigen Anwendungen zu schaffen.

Darüber hinaus bietet die vorliegende Untersuchung nicht nur Erkenntnisse

zur Performance verschiedener Microservice Architekturen, sondern birgt auch einen beträchtlichen Mehrwert für die breitere Entwicklergemeinschaft. Der detailliert beschriebene Versuchsaufbau fungiert als solide Grundlage, die für die Durchführung von Performance-Tests in unterschiedlichen Anwendungen genutzt werden kann.

Der Wert dieses Versuchsaufbaus liegt nicht allein im Endergebnis der durchgeführten Tests, sondern insbesondere in der Möglichkeit, Schwachstellen und Fehler in der Architektur sowie der Code Logik zu identifizieren und zu beheben. Diese Erkenntnisse gehen über die reine Performance-Analyse hinaus und ermöglichen eine tiefgreifende Verbesserung der Anwendungsstabilität undsicherheit.

Ein praxisnahes Beispiel dieser Erkenntnisse ist bereits in der Anwendung JValue implementiert worden. Durch die Erkenntnisse aus der vorliegenden Arbeit wurde auf eine Subprozess Erzeugung beim Verarbeiten der Pipeline-Runs verzichtet. Außerdem wurden mehrere Bugs identifiziert und behoben. Eine Race Condition in einer Stored Procedure beim Umbenennen von Datenbanktabellen wurde adressiert, ebenso wie ein potenzieller Angriffsvektor einer SQL Injection.

Weiterhin wurde eine Race Condition beim Jayvee Interpreter bei paralleler Ausführung erkannt und behoben. Zusätzlich wurde in der Konfiguration von Node.js auf eine Option aufmerksam gemacht, die es ermöglicht, den „heap size limit“ anzupassen. Dies ist von entscheidender Bedeutung, da das Standardlimit auf 2 GB festgelegt ist. Wenn ein Replikat mehr als 2 GB allokieren möchte, führt dies zu Fehlern und letztendlich zum Absturz der Services. Die Einführung dieser Anpassungsoption trägt maßgeblich zur Stabilität der Services bei, indem sie die Auswirkungen von Speicherüberlastungen effektiv adressiert und die Verfügbarkeit sowie Robustheit des Systems verbessert.

Diese praxisorientierte Anwendung der gewonnenen Erkenntnisse unterstreicht die unmittelbare Relevanz der durchgeführten Performance-Tests. Sie ermöglichen nicht nur eine Optimierung der Performance, sondern tragen auch zur Stabilität, Sicherheit und Qualität der Softwareanwendung bei.

Insgesamt zeigen diese Schlussfolgerungen, dass die vorliegende Arbeit praxisrelevante Auswirkungen auf reale Anwendungen hat und einen Beitrag zur kontinuierlichen Verbesserung von Softwarearchitektur im Kontext von Microservices leistet.

Anhänge

A Kubernetes Cluster Setup

Voraussetzungen zum Installieren und Durchführen der Performance-Tests:
SSH-Verbindung zu den Servern der FAU (vpn.fau.de)
Kube-Config ist im Anhang zu finden
Clone GitHub Repository and CD into repo

```
git clone https://github.com/jvalue/hub.git
cd hub
```

Postgres Operator (DB)
Add repo for postgres-operator:

```
helm repo add postgres-operator-charts
--namespace jvalue-doell
https://opensource.zalando.com/
postgres-operator/charts/postgres-operator"
```

Install the postgres-operator:

```
helm upgrade --install postgres-operator
--namespace jvalue-doell
postgres-operator-charts/postgres-operator
```

Wait for it to run, check with:

```
kubectl --namespace=jvalue-doell get pods -l
"app.kubernetes.io/name=postgres-operator"
```

Prometheus Operator (Monitoring) add repo for kube-prometheus

```
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
```

Install kube-prometheus

```
helm upgrade --install kube-prometheus
prometheus-community/kube-prometheus-stack
```

Add helm repo

```
helm repo add jetstack
https://charts.jetstack.io
helm repo update
```

Namespace Setup
Create namespace

```
kubectl create namespace jvalue-doell
```

Create secret to download private container images from github

```
kubectl create secret docker-registry
ghcr-login-secret --namespace jvalue-doell
--docker-server=https://ghcr.io
--docker-username=$JVALUE_USERNAME$
--docker-password=$JVALUE_PASSWORD$
--docker-email=jvaluebot@group.riehle.org
```

Create JWT secrets for apps

```
kubectl create secret generic jwt-secrets
-n jvalue-doell
--from-literal=pipeline-service=123
--from-literal=hub-backend=123
--from-literal=file-service=123
```

Create secret with account for pipeline-service

```
kubectl create secret generic
pipeline-service-account-secret
-n jvalue-doell
--from-literal=username=user
--from-literal=password=password
```

Install nginx ingress controller

```
helm upgrade --install
ingress-nginx ingress-nginx
--repo
https://kubernetes.github.io/ingress-nginx
--namespace jvalue-doell
```

To wait for it to be up:

```
kubectl wait
--namespace jvalue-doell
--for=condition=ready pod
--selector=app.kubernetes.io/component=controller
--timeout=120s
```

Install internal DB

```
helm upgrade --install hub-db
--namespace
jvalue-doell
_k8s/helm/charts/db --values
_k8s/helm/charts/db/values.doell.yaml
```

Install public db

```
helm upgrade --install
public-db
--namespace jvalue-doell
_k8s/helm/charts/public-db
--values
_k8s/helm/charts/public-db/values.doell.yaml
```

Releases der Architekturen zum Stand des Tests:
„interpreter“ (Threadpool Architektur)
„ist architektur“ (Ist-Architektur)
„rabbitmq“ (Message Queue Architektur)

Branches der Architekturen zum Stand des Tests:
„architektur interpreter lib“ (Threadpool Architektur)
„test ist“ (Ist-Architektur)
„architektur rabbitmq“ (Message Queue Architektur)

Installieren der Runtime

```
helm upgrade
--install runtime-simple
--namespace jvalue-doell
_k8s/helm/charts/services/charts/
runtime-simple
--values
_k8s/helm/charts/services/charts/
runtime-simple/values.doell.yaml
```

Installieren Pipeline-Service

```
helm upgrade
--install pipeline-service
--namespace jvalue-doell
_k8s/helm/charts/services/charts/
pipeline-service --values
_k8s/helm/charts/services/charts/
pipeline-service/values.doell.yaml
```

Installieren von RabbitMQ auf Kubernetes:
(https://www.rabbitmq.com/docs/
kubernetes/operator/
quickstart-operator)

Installieren von Locust auf Docker:
(https://docs.locust.io/en/2.0.0/
running-locust-docker.html

#running-locust-docker)

Starten von wrk2 auf Docker:

```
"Pfad␣zu␣den␣Skript-Ordnern"
cylab/wrk2 -R 50
-c 5 -t 1 -d 20s -s
"Pfad␣zum␣ausfuehrenden␣Skript"
http://10.131.64.39
```

B Jayvee Modelle

```
pipeline CarsPipeline {

        block CarsExtractor oftype HttpExtractor {
                url: "$LINK_ZU_DEN_DATEN$";
        }

        pipe {
                from: CarsExtractor;
                to: CarsTextFileInterpreter;
        }

        block CarsTextFileInterpreter
        oftype TextFileInterpreter {

        }

        pipe {
                from: CarsTextFileInterpreter;
                to: CarsCSVInterpreter;
        }

        block CarsCSVInterpreter
        oftype CSVInterpreter {
                enclosing: '"';
        }

        pipe {
                from: CarsCSVInterpreter;
                to: CarsTableInterpreter;
        }

        block CarsTableInterpreter
        oftype TableInterpreter {
                header: true;
                columns: [
                        "name" oftype text,
                        "mpg" oftype decimal,
                        "cyl" oftype integer,
                        "disp" oftype decimal,
                        "hp" oftype integer,
```

```
                        "drat" oftype decimal,
                        "wt" oftype decimal,
                        "qsec" oftype decimal,
                        "vs" oftype integer,
                        "am" oftype integer,
                        "gear" oftype integer,
                        "carb" oftype integer
                ];
        }
}

pipeline ReviewsPipeline {

        block ReviewsExtractor
        oftype HttpExtractor {
                url: "https://gist.githubusercontent.com/
            MarcoDoell/3fb174fc1c3d9b332473d5a81a6bad31/
            raw/b43c967a87213a08f67c453e88bdeaac560c616f/
            reviews_new.csv";
        }

        pipe {
                from: ReviewsExtractor;
                to: ReviewsTextFileInterpreter;
        }

        block ReviewsTextFileInterpreter
        oftype TextFileInterpreter {

        }

        pipe {
                from: ReviewsTextFileInterpreter;
                to: ReviewsCSVInterpreter;
        }

        block ReviewsCSVInterpreter
        oftype CSVInterpreter {
                enclosing: '"';
        }

        pipe {
                from: ReviewsCSVInterpreter;
```

```
            to: ReviewsTableInterpreter;
        }


        block ReviewsTableInterpreter
        oftype TableInterpreter {
                header: true;
                columns: [
                        Review_Title oftype text,
                        Review_Text oftype text,
                        Verified_Buyer oftype boolean,
                        Review_Date oftype text,
                        Review_Location oftype text,
                        Review_Upvotes oftype integer,
                        Review_Downvotes oftype integer,
                        Product oftype text,
                        Brand oftype text,
                        Scrape_Date oftype text
                ];
        }
}

    pipeline BookingsPipeline {

        block BookingsExtractor
        oftype HttpExtractor {
        url: https://gist.githubusercontent.com
        /MarcoDoell/
        b76ecd6a778d4c53db29d2a5d1f1974b/raw/
        6b466f263fd538bd52ad704cca86811db9193af4/
        bookings.csv;
        }

        pipe {
                from: BookingsExtractor;
                to: BookingsTextFileInterpreter;
        }

        block BookingsTextFileInterpreter
        oftype TextFileInterpreter {

        }
```

```
        pipe {
                from: BookingsTextFileInterpreter;
                to: BookingsCSVInterpreter;
        }

        block BookingsCSVInterpreter
        oftype CSVInterpreter {
                enclosing: '';
        }

        pipe {
                from: BookingsCSVInterpreter;
                to: BookingsTableInterpreter;
        }

        block BookingsTableInterpreter
        oftype TableInterpreter {
                header: true;
                columns: [
                         oftype integer,
                        Hotel_Name oftype text,
                        Review oftype text,
                        Total_Review oftype integer,
                        Rating oftype decimal,
                        Location oftype text
                ];
        }
}
```

C Implementierung: Performance-Tests

Locust File Szenario Integration

```
1  from locust import HttpLocust, HttpUser, FastHttpUser, task
2  import uuid
3  import os.path
4  import locust.stats
5
6  class PipelineUserSmallTest(FastHttpUser):
7
8      #Host on Local: http://10.131.64.39
9      #Host on Kube: http://pipeline-service-service.
10     jvalue-doell.svc.cluster.local:3002
```

```
11      pipelineModel = "insert pipeline model"
12      run_ids = []
13
14      def on_start(self):
15          try:
16              run_ids = []
17              response = self.client.post("/pipeline-service/auth/
18              login?username=user&password=password")
19
20              json_var = response.json()
21              token = json_var['access_token']
22              self.client.headers = {'Authorization': 'Bearer '
23              + token}
24              self.run_uuid = uuid.uuid4()
25          except Exception as e:
26              print(e)
27
28      @task
29      def create_pipeline(self):
30
31          try:
32              response = self.client.post("/pipeline-service/
33              instances", json=
34              {
35              "pipelineModel": self.pipelineModel,
36              "targetRuntime": "simple"
37              }
38              )
39              json_response_dict = response.json()
40              instance_id = json_response_dict['id']
41
42              response = self.client.post("/pipeline-service/
43              instances/+ instance_id + "/runs")
44              json_var = response.json()
45              print(response)
46              run_id = json_var['runId']
47              self.run_ids.append(instance_id + "/runs/" + run_id)
48          except Exception as e:
49              print(e)
50
51
52      def on_stop(self):
53          #Copy Output to file
54          save_path = r"C:\Users\marco\IdeaProjects\
55          locust-experiments\kubernetes\results"
```

```
56          name_of_file = "results.txt"
57
58          completeName = os.path.join(save_path, name_of_file)
59          f = open(completeName, "x")
60          f.write(' '.join(self.run_ids))
61          f.close()''
62
63      def checkTestIsLocalOrOnCluster(self):
64          pass
65
66
67
```

Lua Skript Szenario Response Time

```
1  wrk.method = "POST"
2  wrk.headers['Authorization']="Bearer TOKEN"
3
4  runids_ = {}
5
6  request = function()
7      instance_id = wrk.format("POST","http://10.131.64.39/
8      pipeline-service/instances/", { collect_response = true})
9
10     x = wrk.format("POST","http://10.131.64.39/pipeline-service
11     /instances/"
12     + instance_id + "/runs/", { collect_response = true})
13     return x
14   end
15
16 response = function(status, headers, body)
17     runid = json.decode(body).runId
18 end
19
20 function saveRunIds(runIds)
21   local file,err = io.open("results.txt",'w')
22   if file then
23       file:write(tostring(score))
24       file:close()
25   else
26       print("error:", err)
27   end
28 end
29
30
```

```
31 done = function(summary, latency, requests)
32 end
33
```

D Implementierung: Aufbereitung der Antwortzeiten

```
1  #!/usr/bin/env python3
2  import os.path
3  import requests
4  from datetime import datetime
5  from datetime import timedelta
6  import time
7  import pandas as pd
8  import matplotlib.pyplot as plt
9  import numpy as np
10 import uuid
11 import seaborn as sns
12
13  #Copy Output to file
14 save_path = r"C:\Users\marco\IdeaProjects\locust-experiments\
15 kubernetes\results\ist_min\ist_min_1\raw_responses.csv"
16 save_path2 = r"C:\Users\marco\IdeaProjects\locust-experiments\
17 kubernetes\results\ist_min\ist_min_2\raw_responses.csv"
18 save_path3 = r"C:\Users\marco\IdeaProjects\locust-experiments\
19 kubernetes\results\ist_min\ist_min_3\raw_responses.csv"
20 savePathss = r"C:\Users\marco\IdeaProjects\locust-experiments\
21 kubernetes\results\ist_min"
22
23
24 #Safe Raw Data to CSV
25 df = pd.read_csv(save_path)
26 df2 = pd.read_csv(save_path2)
27 df3 = pd.read_csv(save_path3)
28
29
30 requests = pd.Series([len(df), len(df2), len(df3)]).mean()
31 failed_c = pd.Series([df['failed'].values[0],
32                       df2['failed'].values[0],
33                       df3['failed'].values[0]]).mean()
34
35 statstics = {'Requests': requests, 'Failed':failed_c}
```

```
36
37
38 df = df.set_index(pd.DatetimeIndex(df['Start Time']))
39 df2 = df2.set_index(pd.DatetimeIndex(df2['Start Time']))
40 df3 = df3.set_index(pd.DatetimeIndex(df3['Start Time']))
41
42
43 df = df.resample('5S').agg(
44     {'Response Time in Seconds':['sum', 'mean', 'count']})
45 df2 = df2.resample('5S').agg(
46     {'Response Time in Seconds':['sum', 'mean', 'count']})
47 df3 = df3.resample('5S').agg(
48     {'Response Time in Seconds':['sum', 'mean', 'count']})
49
50 x1 = df['Response Time in Seconds']['mean'].values
51 x2 = df2['Response Time in Seconds']['mean'].values
52 x3 = df3['Response Time in Seconds']['mean'].values
53 mean_series = []
54
55
56 list_x =[len(x1),len(x2),len(x3)]
57 mid = min(list_x)
58 for i in range(0,mid):
59     mean_Value = pd.Series([x1[i], x2[i], x3[i]]).mean()
60     print(mean_Value)
61     mean_series.insert(i,mean_Value)
62
63 statstics['mean'] = pd.Series(mean_series).mean()
64
65
66 df_test= pd.DataFrame({'y': pd.Series(np.arange(0,60,5))})
67
68 new_df = pd.DataFrame()
69 new_df['values'] = mean_series
70 try:
71     new_df = new_df.set_index(df_test['y'])
72 except:
73     df_test= pd.DataFrame({'y': pd.Series(np.arange(0,65,5))})
74     new_df = new_df.set_index(df_test['y'])
75
76
77 new_df = new_df[:-1]
78 new_df = new_df[1:]
79
80 new_df.to_csv(os.path.join(savePathss, "endresult.csv"),
```

```
81                              sep=",")
82
83  plt.plot(new_df.index,new_df['values'])
84  plt.xlabel('Zeit')
85  plt.ylabel('Durchschnittliche Antwortzeit in Sekunden')
86
87  plt.savefig(os.path.join(savePathss, "final_results"))
88
89  print(statstics)
90  df_statistics = pd.DataFrame([statstics])
91  df_statistics.to_csv(os.path.join(savePathss,"statistics.csv"),
92                                    sep=',')
93
94
```

E Ergebnisse: Vergleich der Architekturen

E.1 Testkonfiguration Minimum

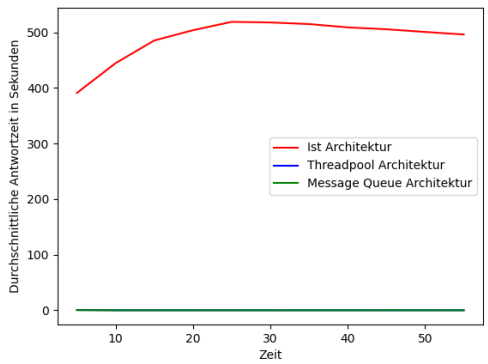

Abbildung 1: Ergebnisse Testkonfiguration Minimum Datensatz Kategorie „Klein“ (Cars)

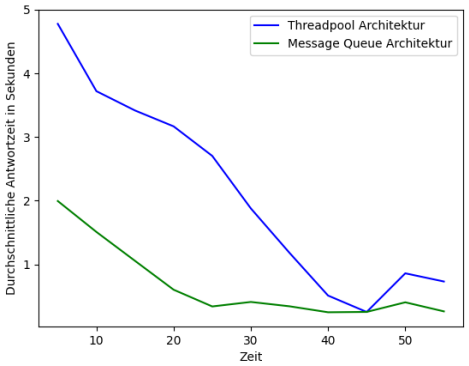

Abbildung 2: Ergebnisse Testkonfiguration Minimum Datensatz Kategorie „Mittel“ (Bookings)

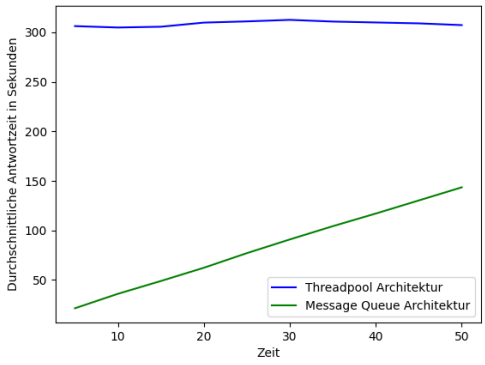

Abbildung 3: Ergebnisse Testkonfiguration Minimum Datensatz Kategorie „Groß“ (Reviews)

E.2 Testkonfiguration Medium

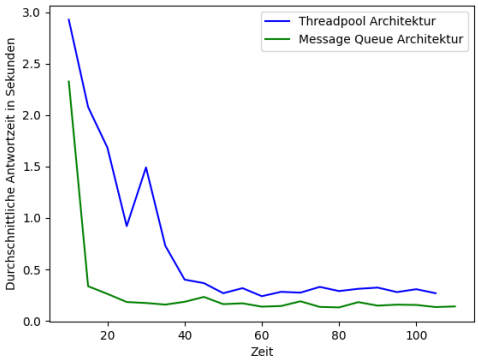

Abbildung 4: Ergebnisse Testkonfiguration Medium Datensatz Kategorie „Klein“ (Cars)

Abbildung 5: Ergebnisse Testkonfiguration Medium Datensatz Kategorie „Mittel“ (Bookings)

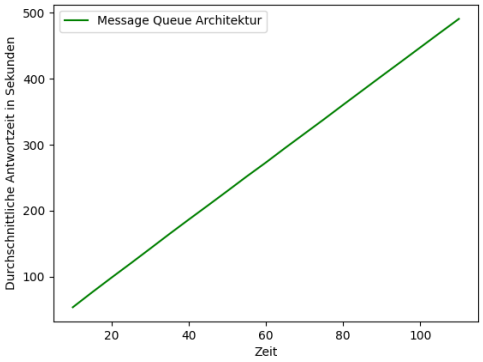

Abbildung 6: Ergebnisse Testkonfiguration Medium Datensatz Kategorie „Groß“ (Reviews)

E.3 Testkonfiguration Maximum

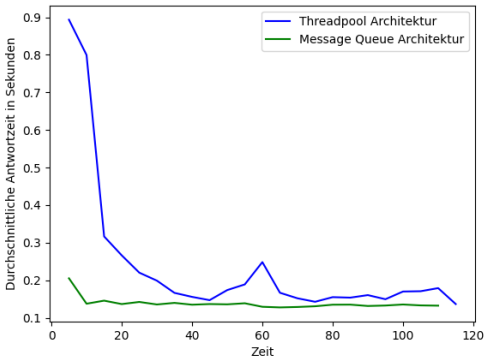

Abbildung 7: Ergebnisse Testkonfiguration Maximum Datensatz Kategorie „Klein“ (Cars)

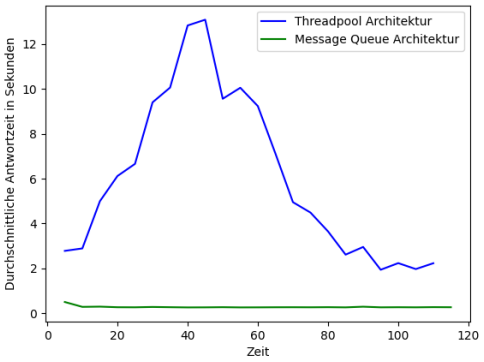

Abbildung 8: Ergebnisse Testkonfiguration Maximum Datensatz Kategorie „Mittel“ (Bookings)

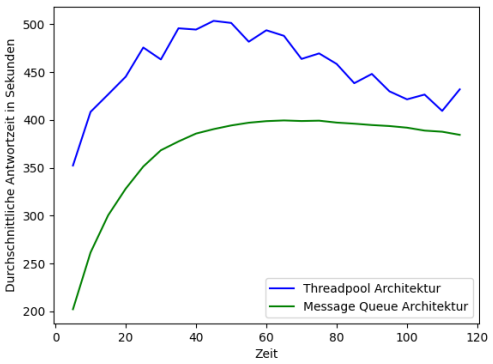

Abbildung 9: Ergebnisse Testkonfiguration Maximum Datensatz Kategorie „Groß“ (Reviews)

Literaturverzeichnis

Abdelfattah, A. S., & Cerny, T. (2022). Microservices Security Challenges and Approaches.

aws.amazon.com. (2024). AWS Docker [https://aws.amazon.com/de/docker/ [Accessed: 22.01.2024 9:30].

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, *14*(1), 70–93.

Chitra, L. P., & Satapathy, R. (2017). Performance comparison and evaluation of Node.js and traditional web server (IIS). *2017 International Conference on Algorithms, Methodology, Models and Applications in Emerging Technologies (ICAMMAET)*, 1–4. https://doi.org/10.1109/ICAMMAET.2017.8186633

Clark, M. (2018). How the BBC builds websites that scale [https://www.creativebloq.com/features/how-the-bbc-builds-websites-that-scale [Accessed: 14.03.2024 16:06].

Czeraszkiweicz, M. (2015). How to Benchmark HTTP Latency with wrk on Ubuntu 14.04 [https://www.digitalocean.com/community/tutorials/how-to-benchmark-http-latency-with-wrk-on-ubuntu-14-04# [Accessed: 28.01.2024 12:06].

Dossot, D. (2014). *RabbitMQ essentials*. Packt Publishing Ltd.

Gorton, I. (2006). *Essential software architecture*. Springer Science & Business Media.

Hong, X. J., Yang, H. S., & Kim, Y. H. (2018). Performance analysis of RESTful API and RabbitMQ for microservice web application. *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, 257–259.

Immonen, A., & Niemelä, E. (2008). Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software & Systems Modeling*, *7*, 49–65.

intersog. (2020). How do Do Load Testing for Web Applications with Locust [https://intersog.com/blog/load-testing-via-locust-framework/ [Accessed: 28.01.2024 11:45].

Jiang, Z. M., & Hassan, A. E. (2015). A survey on load testing of large-scale software systems. *IEEE Transactions on Software Engineering*, *41*(11), 1091–1118.

kinsta. (2023). What Is Node.js and Why You Should Use It [https://kinsta.com/knowledgebase/what-is-node-js/ [Accessed: 9.03.2024 18:39].

Locust. (2024). Locust [https://locust.io/ [Accessed: 24.01.2024 18:34].

Nasab, A. R., Shahin, M., Raviz, S. A. H., Liang, P., Mashmool, A., & Lenarduzzi, V. (2023). An empirical study of security practices for microservices systems. *Journal of Systems and Software*, *198*, 111563.

Newman, S. (2021). *Building microservices*. Ö'Reilly Media, Inc."

Node.js. (2024). Node.js [https://nodejs.org/en/about [Accessed: 20.01.2024 9:30].

Pargaonkar, S. (2023). A Comprehensive Review of Performance Testing Methodologies and Best Practices: Software Quality Engineering. *International Journal of Science and Research (IJSR)*, *12*(8), 2008–2014.

Pradeep, S., & Sharma, Y. K. (2019). A Pragmatic Evaluation of Stress and Performance Testing Technologies for Web Based Applications. *2019 Amity International Conference on Artificial Intelligence (AICAI)*, 399–403. https://doi.org/10.1109/AICAI.2019.8701327

Richardson, L., & Ruby, S. (2008). *RESTful web services*. Ö'Reilly Media, Inc."

Schabowsky, J. (2017). REST vs Messaging for Microservices - Which One is Best [https://solace.com/blog/experience-awesomeness-event-driven-microservices/ [Accessed: 8.02.2024 16:34].

Shah, J., & Dubaria, D. (2019). Building Modern Clouds: Using Docker, Kubernetes Google Cloud Platform. *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, 0184–0189. https://doi.org/10.1109/CCWC.2019.8666479

Shooman, M. L. (1984). Software reliability: A historical perspective. *IEEE Transactions on Reliability*, *R-33*(1), 48–55. https://doi.org/10.1109/TR.1984.6448274

SOPHISTen, D. (2022). MASTeR; Schablonen für alle Fälle. SOPHIST GmbH.

Thönes, J. (2015). Microservices. *IEEE software*, *32*(1), 116–116.

Tilkov, S., & Vinoski, S. (2010). Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, *14*(6), 80–83. https://doi.org/10.1109/MIC.2010.145

Wolff, E. (2018). *Microservices: Grundlagen flexibler Softwarearchitekturen*. dpunkt. verlag.

Xiong, X., & Fu, J. (2011). Active status certificate publish and subscribe based on AMQP. *2011 International Conference on Computational and Information Sciences*, 725–728.